Package ‘keras3’

April 18, 2024

Type Package

Title R Interface to 'Keras'

Version 0.2.0

Description Interface to 'Keras' <https://keras.io>, a high-level neural networks API. 'Keras' was developed with a focus on enabling fast experimentation, supports both convolution based networks and recurrent networks (as well as combinations of the two), and runs seamlessly on both CPU and GPU devices.

Encoding UTF-8

License MIT + file LICENSE

URL https://keras.posit.co/, https://github.com/rstudio/keras

BugReports https://github.com/rstudio/keras/issues

Depends R (>= 4.0)

Imports generics (>= 0.0.1), reticulate (>= 1.35.0.9000), tensorflow (>= 2.15.0.9000), tfruns (>= 1.5.2), magrittr, zeallot, fastmap, glue, cli, rlang

Suggests ggplot2, testthat (>= 2.1.0), knitr, rmarkdown, callr, tfdatasets, withr, png, jsonlite, purrr, rstudioapi, R6, jpeg

RoxygenNote 7.3.1

VignetteBuilder knitr

NeedsCompilation no

Author Tomasz Kalinowski [aut, cph, cre],
Daniel Falbel [ctb, cph],
JJ Allaire [aut, cph],
François Chollet [aut, cph],
Posit Software, PBC [cph, fnd],
Google [cph, fnd],
Yuan Tang [ctb, cph] (<https://orcid.org/0000-0001-5243-233X>),
Wouter Van Der Bijl [ctb, cph],
Martin Studer [ctb, cph],
Sigrid Keydana [ctb]

Maintainer Tomasz Kalinowski <tomasz@posit.co>

Repository CRAN

Date/Publication 2024-04-18 13:22:40 UTC

R topics documented:

`activation_elu` *Exponential Linear Unit.*

Description

The exponential linear unit (ELU) with `alpha > 0` is defined as:

- `x` if `x > 0`
- `alpha * exp(x) - 1` if `x < 0`

ELUs have negative values which pushes the mean of the activations closer to zero.

Mean activations that are closer to zero enable faster learning as they bring the gradient closer to the natural gradient. ELUs saturate to a negative value when the argument gets smaller. Saturation means a small derivative which decreases the variation and the information that is propagated to the next layer.

Usage

```
activation_elu(x, alpha = 1)
```

Arguments

`x`	Input tensor.
`alpha`	Numeric. See description for details.

Value

A tensor, the result from applying the activation to the input tensor `x`.

Reference

- Clevert et al., 2016

See Also

- `https://keras.io/api/layers/activations#elu-function`

Other activations:
`activation_exponential()`
`activation_gelu()`
`activation_hard_sigmoid()`
`activation_leaky_relu()`
`activation_linear()`
`activation_log_softmax()`
`activation_mish()`
`activation_relu()`
`activation_relu6()`
`activation_selu()`
`activation_sigmoid()`
`activation_silu()`
`activation_softmax()`
`activation_softplus()`
`activation_softsign()`
`activation_tanh()`

`activation_exponential` *Exponential activation function.*

Description

Exponential activation function.

Usage

```
activation_exponential(x)
```

Arguments

x	Input tensor.

Value

A tensor, the result from applying the activation to the input tensor `x`.

See Also

- `https://keras.io/api/layers/activations#exponential-function`

Other activations:
`activation_elu()`
`activation_gelu()`
`activation_hard_sigmoid()`
`activation_leaky_relu()`
`activation_linear()`
`activation_log_softmax()`
`activation_mish()`
`activation_relu()`
`activation_relu6()`
`activation_selu()`
`activation_sigmoid()`
`activation_silu()`
`activation_softmax()`
`activation_softplus()`
`activation_softsign()`
`activation_tanh()`

`activation_gelu` *Gaussian error linear unit (GELU) activation function.*

Description

The Gaussian error linear unit (GELU) is defined as:

`gelu(x) = x * P(X <= x)` where `P(X) ~ N(0, 1)`, i.e. `gelu(x) = 0.5 * x * (1 + erf(x / sqrt(2)))`.

GELU weights inputs by their value, rather than gating inputs by their sign as in ReLU.

Usage

```
activation_gelu(x, approximate = FALSE)
```

Arguments

`x`	Input tensor.
`approximate`	A `bool`, whether to enable approximation.

Value

A tensor, the result from applying the activation to the input tensor `x`.

Reference

- Hendrycks et al., 2016

See Also

- `https://keras.io/api/layers/activations#gelu-function`

Other activations:
`activation_elu()`
`activation_exponential()`
`activation_hard_sigmoid()`
`activation_leaky_relu()`
`activation_linear()`
`activation_log_softmax()`
`activation_mish()`
`activation_relu()`
`activation_relu6()`
`activation_selu()`
`activation_sigmoid()`
`activation_silu()`
`activation_softmax()`
`activation_softplus()`
`activation_softsign()`
`activation_tanh()`

`activation_hard_sigmoid`

Hard sigmoid activation function.

Description

The hard sigmoid activation is defined as:

- `0` if `if x <= -3`
- `1` if `x >= 3`
- `(x/6) + 0.5` if `-3 < x < 3`

It's a faster, piecewise linear approximation of the sigmoid activation.

Usage

```
activation_hard_sigmoid(x)
```

Arguments

x	Input tensor.

Value

A tensor, the result from applying the activation to the input tensor x.

Reference

- Wikipedia "Hard sigmoid"

See Also

- https://keras.io/api/layers/activations#hardsigmoid-function

Other activations:
activation_elu()
activation_exponential()
activation_gelu()
activation_leaky_relu()
activation_linear()
activation_log_softmax()
activation_mish()
activation_relu()
activation_relu6()
activation_selu()
activation_sigmoid()
activation_silu()
activation_softmax()
activation_softplus()
activation_softsign()
activation_tanh()

activation_hard_silu *Hard SiLU activation function, also known as Hard Swish.*

Description

It is defined as:

- 0 if if x < -3
- x if x > 3
- x * (x + 3) / 6 if -3 <= x <= 3

It's a faster, piecewise linear approximation of the silu activation.

Usage

```
activation_hard_silu(x)

activation_hard_swish(x)
```

Arguments

`x`	Input tensor.

Value

A tensor, the result from applying the activation to the input tensor `x`.

Reference

- A Howard, 2019

`activation_leaky_relu` *Leaky relu activation function.*

Description

Leaky relu activation function.

Usage

```
activation_leaky_relu(x, negative_slope = 0.2)
```

Arguments

`x`	Input tensor.
`negative_slope`	A `float` that controls the slope for values lower than the threshold.

Value

A tensor, the result from applying the activation to the input tensor `x`.

See Also

- `https://keras.io/api/layers/activations#leakyrelu-function`

Other activations:
`activation_elu()`
`activation_exponential()`
`activation_gelu()`
`activation_hard_sigmoid()`
`activation_linear()`
`activation_log_softmax()`

`activation_mish()`
`activation_relu()`
`activation_relu6()`
`activation_selu()`
`activation_sigmoid()`
`activation_silu()`
`activation_softmax()`
`activation_softplus()`
`activation_softsign()`
`activation_tanh()`

`activation_linear` *Linear activation function (pass-through).*

Description

A "linear" activation is an identity function: it returns the input, unmodified.

Usage

```
activation_linear(x)
```

Arguments

`x`	Input tensor.

Value

A tensor, the result from applying the activation to the input tensor `x`.

See Also

- https://keras.io/api/layers/activations#linear-function

Other activations:
`activation_elu()`
`activation_exponential()`
`activation_gelu()`
`activation_hard_sigmoid()`
`activation_leaky_relu()`
`activation_log_softmax()`
`activation_mish()`
`activation_relu()`
`activation_relu6()`
`activation_selu()`
`activation_sigmoid()`
`activation_silu()`

activation_softmax()
activation_softplus()
activation_softsign()
activation_tanh()

activation_log_softmax

Log-Softmax activation function.

Description

Each input vector is handled independently. The axis argument sets which axis of the input the function is applied along.

Usage

```
activation_log_softmax(x, axis = -1L)
```

Arguments

x	Input tensor.
axis	Integer, axis along which the softmax is applied.

Value

A tensor, the result from applying the activation to the input tensor x.

See Also

- https://keras.io/api/layers/activations#logsoftmax-function

Other activations:
activation_elu()
activation_exponential()
activation_gelu()
activation_hard_sigmoid()
activation_leaky_relu()
activation_linear()
activation_mish()
activation_relu()
activation_relu6()
activation_selu()
activation_sigmoid()
activation_silu()
activation_softmax()
activation_softplus()
activation_softsign()

activation_tanh()

activation_mish *Mish activation function.*

Description

It is defined as:

`mish(x) = x * tanh(softplus(x))`

where `softplus` is defined as:

`softplus(x) = log(exp(x) + 1)`

Usage

```
activation_mish(x)
```

Arguments

x	Input tensor.

Value

A tensor, the result from applying the activation to the input tensor `x`.

Reference

- Misra, 2019

See Also

- https://keras.io/api/layers/activations#mish-function

Other activations:
activation_elu()
activation_exponential()
activation_gelu()
activation_hard_sigmoid()
activation_leaky_relu()
activation_linear()
activation_log_softmax()
activation_relu()
activation_relu6()
activation_selu()
activation_sigmoid()
activation_silu()
activation_softmax()

activation_softplus()
activation_softsign()
activation_tanh()

`activation_relu` *Applies the rectified linear unit activation function.*

Description

With default values, this returns the standard ReLU activation: `max(x, 0)`, the element-wise maximum of 0 and the input tensor.

Modifying default parameters allows you to use non-zero thresholds, change the max value of the activation, and to use a non-zero multiple of the input for values below the threshold.

Usage

```
activation_relu(x, negative_slope = 0, max_value = NULL, threshold = 0)
```

Arguments

`x`	Input tensor.
`negative_slope`	A `numeric` that controls the slope for values lower than the threshold.
`max_value`	A `numeric` that sets the saturation threshold (the largest value the function will return).
`threshold`	A `numeric` giving the threshold value of the activation function below which values will be damped or set to zero.

Value

A tensor with the same shape and dtype as input `x`.

Examples

```
x <- c(-10, -5, 0, 5, 10)
activation_relu(x)

## tf.Tensor([ 0.  0.  0.  5. 10.], shape=(5), dtype=float32)


activation_relu(x, negative_slope = 0.5)

## tf.Tensor([-5.  -2.5  0.   5.  10. ], shape=(5), dtype=float32)


activation_relu(x, max_value = 5)
```

```
## tf.Tensor([0. 0. 0. 5. 5.], shape=(5), dtype=float32)


activation_relu(x, threshold = 5)

## tf.Tensor([-0. -0.  0.  0. 10.], shape=(5), dtype=float32)
```

See Also

- https://keras.io/api/layers/activations#relu-function

Other activations:
activation_elu()
activation_exponential()
activation_gelu()
activation_hard_sigmoid()
activation_leaky_relu()
activation_linear()
activation_log_softmax()
activation_mish()
activation_relu6()
activation_selu()
activation_sigmoid()
activation_silu()
activation_softmax()
activation_softplus()
activation_softsign()
activation_tanh()

---

activation_relu6 *Relu6 activation function.*

Description

It’s the ReLU function, but truncated to a maximum value of 6.

Usage

```
activation_relu6(x)
```

Arguments

x	Input tensor.

Value

A tensor, the result from applying the activation to the input tensor x.

See Also

- https://keras.io/api/layers/activations#relu6-function

Other activations:
activation_elu()
activation_exponential()
activation_gelu()
activation_hard_sigmoid()
activation_leaky_relu()
activation_linear()
activation_log_softmax()
activation_mish()
activation_relu()
activation_selu()
activation_sigmoid()
activation_silu()
activation_softmax()
activation_softplus()
activation_softsign()
activation_tanh()

---

activation_selu *Scaled Exponential Linear Unit (SELU).*

Description

The Scaled Exponential Linear Unit (SELU) activation function is defined as:

- `scale * x` if `x > 0`
- `scale * alpha * (exp(x) - 1)` if `x < 0`

where `alpha` and `scale` are pre-defined constants (`alpha = 1.67326324` and `scale = 1.05070098`).

Basically, the SELU activation function multiplies `scale` (> 1) with the output of the `activation_elu` function to ensure a slope larger than one for positive inputs.

The values of `alpha` and `scale` are chosen so that the mean and variance of the inputs are preserved between two consecutive layers as long as the weights are initialized correctly (see `initializer_lecun_normal()`) and the number of input units is "large enough" (see reference paper for more information).

Usage

```
activation_selu(x)
```

Arguments

x	Input tensor.

Value

A tensor, the result from applying the activation to the input tensor `x`.

Notes

- To be used together with `initializer_lecun_normal()`.
- To be used together with the dropout variant `layer_alpha_dropout()` (legacy, depracated).

Reference

- Klambauer et al., 2017

See Also

- `https://keras.io/api/layers/activations#selu-function`

Other activations:
`activation_elu()`
`activation_exponential()`
`activation_gelu()`
`activation_hard_sigmoid()`
`activation_leaky_relu()`
`activation_linear()`
`activation_log_softmax()`
`activation_mish()`
`activation_relu()`
`activation_relu6()`
`activation_sigmoid()`
`activation_silu()`
`activation_softmax()`
`activation_softplus()`
`activation_softsign()`
`activation_tanh()`

`activation_sigmoid` *Sigmoid activation function.*

Description

It is defined as: `sigmoid(x) = 1 / (1 + exp(-x))`.

For small values (<-5), `sigmoid` returns a value close to zero, and for large values (>5) the result of the function gets close to 1.

Sigmoid is equivalent to a 2-element softmax, where the second element is assumed to be zero. The sigmoid function always returns a value between 0 and 1.

Usage

```
activation_sigmoid(x)
```

Arguments

x	Input tensor.

Value

A tensor, the result from applying the activation to the input tensor `x`.

See Also

- https://keras.io/api/layers/activations#sigmoid-function

Other activations:
activation_elu()
activation_exponential()
activation_gelu()
activation_hard_sigmoid()
activation_leaky_relu()
activation_linear()
activation_log_softmax()
activation_mish()
activation_relu()
activation_relu6()
activation_selu()
activation_silu()
activation_softmax()
activation_softplus()
activation_softsign()
activation_tanh()

activation_silu *Swish (or Silu) activation function.*

Description

It is defined as: `swish(x) = x * sigmoid(x)`.

The Swish (or Silu) activation function is a smooth, non-monotonic function that is unbounded above and bounded below.

Usage

```
activation_silu(x)
```

Arguments

x	Input tensor.

Value

A tensor, the result from applying the activation to the input tensor x.

Reference

- Ramachandran et al., 2017

See Also

- https://keras.io/api/layers/activations#silu-function

Other activations:
activation_elu()
activation_exponential()
activation_gelu()
activation_hard_sigmoid()
activation_leaky_relu()
activation_linear()
activation_log_softmax()
activation_mish()
activation_relu()
activation_relu6()
activation_selu()
activation_sigmoid()
activation_softmax()
activation_softplus()
activation_softsign()
activation_tanh()

---

activation_softmax *Softmax converts a vector of values to a probability distribution.*

Description

The elements of the output vector are in range `[0, 1]` and sum to 1.

Each input vector is handled independently. The `axis` argument sets which axis of the input the function is applied along.

Softmax is often used as the activation for the last layer of a classification network because the result could be interpreted as a probability distribution.

The softmax of each vector x is computed as `exp(x) / sum(exp(x))`.

The input values in are the log-odds of the resulting probability.

Usage

```
activation_softmax(x, axis = -1L)
```

Arguments

`x`	Input tensor.
`axis`	Integer, axis along which the softmax is applied.

Value

A tensor, the result from applying the activation to the input tensor `x`.

See Also

- https://keras.io/api/layers/activations#softmax-function

Other activations:
`activation_elu()`
`activation_exponential()`
`activation_gelu()`
`activation_hard_sigmoid()`
`activation_leaky_relu()`
`activation_linear()`
`activation_log_softmax()`
`activation_mish()`
`activation_relu()`
`activation_relu6()`
`activation_selu()`
`activation_sigmoid()`
`activation_silu()`
`activation_softplus()`
`activation_softsign()`
`activation_tanh()`

`activation_softplus` *Softplus activation function.*

Description

It is defined as: `softplus(x) = log(exp(x) + 1)`.

Usage

```
activation_softplus(x)
```

Arguments

`x`	Input tensor.

Value

A tensor, the result from applying the activation to the input tensor `x`.

See Also

- `https://keras.io/api/layers/activations#softplus-function`

Other activations:
`activation_elu()`
`activation_exponential()`
`activation_gelu()`
`activation_hard_sigmoid()`
`activation_leaky_relu()`
`activation_linear()`
`activation_log_softmax()`
`activation_mish()`
`activation_relu()`
`activation_relu6()`
`activation_selu()`
`activation_sigmoid()`
`activation_silu()`
`activation_softmax()`
`activation_softsign()`
`activation_tanh()`

`activation_softsign` *Softsign activation function.*

Description

Softsign is defined as: `softsign(x) = x / (abs(x) + 1)`.

Usage

```
activation_softsign(x)
```

Arguments

`x`	Input tensor.

Value

A tensor, the result from applying the activation to the input tensor `x`.

See Also

- https://keras.io/api/layers/activations#softsign-function

Other activations:
`activation_elu()`
`activation_exponential()`
`activation_gelu()`
`activation_hard_sigmoid()`
`activation_leaky_relu()`
`activation_linear()`
`activation_log_softmax()`
`activation_mish()`
`activation_relu()`
`activation_relu6()`
`activation_selu()`
`activation_sigmoid()`
`activation_silu()`
`activation_softmax()`
`activation_softplus()`
`activation_tanh()`

`activation_tanh` *Hyperbolic tangent activation function.*

Description

It is defined as: `tanh(x) = sinh(x) / cosh(x)`, i.e. `tanh(x) = ((exp(x) - exp(-x)) / (exp(x) + exp(-x)))`.

Usage

```
activation_tanh(x)
```

Arguments

`x`	Input tensor.

Value

A tensor, the result from applying the activation to the input tensor `x`.

See Also

- https://keras.io/api/layers/activations#tanh-function

Other activations:
`activation_elu()`

activation_exponential()
activation_gelu()
activation_hard_sigmoid()
activation_leaky_relu()
activation_linear()
activation_log_softmax()
activation_mish()
activation_relu()
activation_relu6()
activation_selu()
activation_sigmoid()
activation_silu()
activation_softmax()
activation_softplus()
activation_softsign()

`active_property` *Create an active property class method*

Description

Create an active property class method

Usage

```
active_property(fn)
```

Arguments

`fn`	An R function

Value

`fn`, with an additional R attribute that will cause `fn` to be converted to an active property when being converted to a method of a custom subclass.

Example

```
layer_foo <- Model("Foo", ...,
  metrics = active_property(function() {
    list(self$d_loss_metric,
         self$g_loss_metric)
  }))
```

adapt — *Fits the state of the preprocessing layer to the data being passed*

Description

Fits the state of the preprocessing layer to the data being passed

Usage

```
adapt(object, data, ..., batch_size = NULL, steps = NULL)
```

Arguments

`object`	Preprocessing layer object
`data`	The data to train on. It can be passed either as a `tf.data.Dataset` or as an R array.
`...`	Used for forwards and backwards compatibility. Passed on to the underlying method.
`batch_size`	Integer or `NULL`. Number of asamples per state update. If unspecified, `batch_size` will default to `32`. Do not specify the batch_size if your data is in the form of a TF Dataset or a generator (since they generate batches).
`steps`	Integer or `NULL`. Total number of steps (batches of samples) When training with input tensors such as TensorFlow data tensors, the default `NULL` is equal to the number of samples in your dataset divided by the batch size, or `1` if that cannot be determined. If x is a `tf.data.Dataset`, and `steps` is `NULL`, the epoch will run until the input dataset is exhausted. When passing an infinitely repeating dataset, you must specify the steps argument. This argument is not supported with array inputs.

Details

After calling `adapt` on a layer, a preprocessing layer's state will not update during training. In order to make preprocessing layers efficient in any distribution context, they are kept constant with respect to any compiled `tf.Graph`s that call the layer. This does not affect the layer use when adapting each layer only once, but if you adapt a layer multiple times you will need to take care to re-compile any compiled functions as follows:

- If you are adding a preprocessing layer to a keras model, you need to call `compile(model)` after each subsequent call to `adapt()`.
- If you are calling a preprocessing layer inside `tfdatasets::dataset_map()`, you should call `dataset_map()` again on the input `Dataset` after each `adapt()`.
- If you are using a `tensorflow::tf_function()` directly which calls a preprocessing layer, you need to call `tf_function()` again on your callable after each subsequent call to `adapt()`.

`keras_model()` example with multiple adapts:

```
layer <- layer_normalization(axis = NULL)
adapt(layer, c(0, 2))
model <- keras_model_sequential() |> layer()
predict(model, c(0, 1, 2), verbose = FALSE) # [1] -1  0  1

## [1] -1  0  1


adapt(layer, c(-1, 1))
compile(model)  # This is needed to re-compile model.predict!
predict(model, c(0, 1, 2), verbose = FALSE) # [1] 0 1 2

## [1] 0 1 2
```

`tfdatasets` example with multiple adapts:

```
layer <- layer_normalization(axis = NULL)
adapt(layer, c(0, 2))
input_ds <- tfdatasets::range_dataset(0, 3)
normalized_ds <- input_ds |>
  tfdatasets::dataset_map(layer)
str(tfdatasets::iterate(normalized_ds))

## List of 3
##  $ :<tf.Tensor: shape=(1), dtype=float32, numpy=array([-1.], dtype=float32)>
##  $ :<tf.Tensor: shape=(1), dtype=float32, numpy=array([0.], dtype=float32)>
##  $ :<tf.Tensor: shape=(1), dtype=float32, numpy=array([1.], dtype=float32)>


adapt(layer, c(-1, 1))
normalized_ds <- input_ds |>
  tfdatasets::dataset_map(layer) # Re-map over the input dataset.

normalized_ds |>
  tfdatasets::as_array_iterator() |>
  tfdatasets::iterate(simplify = FALSE) |>
  str()

## List of 3
##  $ : num [1(1d)] 0
##  $ : num [1(1d)] 1
##  $ : num [1(1d)] 2
```

Value

Returns `object`, invisibly.

application_convnext_base

Instantiates the ConvNeXtBase architecture.

Description

Instantiates the ConvNeXtBase architecture.

Usage

```
application_convnext_base(
  model_name = "convnext_base",
  include_top = TRUE,
  include_preprocessing = TRUE,
  weights = "imagenet",
  input_tensor = NULL,
  input_shape = NULL,
  pooling = NULL,
  classes = 1000L,
  classifier_activation = "softmax"
)
```

Arguments

`model_name`	String, name for the model.
`include_top`	Whether to include the fully-connected layer at the top of the network. Defaults to `TRUE`.
`include_preprocessing`	Boolean, whether to include the preprocessing layer at the bottom of the network.
`weights`	One of `NULL` (random initialization), `"imagenet"` (pre-training on ImageNet-1k), or the path to the weights file to be loaded. Defaults to `"imagenet"`.
`input_tensor`	Optional Keras tensor (i.e. output of `layers.Input()`) to use as image input for the model.
`input_shape`	Optional shape tuple, only to be specified if `include_top` is `FALSE`. It should have exactly 3 inputs channels.
`pooling`	Optional pooling mode for feature extraction when `include_top` is `FALSE`. Defaults to `NULL`. • `NULL` means that the output of the model will be the 4D tensor output of the last convolutional layer. • `avg` means that global average pooling will be applied to the output of the last convolutional layer, and thus the output of the model will be a 2D tensor. • `max` means that global max pooling will be applied.

`classes` Optional number of classes to classify images into, only to be specified if `include_top` is `TRUE`, and if no `weights` argument is specified. Defaults to 1000 (number of ImageNet classes).

`classifier_activation`
A `str` or callable. The activation function to use on the "top" layer. Ignored unless `include_top=TRUE`. Set `classifier_activation=NULL` to return the logits of the "top" layer. Defaults to `"softmax"`. When loading pretrained weights, `classifier_activation` can only be `NULL` or `"softmax"`.

Value

A model instance.

References

- A ConvNet for the 2020s (CVPR 2022)

For image classification use cases, see this page for detailed examples. For transfer learning use cases, make sure to read the guide to transfer learning & fine-tuning.

The `base`, `large`, and `xlarge` models were first pre-trained on the ImageNet-21k dataset and then fine-tuned on the ImageNet-1k dataset. The pre-trained parameters of the models were assembled from the official repository. To get a sense of how these parameters were converted to Keras compatible parameters, please refer to this repository.

Note

Each Keras Application expects a specific kind of input preprocessing. For ConvNeXt, preprocessing is included in the model using a `Normalization` layer. ConvNeXt models expect their inputs to be float or uint8 tensors of pixels with values in the `[0-255]` range.

When calling the `summary()` method after instantiating a ConvNeXt model, prefer setting the `expand_nested` argument `summary()` to `TRUE` to better investigate the instantiated model.

See Also

- `https://keras.io/api/applications/convnext#convnextbase-function`

`application_convnext_large`
Instantiates the ConvNeXtLarge architecture.

Description

Instantiates the ConvNeXtLarge architecture.

Usage

```
application_convnext_large(
  model_name = "convnext_large",
  include_top = TRUE,
  include_preprocessing = TRUE,
  weights = "imagenet",
  input_tensor = NULL,
  input_shape = NULL,
  pooling = NULL,
  classes = 1000L,
  classifier_activation = "softmax"
)
```

Arguments

`model_name`	String, name for the model.
`include_top`	Whether to include the fully-connected layer at the top of the network. Defaults to `TRUE`.
`include_preprocessing`	Boolean, whether to include the preprocessing layer at the bottom of the network.
`weights`	One of `NULL` (random initialization), `"imagenet"` (pre-training on ImageNet-1k), or the path to the weights file to be loaded. Defaults to `"imagenet"`.
`input_tensor`	Optional Keras tensor (i.e. output of `layers.Input()`) to use as image input for the model.
`input_shape`	Optional shape tuple, only to be specified if `include_top` is `FALSE`. It should have exactly 3 inputs channels.
`pooling`	Optional pooling mode for feature extraction when `include_top` is `FALSE`. Defaults to `NULL`. • `NULL` means that the output of the model will be the 4D tensor output of the last convolutional layer. • `avg` means that global average pooling will be applied to the output of the last convolutional layer, and thus the output of the model will be a 2D tensor. • `max` means that global max pooling will be applied.
`classes`	Optional number of classes to classify images into, only to be specified if `include_top` is `TRUE`, and if no `weights` argument is specified. Defaults to 1000 (number of ImageNet classes).
`classifier_activation`	A `str` or callable. The activation function to use on the "top" layer. Ignored unless `include_top=TRUE`. Set `classifier_activation=NULL` to return the logits of the "top" layer. Defaults to `"softmax"`. When loading pretrained weights, `classifier_activation` can only be `NULL` or `"softmax"`.

Value

A model instance.

References

- A ConvNet for the 2020s (CVPR 2022)

For image classification use cases, see this page for detailed examples. For transfer learning use cases, make sure to read the guide to transfer learning & fine-tuning.

The `base`, `large`, and `xlarge` models were first pre-trained on the ImageNet-21k dataset and then fine-tuned on the ImageNet-1k dataset. The pre-trained parameters of the models were assembled from the official repository. To get a sense of how these parameters were converted to Keras compatible parameters, please refer to this repository.

Note

Each Keras Application expects a specific kind of input preprocessing. For ConvNeXt, preprocessing is included in the model using a `Normalization` layer. ConvNeXt models expect their inputs to be float or uint8 tensors of pixels with values in the `[0-255]` range.

When calling the `summary()` method after instantiating a ConvNeXt model, prefer setting the `expand_nested` argument `summary()` to `TRUE` to better investigate the instantiated model.

See Also

- https://keras.io/api/applications/convnext#convnextlarge-function

---

application_convnext_small

Instantiates the ConvNeXtSmall architecture.

Description

Instantiates the ConvNeXtSmall architecture.

Usage

```
application_convnext_small(
  model_name = "convnext_small",
  include_top = TRUE,
  include_preprocessing = TRUE,
  weights = "imagenet",
  input_tensor = NULL,
  input_shape = NULL,
  pooling = NULL,
  classes = 1000L,
  classifier_activation = "softmax"
)
```

Arguments

`model_name`	String, name for the model.
`include_top`	Whether to include the fully-connected layer at the top of the network. Defaults to `TRUE`.
`include_preprocessing`	Boolean, whether to include the preprocessing layer at the bottom of the network.
`weights`	One of `NULL` (random initialization), `"imagenet"` (pre-training on ImageNet-1k), or the path to the weights file to be loaded. Defaults to `"imagenet"`.
`input_tensor`	Optional Keras tensor (i.e. output of `layers.Input()`) to use as image input for the model.
`input_shape`	Optional shape tuple, only to be specified if `include_top` is `FALSE`. It should have exactly 3 inputs channels.
`pooling`	Optional pooling mode for feature extraction when `include_top` is `FALSE`. Defaults to `NULL`. • `NULL` means that the output of the model will be the 4D tensor output of the last convolutional layer. • `avg` means that global average pooling will be applied to the output of the last convolutional layer, and thus the output of the model will be a 2D tensor. • `max` means that global max pooling will be applied.
`classes`	Optional number of classes to classify images into, only to be specified if `include_top` is `TRUE`, and if no `weights` argument is specified. Defaults to 1000 (number of ImageNet classes).
`classifier_activation`	A `str` or callable. The activation function to use on the "top" layer. Ignored unless `include_top=TRUE`. Set `classifier_activation=NULL` to return the logits of the "top" layer. Defaults to `"softmax"`. When loading pretrained weights, `classifier_activation` can only be `NULL` or `"softmax"`.

Value

A model instance.

References

- A ConvNet for the 2020s (CVPR 2022)

For image classification use cases, see this page for detailed examples. For transfer learning use cases, make sure to read the guide to transfer learning & fine-tuning.

The `base`, `large`, and `xlarge` models were first pre-trained on the ImageNet-21k dataset and then fine-tuned on the ImageNet-1k dataset. The pre-trained parameters of the models were assembled from the official repository. To get a sense of how these parameters were converted to Keras compatible parameters, please refer to this repository.

Note

Each Keras Application expects a specific kind of input preprocessing. For ConvNeXt, preprocessing is included in the model using a `Normalization` layer. ConvNeXt models expect their inputs to be float or uint8 tensors of pixels with values in the `[0-255]` range.

When calling the `summary()` method after instantiating a ConvNeXt model, prefer setting the `expand_nested` argument `summary()` to `TRUE` to better investigate the instantiated model.

See Also

- https://keras.io/api/applications/convnext#convnextsmall-function

---

`application_convnext_tiny`

Instantiates the ConvNeXtTiny architecture.

Description

Instantiates the ConvNeXtTiny architecture.

Usage

```
application_convnext_tiny(
  model_name = "convnext_tiny",
  include_top = TRUE,
  include_preprocessing = TRUE,
  weights = "imagenet",
  input_tensor = NULL,
  input_shape = NULL,
  pooling = NULL,
  classes = 1000L,
  classifier_activation = "softmax"
)
```

Arguments

`model_name`	String, name for the model.
`include_top`	Whether to include the fully-connected layer at the top of the network. Defaults to `TRUE`.
`include_preprocessing`	Boolean, whether to include the preprocessing layer at the bottom of the network.
`weights`	One of `NULL` (random initialization), `"imagenet"` (pre-training on ImageNet-1k), or the path to the weights file to be loaded. Defaults to `"imagenet"`.
`input_tensor`	Optional Keras tensor (i.e. output of `layers.Input()`) to use as image input for the model.

input_shape	Optional shape tuple, only to be specified if include_top is FALSE. It should have exactly 3 inputs channels.
pooling	Optional pooling mode for feature extraction when include_top is FALSE. Defaults to NULL. • NULL means that the output of the model will be the 4D tensor output of the last convolutional layer. • avg means that global average pooling will be applied to the output of the last convolutional layer, and thus the output of the model will be a 2D tensor. • max means that global max pooling will be applied.
classes	Optional number of classes to classify images into, only to be specified if include_top is TRUE, and if no weights argument is specified. Defaults to 1000 (number of ImageNet classes).
classifier_activation	A str or callable. The activation function to use on the "top" layer. Ignored unless include_top=TRUE. Set classifier_activation=NULL to return the logits of the "top" layer. Defaults to "softmax". When loading pretrained weights, classifier_activation can only be NULL or "softmax".

Value

A model instance.

References

- A ConvNet for the 2020s (CVPR 2022)

For image classification use cases, see this page for detailed examples. For transfer learning use cases, make sure to read the guide to transfer learning & fine-tuning.

The base, large, and xlarge models were first pre-trained on the ImageNet-21k dataset and then fine-tuned on the ImageNet-1k dataset. The pre-trained parameters of the models were assembled from the official repository. To get a sense of how these parameters were converted to Keras compatible parameters, please refer to this repository.

Note

Each Keras Application expects a specific kind of input preprocessing. For ConvNeXt, preprocessing is included in the model using a Normalization layer. ConvNeXt models expect their inputs to be float or uint8 tensors of pixels with values in the [0-255] range.

When calling the summary() method after instantiating a ConvNeXt model, prefer setting the expand_nested argument summary() to TRUE to better investigate the instantiated model.

See Also

- https://keras.io/api/applications/convnext#convnexttiny-function

application_convnext_xlarge

Instantiates the ConvNeXtXLarge architecture.

Description

Instantiates the ConvNeXtXLarge architecture.

Usage

```
application_convnext_xlarge(
  model_name = "convnext_xlarge",
  include_top = TRUE,
  include_preprocessing = TRUE,
  weights = "imagenet",
  input_tensor = NULL,
  input_shape = NULL,
  pooling = NULL,
  classes = 1000L,
  classifier_activation = "softmax"
)
```

Arguments

`model_name`	String, name for the model.
`include_top`	Whether to include the fully-connected layer at the top of the network. Defaults to `TRUE`.
`include_preprocessing`	Boolean, whether to include the preprocessing layer at the bottom of the network.
`weights`	One of `NULL` (random initialization), `"imagenet"` (pre-training on ImageNet-1k), or the path to the weights file to be loaded. Defaults to `"imagenet"`.
`input_tensor`	Optional Keras tensor (i.e. output of `layers.Input()`) to use as image input for the model.
`input_shape`	Optional shape tuple, only to be specified if `include_top` is `FALSE`. It should have exactly 3 inputs channels.
`pooling`	Optional pooling mode for feature extraction when `include_top` is `FALSE`. Defaults to `NULL`. • `NULL` means that the output of the model will be the 4D tensor output of the last convolutional layer. • `avg` means that global average pooling will be applied to the output of the last convolutional layer, and thus the output of the model will be a 2D tensor. • `max` means that global max pooling will be applied.

`classes`	Optional number of classes to classify images into, only to be specified if `include_top` is `TRUE`, and if no `weights` argument is specified. Defaults to 1000 (number of ImageNet classes).
`classifier_activation`	A `str` or callable. The activation function to use on the "top" layer. Ignored unless `include_top=TRUE`. Set `classifier_activation=NULL` to return the logits of the "top" layer. Defaults to `"softmax"`. When loading pretrained weights, `classifier_activation` can only be `NULL` or `"softmax"`.

Value

A model instance.

References

- A ConvNet for the 2020s (CVPR 2022)

For image classification use cases, see this page for detailed examples. For transfer learning use cases, make sure to read the guide to transfer learning & fine-tuning.

The `base`, `large`, and `xlarge` models were first pre-trained on the ImageNet-21k dataset and then fine-tuned on the ImageNet-1k dataset. The pre-trained parameters of the models were assembled from the official repository. To get a sense of how these parameters were converted to Keras compatible parameters, please refer to this repository.

Note

Each Keras Application expects a specific kind of input preprocessing. For ConvNeXt, preprocessing is included in the model using a `Normalization` layer. ConvNeXt models expect their inputs to be float or uint8 tensors of pixels with values in the `[0-255]` range.

When calling the `summary()` method after instantiating a ConvNeXt model, prefer setting the `expand_nested` argument `summary()` to `TRUE` to better investigate the instantiated model.

See Also

- `https://keras.io/api/applications/convnext#convnextxlarge-function`

`application_densenet121`

Instantiates the Densenet121 architecture.

Description

Instantiates the Densenet121 architecture.

Usage

```
application_densenet121(
  include_top = TRUE,
  weights = "imagenet",
  input_tensor = NULL,
  input_shape = NULL,
  pooling = NULL,
  classes = 1000L,
  classifier_activation = "softmax"
)
```

Arguments

`include_top`	whether to include the fully-connected layer at the top of the network.
`weights`	one of `NULL` (random initialization), `"imagenet"` (pre-training on ImageNet), or the path to the weights file to be loaded.
`input_tensor`	optional Keras tensor (i.e. output of `layers.Input()`) to use as image input for the model.
`input_shape`	optional shape tuple, only to be specified if `include_top` is `FALSE` (otherwise the input shape has to be `(224, 224, 3)` (with `'channels_last'` data format) or `(3, 224, 224)` (with `'channels_first'` data format). It should have exactly 3 inputs channels, and width and height should be no smaller than 32. E.g. `(200, 200, 3)` would be one valid value.
`pooling`	Optional pooling mode for feature extraction when `include_top` is `FALSE`. • `NULL` means that the output of the model will be the 4D tensor output of the last convolutional block. • `avg` means that global average pooling will be applied to the output of the last convolutional block, and thus the output of the model will be a 2D tensor. • `max` means that global max pooling will be applied.
`classes`	optional number of classes to classify images into, only to be specified if `include_top` is `TRUE`, and if no `weights` argument is specified.
`classifier_activation`	A `str` or callable. The activation function to use on the "top" layer. Ignored unless `include_top=TRUE`. Set `classifier_activation=NULL` to return the logits of the "top" layer. When loading pretrained weights, `classifier_activation` can only be `NULL` or `"softmax"`.

Value

A Keras model instance.

Reference

- Densely Connected Convolutional Networks (CVPR 2017)

Optionally loads weights pre-trained on ImageNet. Note that the data format convention used by the model is the one specified in your Keras config at `~/.keras/keras.json`.

Note

Each Keras Application expects a specific kind of input preprocessing. For DenseNet, call application_preprocess_input on your inputs before passing them to the model.

See Also

- https://keras.io/api/applications/densenet#densenet121-function

---

application_densenet169

Instantiates the Densenet169 architecture.

Description

Instantiates the Densenet169 architecture.

Usage

```
application_densenet169(
  include_top = TRUE,
  weights = "imagenet",
  input_tensor = NULL,
  input_shape = NULL,
  pooling = NULL,
  classes = 1000L,
  classifier_activation = "softmax"
)
```

Arguments

include_top	whether to include the fully-connected layer at the top of the network.
weights	one of NULL (random initialization), "imagenet" (pre-training on ImageNet), or the path to the weights file to be loaded.
input_tensor	optional Keras tensor (i.e. output of layers.Input()) to use as image input for the model.
input_shape	optional shape tuple, only to be specified if include_top is FALSE (otherwise the input shape has to be (224, 224, 3) (with 'channels_last' data format) or (3, 224, 224) (with 'channels_first' data format). It should have exactly 3 inputs channels, and width and height should be no smaller than 32. E.g. (200, 200, 3) would be one valid value.
pooling	Optional pooling mode for feature extraction when include_top is FALSE. • NULL means that the output of the model will be the 4D tensor output of the last convolutional block. • avg means that global average pooling will be applied to the output of the last convolutional block, and thus the output of the model will be a 2D tensor.

- max means that global max pooling will be applied.

classes
: optional number of classes to classify images into, only to be specified if include_top is TRUE, and if no weights argument is specified.

classifier_activation
: A str or callable. The activation function to use on the "top" layer. Ignored unless include_top=TRUE. Set classifier_activation=NULL to return the logits of the "top" layer. When loading pretrained weights, classifier_activation can only be NULL or "softmax".

Value

A Keras model instance.

Reference

- Densely Connected Convolutional Networks (CVPR 2017)

Optionally loads weights pre-trained on ImageNet. Note that the data format convention used by the model is the one specified in your Keras config at ~/.keras/keras.json.

Note

Each Keras Application expects a specific kind of input preprocessing. For DenseNet, call application_preprocess_input on your inputs before passing them to the model.

See Also

- https://keras.io/api/applications/densenet#densenet169-function

---

application_densenet201

Instantiates the Densenet201 architecture.

Description

Instantiates the Densenet201 architecture.

Usage

```
application_densenet201(
  include_top = TRUE,
  weights = "imagenet",
  input_tensor = NULL,
  input_shape = NULL,
  pooling = NULL,
  classes = 1000L,
  classifier_activation = "softmax"
)
```

Arguments

`include_top`	whether to include the fully-connected layer at the top of the network.
`weights`	one of `NULL` (random initialization), `"imagenet"` (pre-training on ImageNet), or the path to the weights file to be loaded.
`input_tensor`	optional Keras tensor (i.e. output of `layers.Input()`) to use as image input for the model.
`input_shape`	optional shape tuple, only to be specified if `include_top` is `FALSE` (otherwise the input shape has to be `(224, 224, 3)` (with `'channels_last'` data format) or `(3, 224, 224)` (with `'channels_first'` data format). It should have exactly 3 inputs channels, and width and height should be no smaller than 32. E.g. `(200, 200, 3)` would be one valid value.
`pooling`	Optional pooling mode for feature extraction when `include_top` is `FALSE`. • `NULL` means that the output of the model will be the 4D tensor output of the last convolutional block. • `avg` means that global average pooling will be applied to the output of the last convolutional block, and thus the output of the model will be a 2D tensor. • `max` means that global max pooling will be applied.
`classes`	optional number of classes to classify images into, only to be specified if `include_top` is `TRUE`, and if no `weights` argument is specified.
`classifier_activation`	A `str` or callable. The activation function to use on the "top" layer. Ignored unless `include_top=TRUE`. Set `classifier_activation=NULL` to return the logits of the "top" layer. When loading pretrained weights, `classifier_activation` can only be `NULL` or `"softmax"`.

Value

A Keras model instance.

Reference

- Densely Connected Convolutional Networks (CVPR 2017)

Optionally loads weights pre-trained on ImageNet. Note that the data format convention used by the model is the one specified in your Keras config at `~/.keras/keras.json`.

Note

Each Keras Application expects a specific kind of input preprocessing. For DenseNet, call `application_preprocess_input` on your inputs before passing them to the model.

See Also

- `https://keras.io/api/applications/densenet#densenet201-function`

application_efficientnet_b0

Instantiates the EfficientNetB0 architecture.

Description

Instantiates the EfficientNetB0 architecture.

Usage

```
application_efficientnet_b0(
  include_top = TRUE,
  weights = "imagenet",
  input_tensor = NULL,
  input_shape = NULL,
  pooling = NULL,
  classes = 1000L,
  classifier_activation = "softmax",
  ...
)
```

Arguments

`include_top`	Whether to include the fully-connected layer at the top of the network. Defaults to `TRUE`.
`weights`	One of `NULL` (random initialization), `"imagenet"` (pre-training on ImageNet), or the path to the weights file to be loaded. Defaults to `"imagenet"`.
`input_tensor`	Optional Keras tensor (i.e. output of `layers.Input()`) to use as image input for the model.
`input_shape`	Optional shape tuple, only to be specified if `include_top` is `FALSE`. It should have exactly 3 inputs channels.
`pooling`	Optional pooling mode for feature extraction when `include_top` is `FALSE`. Defaults to `NULL`. • `NULL` means that the output of the model will be the 4D tensor output of the last convolutional layer. • `avg` means that global average pooling will be applied to the output of the last convolutional layer, and thus the output of the model will be a 2D tensor. • `max` means that global max pooling will be applied.
`classes`	Optional number of classes to classify images into, only to be specified if `include_top` is TRUE, and if no `weights` argument is specified. 1000 is how many ImageNet classes there are. Defaults to `1000`.

`classifier_activation`
: A `str` or callable. The activation function to use on the "top" layer. Ignored unless `include_top=TRUE`. Set `classifier_activation=NULL` to return the logits of the "top" layer. Defaults to `'softmax'`. When loading pretrained weights, `classifier_activation` can only be `NULL` or `"softmax"`.

`...`
: For forward/backward compatability.

Value

A model instance.

Reference

- EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks (ICML 2019)

This function returns a Keras image classification model, optionally loaded with weights pre-trained on ImageNet.

For image classification use cases, see this page for detailed examples.

For transfer learning use cases, make sure to read the guide to transfer learning & fine-tuning.

Note

Each Keras Application expects a specific kind of input preprocessing. For EfficientNet, input preprocessing is included as part of the model (as a `Rescaling` layer), and thus `application_preprocess_inputs()` is actually a pass-through function. EfficientNet models expect their inputs to be float tensors of pixels with values in the `[0-255]` range.

See Also

- `https://keras.io/api/applications/efficientnet#efficientnetb0-function`

`application_efficientnet_b1`
Instantiates the EfficientNetB1 architecture.

Description

Instantiates the EfficientNetB1 architecture.

Usage

```
application_efficientnet_b1(
  include_top = TRUE,
  weights = "imagenet",
  input_tensor = NULL,
  input_shape = NULL,
  pooling = NULL,
  classes = 1000L,
  classifier_activation = "softmax",
  ...
)
```

Arguments

`include_top`	Whether to include the fully-connected layer at the top of the network. Defaults to `TRUE`.
`weights`	One of `NULL` (random initialization), `"imagenet"` (pre-training on ImageNet), or the path to the weights file to be loaded. Defaults to `"imagenet"`.
`input_tensor`	Optional Keras tensor (i.e. output of `layers.Input()`) to use as image input for the model.
`input_shape`	Optional shape tuple, only to be specified if `include_top` is `FALSE`. It should have exactly 3 inputs channels.
`pooling`	Optional pooling mode for feature extraction when `include_top` is `FALSE`. Defaults to `NULL`. • `NULL` means that the output of the model will be the 4D tensor output of the last convolutional layer. • `avg` means that global average pooling will be applied to the output of the last convolutional layer, and thus the output of the model will be a 2D tensor. • `max` means that global max pooling will be applied.
`classes`	Optional number of classes to classify images into, only to be specified if `include_top` is TRUE, and if no `weights` argument is specified. 1000 is how many ImageNet classes there are. Defaults to `1000`.
`classifier_activation`	A `str` or callable. The activation function to use on the "top" layer. Ignored unless `include_top=TRUE`. Set `classifier_activation=NULL` to return the logits of the "top" layer. Defaults to `'softmax'`. When loading pretrained weights, `classifier_activation` can only be `NULL` or `"softmax"`.
`...`	For forward/backward compatability.

Value

A model instance.

Reference

- EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks (ICML 2019)

This function returns a Keras image classification model, optionally loaded with weights pre-trained on ImageNet.

For image classification use cases, see this page for detailed examples.

For transfer learning use cases, make sure to read the guide to transfer learning & fine-tuning.

Note

Each Keras Application expects a specific kind of input preprocessing. For EfficientNet, input preprocessing is included as part of the model (as a `Rescaling` layer), and thus `application_preprocess_inputs()` is actually a pass-through function. EfficientNet models expect their inputs to be float tensors of pixels with values in the `[0-255]` range.

See Also

- `https://keras.io/api/applications/efficientnet#efficientnetb1-function`

`application_efficientnet_b2`
Instantiates the EfficientNetB2 architecture.

Description

Instantiates the EfficientNetB2 architecture.

Usage

```
application_efficientnet_b2(
  include_top = TRUE,
  weights = "imagenet",
  input_tensor = NULL,
  input_shape = NULL,
  pooling = NULL,
  classes = 1000L,
  classifier_activation = "softmax",
  ...
)
```

Arguments

`include_top`	Whether to include the fully-connected layer at the top of the network. Defaults to `TRUE`.
`weights`	One of `NULL` (random initialization), `"imagenet"` (pre-training on ImageNet), or the path to the weights file to be loaded. Defaults to `"imagenet"`.
`input_tensor`	Optional Keras tensor (i.e. output of `layers.Input()`) to use as image input for the model.
`input_shape`	Optional shape tuple, only to be specified if `include_top` is `FALSE`. It should have exactly 3 inputs channels.
`pooling`	Optional pooling mode for feature extraction when `include_top` is `FALSE`. Defaults to `NULL`. • `NULL` means that the output of the model will be the 4D tensor output of the last convolutional layer.

- `avg` means that global average pooling will be applied to the output of the last convolutional layer, and thus the output of the model will be a 2D tensor.
- `max` means that global max pooling will be applied.

`classes`	Optional number of classes to classify images into, only to be specified if `include_top` is TRUE, and if no `weights` argument is specified. 1000 is how many ImageNet classes there are. Defaults to `1000`.
`classifier_activation`	A `str` or callable. The activation function to use on the "top" layer. Ignored unless `include_top=TRUE`. Set `classifier_activation=NULL` to return the logits of the "top" layer. Defaults to `'softmax'`. When loading pretrained weights, `classifier_activation` can only be `NULL` or `"softmax"`.
`...`	For forward/backward compatability.

Value

A model instance.

Reference

- EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks (ICML 2019)

This function returns a Keras image classification model, optionally loaded with weights pre-trained on ImageNet.

For image classification use cases, see this page for detailed examples.

For transfer learning use cases, make sure to read the guide to transfer learning & fine-tuning.

Note

Each Keras Application expects a specific kind of input preprocessing. For EfficientNet, input preprocessing is included as part of the model (as a `Rescaling` layer), and thus `application_preprocess_inputs()` is actually a pass-through function. EfficientNet models expect their inputs to be float tensors of pixels with values in the `[0-255]` range.

See Also

- `https://keras.io/api/applications/efficientnet#efficientnetb2-function`

---

application_efficientnet_b3

Instantiates the EfficientNetB3 architecture.

Description

Instantiates the EfficientNetB3 architecture.

Usage

```
application_efficientnet_b3(
  include_top = TRUE,
  weights = "imagenet",
  input_tensor = NULL,
  input_shape = NULL,
  pooling = NULL,
  classes = 1000L,
  classifier_activation = "softmax",
  ...
)
```

Arguments

`include_top`	Whether to include the fully-connected layer at the top of the network. Defaults to `TRUE`.
`weights`	One of `NULL` (random initialization), `"imagenet"` (pre-training on ImageNet), or the path to the weights file to be loaded. Defaults to `"imagenet"`.
`input_tensor`	Optional Keras tensor (i.e. output of `layers.Input()`) to use as image input for the model.
`input_shape`	Optional shape tuple, only to be specified if `include_top` is `FALSE`. It should have exactly 3 inputs channels.
`pooling`	Optional pooling mode for feature extraction when `include_top` is `FALSE`. Defaults to `NULL`. • `NULL` means that the output of the model will be the 4D tensor output of the last convolutional layer. • `avg` means that global average pooling will be applied to the output of the last convolutional layer, and thus the output of the model will be a 2D tensor. • `max` means that global max pooling will be applied.
`classes`	Optional number of classes to classify images into, only to be specified if `include_top` is TRUE, and if no `weights` argument is specified. 1000 is how many ImageNet classes there are. Defaults to `1000`.
`classifier_activation`	A `str` or callable. The activation function to use on the "top" layer. Ignored unless `include_top=TRUE`. Set `classifier_activation=NULL` to return the logits of the "top" layer. Defaults to `'softmax'`. When loading pretrained weights, `classifier_activation` can only be `NULL` or `"softmax"`.
`...`	For forward/backward compatability.

Value

A model instance.

Reference

- EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks (ICML 2019)

This function returns a Keras image classification model, optionally loaded with weights pre-trained on ImageNet.

For image classification use cases, see this page for detailed examples.

For transfer learning use cases, make sure to read the guide to transfer learning & fine-tuning.

Note

Each Keras Application expects a specific kind of input preprocessing. For EfficientNet, input preprocessing is included as part of the model (as a `Rescaling` layer), and thus `application_preprocess_inputs()` is actually a pass-through function. EfficientNet models expect their inputs to be float tensors of pixels with values in the `[0-255]` range.

See Also

- `https://keras.io/api/applications/efficientnet#efficientnetb3-function`

---

application_efficientnet_b4

Instantiates the EfficientNetB4 architecture.

Description

Instantiates the EfficientNetB4 architecture.

Usage

```
application_efficientnet_b4(
  include_top = TRUE,
  weights = "imagenet",
  input_tensor = NULL,
  input_shape = NULL,
  pooling = NULL,
  classes = 1000L,
  classifier_activation = "softmax",
  ...
)
```

Arguments

`include_top`	Whether to include the fully-connected layer at the top of the network. Defaults to `TRUE`.
`weights`	One of `NULL` (random initialization), `"imagenet"` (pre-training on ImageNet), or the path to the weights file to be loaded. Defaults to `"imagenet"`.

`input_tensor`	Optional Keras tensor (i.e. output of `layers.Input()`) to use as image input for the model.
`input_shape`	Optional shape tuple, only to be specified if `include_top` is `FALSE`. It should have exactly 3 inputs channels.
`pooling`	Optional pooling mode for feature extraction when `include_top` is `FALSE`. Defaults to `NULL`. • `NULL` means that the output of the model will be the 4D tensor output of the last convolutional layer. • `avg` means that global average pooling will be applied to the output of the last convolutional layer, and thus the output of the model will be a 2D tensor. • `max` means that global max pooling will be applied.
`classes`	Optional number of classes to classify images into, only to be specified if `include_top` is TRUE, and if no `weights` argument is specified. 1000 is how many ImageNet classes there are. Defaults to `1000`.
`classifier_activation`	A `str` or callable. The activation function to use on the "top" layer. Ignored unless `include_top=TRUE`. Set `classifier_activation=NULL` to return the logits of the "top" layer. Defaults to `'softmax'`. When loading pretrained weights, `classifier_activation` can only be `NULL` or `"softmax"`.
`...`	For forward/backward compatability.

Value

A model instance.

Reference

- EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks (ICML 2019)

This function returns a Keras image classification model, optionally loaded with weights pre-trained on ImageNet.

For image classification use cases, see this page for detailed examples.

For transfer learning use cases, make sure to read the guide to transfer learning & fine-tuning.

Note

Each Keras Application expects a specific kind of input preprocessing. For EfficientNet, input preprocessing is included as part of the model (as a `Rescaling` layer), and thus `application_preprocess_inputs()` is actually a pass-through function. EfficientNet models expect their inputs to be float tensors of pixels with values in the `[0-255]` range.

See Also

- `https://keras.io/api/applications/efficientnet#efficientnetb4-function`

application_efficientnet_b5

Instantiates the EfficientNetB5 architecture.

Description

Instantiates the EfficientNetB5 architecture.

Usage

```
application_efficientnet_b5(
  include_top = TRUE,
  weights = "imagenet",
  input_tensor = NULL,
  input_shape = NULL,
  pooling = NULL,
  classes = 1000L,
  classifier_activation = "softmax",
  ...
)
```

Arguments

`include_top`	Whether to include the fully-connected layer at the top of the network. Defaults to `TRUE`.
`weights`	One of `NULL` (random initialization), `"imagenet"` (pre-training on ImageNet), or the path to the weights file to be loaded. Defaults to `"imagenet"`.
`input_tensor`	Optional Keras tensor (i.e. output of `layers.Input()`) to use as image input for the model.
`input_shape`	Optional shape tuple, only to be specified if `include_top` is `FALSE`. It should have exactly 3 inputs channels.
`pooling`	Optional pooling mode for feature extraction when `include_top` is `FALSE`. Defaults to `NULL`. • `NULL` means that the output of the model will be the 4D tensor output of the last convolutional layer. • `avg` means that global average pooling will be applied to the output of the last convolutional layer, and thus the output of the model will be a 2D tensor. • `max` means that global max pooling will be applied.
`classes`	Optional number of classes to classify images into, only to be specified if `include_top` is TRUE, and if no `weights` argument is specified. 1000 is how many ImageNet classes there are. Defaults to `1000`.

`classifier_activation`	A `str` or callable. The activation function to use on the "top" layer. Ignored unless `include_top=TRUE`. Set `classifier_activation=NULL` to return the logits of the "top" layer. Defaults to `'softmax'`. When loading pretrained weights, `classifier_activation` can only be `NULL` or `"softmax"`.
`...`	For forward/backward compatability.

Value

A model instance.

Reference

- EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks (ICML 2019)

This function returns a Keras image classification model, optionally loaded with weights pre-trained on ImageNet.

For image classification use cases, see this page for detailed examples.

For transfer learning use cases, make sure to read the guide to transfer learning & fine-tuning.

Note

Each Keras Application expects a specific kind of input preprocessing. For EfficientNet, input preprocessing is included as part of the model (as a `Rescaling` layer), and thus `application_preprocess_inputs()` is actually a pass-through function. EfficientNet models expect their inputs to be float tensors of pixels with values in the `[0-255]` range.

See Also

- `https://keras.io/api/applications/efficientnet#efficientnetb5-function`

`application_efficientnet_b6`

Instantiates the EfficientNetB6 architecture.

Description

Instantiates the EfficientNetB6 architecture.

Usage

```
application_efficientnet_b6(
  include_top = TRUE,
  weights = "imagenet",
  input_tensor = NULL,
  input_shape = NULL,
  pooling = NULL,
  classes = 1000L,
  classifier_activation = "softmax",
  ...
)
```

Arguments

`include_top`	Whether to include the fully-connected layer at the top of the network. Defaults to `TRUE`.
`weights`	One of `NULL` (random initialization), `"imagenet"` (pre-training on ImageNet), or the path to the weights file to be loaded. Defaults to `"imagenet"`.
`input_tensor`	Optional Keras tensor (i.e. output of `layers.Input()`) to use as image input for the model.
`input_shape`	Optional shape tuple, only to be specified if `include_top` is `FALSE`. It should have exactly 3 inputs channels.
`pooling`	Optional pooling mode for feature extraction when `include_top` is `FALSE`. Defaults to `NULL`. • `NULL` means that the output of the model will be the 4D tensor output of the last convolutional layer. • `avg` means that global average pooling will be applied to the output of the last convolutional layer, and thus the output of the model will be a 2D tensor. • `max` means that global max pooling will be applied.
`classes`	Optional number of classes to classify images into, only to be specified if `include_top` is TRUE, and if no `weights` argument is specified. 1000 is how many ImageNet classes there are. Defaults to `1000`.
`classifier_activation`	A `str` or callable. The activation function to use on the "top" layer. Ignored unless `include_top=TRUE`. Set `classifier_activation=NULL` to return the logits of the "top" layer. Defaults to `'softmax'`. When loading pretrained weights, `classifier_activation` can only be `NULL` or `"softmax"`.
`...`	For forward/backward compatability.

Value

A model instance.

Reference

- EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks (ICML 2019)

This function returns a Keras image classification model, optionally loaded with weights pre-trained on ImageNet.

For image classification use cases, see this page for detailed examples.

For transfer learning use cases, make sure to read the guide to transfer learning & fine-tuning.

Note

Each Keras Application expects a specific kind of input preprocessing. For EfficientNet, input preprocessing is included as part of the model (as a `Rescaling` layer), and thus `application_preprocess_inputs()` is actually a pass-through function. EfficientNet models expect their inputs to be float tensors of pixels with values in the `[0-255]` range.

See Also

- `https://keras.io/api/applications/efficientnet#efficientnetb6-function`

---

application_efficientnet_b7

Instantiates the EfficientNetB7 architecture.

Description

Instantiates the EfficientNetB7 architecture.

Usage

```
application_efficientnet_b7(
  include_top = TRUE,
  weights = "imagenet",
  input_tensor = NULL,
  input_shape = NULL,
  pooling = NULL,
  classes = 1000L,
  classifier_activation = "softmax",
  ...
)
```

Arguments

`include_top`	Whether to include the fully-connected layer at the top of the network. Defaults to `TRUE`.
`weights`	One of `NULL` (random initialization), `"imagenet"` (pre-training on ImageNet), or the path to the weights file to be loaded. Defaults to `"imagenet"`.
`input_tensor`	Optional Keras tensor (i.e. output of `layers.Input()`) to use as image input for the model.
`input_shape`	Optional shape tuple, only to be specified if `include_top` is `FALSE`. It should have exactly 3 inputs channels.
`pooling`	Optional pooling mode for feature extraction when `include_top` is `FALSE`. Defaults to `NULL`. • `NULL` means that the output of the model will be the 4D tensor output of the last convolutional layer.

- avg means that global average pooling will be applied to the output of the last convolutional layer, and thus the output of the model will be a 2D tensor.
- max means that global max pooling will be applied.

classes	Optional number of classes to classify images into, only to be specified if include_top is TRUE, and if no weights argument is specified. 1000 is how many ImageNet classes there are. Defaults to 1000.
classifier_activation	A str or callable. The activation function to use on the "top" layer. Ignored unless include_top=TRUE. Set classifier_activation=NULL to return the logits of the "top" layer. Defaults to 'softmax'. When loading pretrained weights, classifier_activation can only be NULL or "softmax".
...	For forward/backward compatability.

Value

A model instance.

Reference

- EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks (ICML 2019)

This function returns a Keras image classification model, optionally loaded with weights pre-trained on ImageNet.

For image classification use cases, see this page for detailed examples.

For transfer learning use cases, make sure to read the guide to transfer learning & fine-tuning.

Note

Each Keras Application expects a specific kind of input preprocessing. For EfficientNet, input preprocessing is included as part of the model (as a Rescaling layer), and thus application_preprocess_inputs() is actually a pass-through function. EfficientNet models expect their inputs to be float tensors of pixels with values in the [0-255] range.

See Also

- https://keras.io/api/applications/efficientnet#efficientnetb7-function

application_efficientnet_v2b0

Instantiates the EfficientNetV2B0 architecture.

Description

Instantiates the EfficientNetV2B0 architecture.

Usage

```
application_efficientnet_v2b0(
  include_top = TRUE,
  weights = "imagenet",
  input_tensor = NULL,
  input_shape = NULL,
  pooling = NULL,
  classes = 1000L,
  classifier_activation = "softmax",
  include_preprocessing = TRUE
)
```

Arguments

`include_top`	Boolean, whether to include the fully-connected layer at the top of the network. Defaults to `TRUE`.
`weights`	One of `NULL` (random initialization), `"imagenet"` (pre-training on ImageNet), or the path to the weights file to be loaded. Defaults to `"imagenet"`.
`input_tensor`	Optional Keras tensor (i.e. output of `layers.Input()`) to use as image input for the model.
`input_shape`	Optional shape tuple, only to be specified if `include_top` is `FALSE`. It should have exactly 3 inputs channels.
`pooling`	Optional pooling mode for feature extraction when `include_top` is `FALSE`. Defaults to NULL. • `NULL` means that the output of the model will be the 4D tensor output of the last convolutional layer. • `"avg"` means that global average pooling will be applied to the output of the last convolutional layer, and thus the output of the model will be a 2D tensor. • `"max"` means that global max pooling will be applied.
`classes`	Optional number of classes to classify images into, only to be specified if `include_top` is `TRUE`, and if no `weights` argument is specified. Defaults to 1000 (number of ImageNet classes).
`classifier_activation`	A string or callable. The activation function to use on the "top" layer. Ignored unless `include_top=TRUE`. Set `classifier_activation=NULL` to return the logits of the "top" layer. Defaults to `"softmax"`. When loading pretrained weights, `classifier_activation` can only be `NULL` or `"softmax"`.
`include_preprocessing`	Boolean, whether to include the preprocessing layer at the bottom of the network.

Value

A model instance.

Reference

- EfficientNetV2: Smaller Models and Faster Training (ICML 2021)

This function returns a Keras image classification model, optionally loaded with weights pre-trained on ImageNet.

For image classification use cases, see this page for detailed examples.

For transfer learning use cases, make sure to read the guide to transfer learning & fine-tuning.

Note

Each Keras Application expects a specific kind of input preprocessing. For EfficientNetV2, by default input preprocessing is included as a part of the model (as a `Rescaling` layer), and thus `application_preprocess_inputs()` is actually a pass-through function. In this use case, EfficientNetV2 models expect their inputs to be float tensors of pixels with values in the `[0, 255]` range. At the same time, preprocessing as a part of the model (i.e. `Rescaling` layer) can be disabled by setting `include_preprocessing` argument to `FALSE`. With preprocessing disabled EfficientNetV2 models expect their inputs to be float tensors of pixels with values in the `[-1, 1]` range.

See Also

- `https://keras.io/api/applications/efficientnet_v2#efficientnetv2b0-function`

`application_efficientnet_v2b1`

Instantiates the EfficientNetV2B1 architecture.

Description

Instantiates the EfficientNetV2B1 architecture.

Usage

```
application_efficientnet_v2b1(
  include_top = TRUE,
  weights = "imagenet",
  input_tensor = NULL,
  input_shape = NULL,
  pooling = NULL,
  classes = 1000L,
  classifier_activation = "softmax",
  include_preprocessing = TRUE
)
```

Arguments

`include_top`	Boolean, whether to include the fully-connected layer at the top of the network. Defaults to `TRUE`.
`weights`	One of `NULL` (random initialization), `"imagenet"` (pre-training on ImageNet), or the path to the weights file to be loaded. Defaults to `"imagenet"`.
`input_tensor`	Optional Keras tensor (i.e. output of `layers.Input()`) to use as image input for the model.
`input_shape`	Optional shape tuple, only to be specified if `include_top` is `FALSE`. It should have exactly 3 inputs channels.
`pooling`	Optional pooling mode for feature extraction when `include_top` is `FALSE`. Defaults to NULL. • `NULL` means that the output of the model will be the 4D tensor output of the last convolutional layer. • `"avg"` means that global average pooling will be applied to the output of the last convolutional layer, and thus the output of the model will be a 2D tensor. • `"max"` means that global max pooling will be applied.
`classes`	Optional number of classes to classify images into, only to be specified if `include_top` is `TRUE`, and if no `weights` argument is specified. Defaults to 1000 (number of ImageNet classes).
`classifier_activation`	A string or callable. The activation function to use on the "top" layer. Ignored unless `include_top=TRUE`. Set `classifier_activation=NULL` to return the logits of the "top" layer. Defaults to `"softmax"`. When loading pretrained weights, `classifier_activation` can only be `NULL` or `"softmax"`.
`include_preprocessing`	Boolean, whether to include the preprocessing layer at the bottom of the network.

Value

A model instance.

Reference

- EfficientNetV2: Smaller Models and Faster Training (ICML 2021)

This function returns a Keras image classification model, optionally loaded with weights pre-trained on ImageNet.

For image classification use cases, see this page for detailed examples.

For transfer learning use cases, make sure to read the guide to transfer learning & fine-tuning.

Note

Each Keras Application expects a specific kind of input preprocessing. For EfficientNetV2, by default input preprocessing is included as a part of the model (as a `Rescaling` layer), and thus application_preprocess_inputs() is actually a pass-through function. In this use case, EfficientNetV2 models expect their inputs to be float tensors of pixels with values in the [0, 255] range. At the same time, preprocessing as a part of the model (i.e. Rescaling layer) can be disabled by setting include_preprocessing argument to FALSE. With preprocessing disabled EfficientNetV2 models expect their inputs to be float tensors of pixels with values in the [-1, 1] range.

See Also

- https://keras.io/api/applications/efficientnet_v2#efficientnetv2b1-function

---

application_efficientnet_v2b2

Instantiates the EfficientNetV2B2 architecture.

Description

Instantiates the EfficientNetV2B2 architecture.

Usage

```
application_efficientnet_v2b2(
  include_top = TRUE,
  weights = "imagenet",
  input_tensor = NULL,
  input_shape = NULL,
  pooling = NULL,
  classes = 1000L,
  classifier_activation = "softmax",
  include_preprocessing = TRUE
)
```

Arguments

include_top	Boolean, whether to include the fully-connected layer at the top of the network. Defaults to TRUE.
weights	One of NULL (random initialization), "imagenet" (pre-training on ImageNet), or the path to the weights file to be loaded. Defaults to "imagenet".
input_tensor	Optional Keras tensor (i.e. output of layers.Input()) to use as image input for the model.
input_shape	Optional shape tuple, only to be specified if include_top is FALSE. It should have exactly 3 inputs channels.
pooling	Optional pooling mode for feature extraction when include_top is FALSE. Defaults to NULL. • NULL means that the output of the model will be the 4D tensor output of the last convolutional layer.

- "avg" means that global average pooling will be applied to the output of the last convolutional layer, and thus the output of the model will be a 2D tensor.
- "max" means that global max pooling will be applied.

classes	Optional number of classes to classify images into, only to be specified if include_top is TRUE, and if no weights argument is specified. Defaults to 1000 (number of ImageNet classes).
classifier_activation	A string or callable. The activation function to use on the "top" layer. Ignored unless include_top=TRUE. Set classifier_activation=NULL to return the logits of the "top" layer. Defaults to "softmax". When loading pretrained weights, classifier_activation can only be NULL or "softmax".
include_preprocessing	Boolean, whether to include the preprocessing layer at the bottom of the network.

Value

A model instance.

Reference

- EfficientNetV2: Smaller Models and Faster Training (ICML 2021)

This function returns a Keras image classification model, optionally loaded with weights pre-trained on ImageNet.

For image classification use cases, see this page for detailed examples.

For transfer learning use cases, make sure to read the guide to transfer learning & fine-tuning.

Note

Each Keras Application expects a specific kind of input preprocessing. For EfficientNetV2, by default input preprocessing is included as a part of the model (as a Rescaling layer), and thus application_preprocess_inputs() is actually a pass-through function. In this use case, EfficientNetV2 models expect their inputs to be float tensors of pixels with values in the [0, 255] range. At the same time, preprocessing as a part of the model (i.e. Rescaling layer) can be disabled by setting include_preprocessing argument to FALSE. With preprocessing disabled EfficientNetV2 models expect their inputs to be float tensors of pixels with values in the [-1, 1] range.

See Also

- https://keras.io/api/applications/efficientnet_v2#efficientnetv2b2-function

application_efficientnet_v2b3

Instantiates the EfficientNetV2B3 architecture.

Description

Instantiates the EfficientNetV2B3 architecture.

Usage

```
application_efficientnet_v2b3(
  include_top = TRUE,
  weights = "imagenet",
  input_tensor = NULL,
  input_shape = NULL,
  pooling = NULL,
  classes = 1000L,
  classifier_activation = "softmax",
  include_preprocessing = TRUE
)
```

Arguments

`include_top`	Boolean, whether to include the fully-connected layer at the top of the network. Defaults to `TRUE`.
`weights`	One of `NULL` (random initialization), `"imagenet"` (pre-training on ImageNet), or the path to the weights file to be loaded. Defaults to `"imagenet"`.
`input_tensor`	Optional Keras tensor (i.e. output of `layers.Input()`) to use as image input for the model.
`input_shape`	Optional shape tuple, only to be specified if `include_top` is `FALSE`. It should have exactly 3 inputs channels.
`pooling`	Optional pooling mode for feature extraction when `include_top` is `FALSE`. Defaults to `NULL`. • `NULL` means that the output of the model will be the 4D tensor output of the last convolutional layer. • `"avg"` means that global average pooling will be applied to the output of the last convolutional layer, and thus the output of the model will be a 2D tensor. • `"max"` means that global max pooling will be applied.
`classes`	Optional number of classes to classify images into, only to be specified if `include_top` is `TRUE`, and if no `weights` argument is specified. Defaults to 1000 (number of ImageNet classes).

`classifier_activation`
: A string or callable. The activation function to use on the "top" layer. Ignored unless `include_top=TRUE`. Set `classifier_activation=NULL` to return the logits of the "top" layer. Defaults to `"softmax"`. When loading pretrained weights, `classifier_activation` can only be `NULL` or `"softmax"`.

`include_preprocessing`
: Boolean, whether to include the preprocessing layer at the bottom of the network.

Value

A model instance.

Reference

- EfficientNetV2: Smaller Models and Faster Training (ICML 2021)

This function returns a Keras image classification model, optionally loaded with weights pre-trained on ImageNet.

For image classification use cases, see this page for detailed examples.

For transfer learning use cases, make sure to read the guide to transfer learning & fine-tuning.

Note

Each Keras Application expects a specific kind of input preprocessing. For EfficientNetV2, by default input preprocessing is included as a part of the model (as a `Rescaling` layer), and thus `application_preprocess_inputs()` is actually a pass-through function. In this use case, EfficientNetV2 models expect their inputs to be float tensors of pixels with values in the `[0, 255]` range. At the same time, preprocessing as a part of the model (i.e. `Rescaling` layer) can be disabled by setting `include_preprocessing` argument to `FALSE`. With preprocessing disabled EfficientNetV2 models expect their inputs to be float tensors of pixels with values in the `[-1, 1]` range.

See Also

- `https://keras.io/api/applications/efficientnet_v2#efficientnetv2b3-function`

`application_efficientnet_v2l` *Instantiates the EfficientNetV2L architecture.*

Description

Instantiates the EfficientNetV2L architecture.

Usage

```
application_efficientnet_v2l(
  include_top = TRUE,
  weights = "imagenet",
  input_tensor = NULL,
  input_shape = NULL,
  pooling = NULL,
  classes = 1000L,
  classifier_activation = "softmax",
  include_preprocessing = TRUE
)
```

Arguments

`include_top`	Boolean, whether to include the fully-connected layer at the top of the network. Defaults to `TRUE`.
`weights`	One of `NULL` (random initialization), `"imagenet"` (pre-training on ImageNet), or the path to the weights file to be loaded. Defaults to `"imagenet"`.
`input_tensor`	Optional Keras tensor (i.e. output of `layers.Input()`) to use as image input for the model.
`input_shape`	Optional shape tuple, only to be specified if `include_top` is `FALSE`. It should have exactly 3 inputs channels.
`pooling`	Optional pooling mode for feature extraction when `include_top` is `FALSE`. Defaults to `NULL`. • `NULL` means that the output of the model will be the 4D tensor output of the last convolutional layer. • `"avg"` means that global average pooling will be applied to the output of the last convolutional layer, and thus the output of the model will be a 2D tensor. • `"max"` means that global max pooling will be applied.
`classes`	Optional number of classes to classify images into, only to be specified if `include_top` is `TRUE`, and if no `weights` argument is specified. Defaults to 1000 (number of ImageNet classes).
`classifier_activation`	A string or callable. The activation function to use on the "top" layer. Ignored unless `include_top=TRUE`. Set `classifier_activation=NULL` to return the logits of the "top" layer. Defaults to `"softmax"`. When loading pretrained weights, `classifier_activation` can only be `NULL` or `"softmax"`.
`include_preprocessing`	Boolean, whether to include the preprocessing layer at the bottom of the network.

Value

A model instance.

Reference

- EfficientNetV2: Smaller Models and Faster Training (ICML 2021)

This function returns a Keras image classification model, optionally loaded with weights pre-trained on ImageNet.

For image classification use cases, see this page for detailed examples.

For transfer learning use cases, make sure to read the guide to transfer learning & fine-tuning.

Note

Each Keras Application expects a specific kind of input preprocessing. For EfficientNetV2, by default input preprocessing is included as a part of the model (as a `Rescaling` layer), and thus `application_preprocess_inputs()` is actually a pass-through function. In this use case, EfficientNetV2 models expect their inputs to be float tensors of pixels with values in the `[0, 255]` range. At the same time, preprocessing as a part of the model (i.e. `Rescaling` layer) can be disabled by setting `include_preprocessing` argument to `FALSE`. With preprocessing disabled EfficientNetV2 models expect their inputs to be float tensors of pixels with values in the `[-1, 1]` range.

See Also

- `https://keras.io/api/applications/efficientnet_v2#efficientnetv2l-function`

`application_efficientnet_v2m` *Instantiates the EfficientNetV2M architecture.*

Description

Instantiates the EfficientNetV2M architecture.

Usage

```
application_efficientnet_v2m(
  include_top = TRUE,
  weights = "imagenet",
  input_tensor = NULL,
  input_shape = NULL,
  pooling = NULL,
  classes = 1000L,
  classifier_activation = "softmax",
  include_preprocessing = TRUE
)
```

Arguments

`include_top`	Boolean, whether to include the fully-connected layer at the top of the network. Defaults to `TRUE`.
`weights`	One of `NULL` (random initialization), `"imagenet"` (pre-training on ImageNet), or the path to the weights file to be loaded. Defaults to `"imagenet"`.
`input_tensor`	Optional Keras tensor (i.e. output of `layers.Input()`) to use as image input for the model.
`input_shape`	Optional shape tuple, only to be specified if `include_top` is `FALSE`. It should have exactly 3 inputs channels.
`pooling`	Optional pooling mode for feature extraction when `include_top` is `FALSE`. Defaults to `NULL`. • `NULL` means that the output of the model will be the 4D tensor output of the last convolutional layer. • `"avg"` means that global average pooling will be applied to the output of the last convolutional layer, and thus the output of the model will be a 2D tensor. • `"max"` means that global max pooling will be applied.
`classes`	Optional number of classes to classify images into, only to be specified if `include_top` is `TRUE`, and if no `weights` argument is specified. Defaults to 1000 (number of ImageNet classes).
`classifier_activation`	A string or callable. The activation function to use on the "top" layer. Ignored unless `include_top=TRUE`. Set `classifier_activation=NULL` to return the logits of the "top" layer. Defaults to `"softmax"`. When loading pretrained weights, `classifier_activation` can only be `NULL` or `"softmax"`.
`include_preprocessing`	Boolean, whether to include the preprocessing layer at the bottom of the network.

Value

A model instance.

Reference

- EfficientNetV2: Smaller Models and Faster Training (ICML 2021)

This function returns a Keras image classification model, optionally loaded with weights pre-trained on ImageNet.

For image classification use cases, see this page for detailed examples.

For transfer learning use cases, make sure to read the guide to transfer learning & fine-tuning.

Note

Each Keras Application expects a specific kind of input preprocessing. For EfficientNetV2, by default input preprocessing is included as a part of the model (as a `Rescaling` layer), and thus application_preprocess_inputs() is actually a pass-through function. In this use case, EfficientNetV2 models expect their inputs to be float tensors of pixels with values in the [0, 255] range. At the same time, preprocessing as a part of the model (i.e. Rescaling layer) can be disabled by setting include_preprocessing argument to FALSE. With preprocessing disabled EfficientNetV2 models expect their inputs to be float tensors of pixels with values in the [-1, 1] range.

See Also

- https://keras.io/api/applications/efficientnet_v2#efficientnetv2m-function

---

application_efficientnet_v2s

Instantiates the EfficientNetV2S architecture.

Description

Instantiates the EfficientNetV2S architecture.

Usage

```
application_efficientnet_v2s(
  include_top = TRUE,
  weights = "imagenet",
  input_tensor = NULL,
  input_shape = NULL,
  pooling = NULL,
  classes = 1000L,
  classifier_activation = "softmax",
  include_preprocessing = TRUE
)
```

Arguments

include_top	Boolean, whether to include the fully-connected layer at the top of the network. Defaults to TRUE.
weights	One of NULL (random initialization), "imagenet" (pre-training on ImageNet), or the path to the weights file to be loaded. Defaults to "imagenet".
input_tensor	Optional Keras tensor (i.e. output of layers.Input()) to use as image input for the model.
input_shape	Optional shape tuple, only to be specified if include_top is FALSE. It should have exactly 3 inputs channels.
pooling	Optional pooling mode for feature extraction when include_top is FALSE. Defaults to NULL. • NULL means that the output of the model will be the 4D tensor output of the last convolutional layer.

- `"avg"` means that global average pooling will be applied to the output of the last convolutional layer, and thus the output of the model will be a 2D tensor.
- `"max"` means that global max pooling will be applied.

`classes` Optional number of classes to classify images into, only to be specified if `include_top` is `TRUE`, and if no `weights` argument is specified. Defaults to 1000 (number of ImageNet classes).

`classifier_activation`
A string or callable. The activation function to use on the "top" layer. Ignored unless `include_top=TRUE`. Set `classifier_activation=NULL` to return the logits of the "top" layer. Defaults to `"softmax"`. When loading pretrained weights, `classifier_activation` can only be `NULL` or `"softmax"`.

`include_preprocessing`
Boolean, whether to include the preprocessing layer at the bottom of the network.

Value

A model instance.

Reference

- EfficientNetV2: Smaller Models and Faster Training (ICML 2021)

This function returns a Keras image classification model, optionally loaded with weights pre-trained on ImageNet.

For image classification use cases, see this page for detailed examples.

For transfer learning use cases, make sure to read the guide to transfer learning & fine-tuning.

Note

Each Keras Application expects a specific kind of input preprocessing. For EfficientNetV2, by default input preprocessing is included as a part of the model (as a `Rescaling` layer), and thus `application_preprocess_inputs()` is actually a pass-through function. In this use case, EfficientNetV2 models expect their inputs to be float tensors of pixels with values in the `[0, 255]` range. At the same time, preprocessing as a part of the model (i.e. `Rescaling` layer) can be disabled by setting `include_preprocessing` argument to `FALSE`. With preprocessing disabled EfficientNetV2 models expect their inputs to be float tensors of pixels with values in the `[-1, 1]` range.

See Also

- `https://keras.io/api/applications/efficientnet_v2#efficientnetv2s-function`

application_inception_resnet_v2

Instantiates the Inception-ResNet v2 architecture.

Description

Instantiates the Inception-ResNet v2 architecture.

Usage

```
application_inception_resnet_v2(
  include_top = TRUE,
  weights = "imagenet",
  input_tensor = NULL,
  input_shape = NULL,
  pooling = NULL,
  classes = 1000L,
  classifier_activation = "softmax"
)
```

Arguments

`include_top`	whether to include the fully-connected layer at the top of the network.
`weights`	one of `NULL` (random initialization), `"imagenet"` (pre-training on ImageNet), or the path to the weights file to be loaded.
`input_tensor`	optional Keras tensor (i.e. output of `layers.Input()`) to use as image input for the model.
`input_shape`	optional shape tuple, only to be specified if `include_top` is `FALSE` (otherwise the input shape has to be `(299, 299, 3)` (with `'channels_last'` data format) or `(3, 299, 299)` (with `'channels_first'` data format). It should have exactly 3 inputs channels, and width and height should be no smaller than 75. E.g. `(150, 150, 3)` would be one valid value.
`pooling`	Optional pooling mode for feature extraction when `include_top` is `FALSE`. • `NULL` means that the output of the model will be the 4D tensor output of the last convolutional block. • `'avg'` means that global average pooling will be applied to the output of the last convolutional block, and thus the output of the model will be a 2D tensor. • `'max'` means that global max pooling will be applied.
`classes`	optional number of classes to classify images into, only to be specified if `include_top` is `TRUE`, and if no `weights` argument is specified.
`classifier_activation`	A `str` or callable. The activation function to use on the "top" layer. Ignored unless `include_top=TRUE`. Set `classifier_activation=NULL` to return the logits of the "top" layer. When loading pretrained weights, `classifier_activation` can only be `NULL` or `"softmax"`.

Value

A model instance.

Reference

- Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning (AAAI 2017)

This function returns a Keras image classification model, optionally loaded with weights pre-trained on ImageNet.

For image classification use cases, see this page for detailed examples.

For transfer learning use cases, make sure to read the guide to transfer learning & fine-tuning.

Note

Each Keras Application expects a specific kind of input preprocessing. For `InceptionResNetV2`, call `application_preprocess_inputs()` on your inputs before passing them to the model. `application_preprocess_inp` will scale input pixels between -1 and 1.

See Also

- `https://keras.io/api/applications/inceptionresnetv2#inceptionresnetv2-function`

`application_inception_v3`

Instantiates the Inception v3 architecture.

Description

Instantiates the Inception v3 architecture.

Usage

```
application_inception_v3(
  include_top = TRUE,
  weights = "imagenet",
  input_tensor = NULL,
  input_shape = NULL,
  pooling = NULL,
  classes = 1000L,
  classifier_activation = "softmax"
)
```

Arguments

`include_top`	Boolean, whether to include the fully-connected layer at the top, as the last layer of the network. Defaults to `TRUE`.
`weights`	One of `NULL` (random initialization), `imagenet` (pre-training on ImageNet), or the path to the weights file to be loaded. Defaults to `"imagenet"`.
`input_tensor`	Optional Keras tensor (i.e. output of `layers.Input()`) to use as image input for the model. `input_tensor` is useful for sharing inputs between multiple different networks. Defaults to `NULL`.
`input_shape`	Optional shape tuple, only to be specified if `include_top` is `FALSE` (otherwise the input shape has to be `(299, 299, 3)` (with `channels_last` data format) or `(3, 299, 299)` (with `channels_first` data format). It should have exactly 3 inputs channels, and width and height should be no smaller than 75. E.g. `(150, 150, 3)` would be one valid value. `input_shape` will be ignored if the `input_tensor` is provided.
`pooling`	Optional pooling mode for feature extraction when `include_top` is `FALSE`. • `NULL` (default) means that the output of the model will be the 4D tensor output of the last convolutional block. • `avg` means that global average pooling will be applied to the output of the last convolutional block, and thus the output of the model will be a 2D tensor. • `max` means that global max pooling will be applied.
`classes`	optional number of classes to classify images into, only to be specified if `include_top` is `TRUE`, and if no `weights` argument is specified. Defaults to 1000.
`classifier_activation`	A `str` or callable. The activation function to use on the "top" layer. Ignored unless `include_top=TRUE`. Set `classifier_activation=NULL` to return the logits of the "top" layer. When loading pretrained weights, `classifier_activation` can only be `NULL` or `"softmax"`.

Value

A model instance.

Reference

- Rethinking the Inception Architecture for Computer Vision (CVPR 2016)

This function returns a Keras image classification model, optionally loaded with weights pre-trained on ImageNet.

For image classification use cases, see this page for detailed examples.

For transfer learning use cases, make sure to read the guide to transfer learning & fine-tuning.

Note

Each Keras Application expects a specific kind of input preprocessing. For `InceptionV3`, call `application_preprocess_inputs()` on your inputs before passing them to the model. `application_preprocess_inputs` will scale input pixels between `-1` and `1`.

See Also

- https://keras.io/api/applications/inceptionv3#inceptionv3-function

---

application_mobilenet *Instantiates the MobileNet architecture.*

Description

Instantiates the MobileNet architecture.

Usage

```
application_mobilenet(
  input_shape = NULL,
  alpha = 1,
  depth_multiplier = 1L,
  dropout = 0.001,
  include_top = TRUE,
  weights = "imagenet",
  input_tensor = NULL,
  pooling = NULL,
  classes = 1000L,
  classifier_activation = "softmax"
)
```

Arguments

`input_shape` Optional shape tuple, only to be specified if `include_top` is `FALSE` (otherwise the input shape has to be `(224, 224, 3)` (with `"channels_last"` data format) or `(3, 224, 224)` (with `"channels_first"` data format). It should have exactly 3 inputs channels, and width and height should be no smaller than 32. E.g. `(200, 200, 3)` would be one valid value. Defaults to `NULL`. `input_shape` will be ignored if the `input_tensor` is provided.

`alpha` Controls the width of the network. This is known as the width multiplier in the MobileNet paper.

- If `alpha` < `1.0`, proportionally decreases the number of filters in each layer.
- If `alpha` > `1.0`, proportionally increases the number of filters in each layer.
- If `alpha` == `1`, default number of filters from the paper are used at each layer. Defaults to `1.0`.

`depth_multiplier`
Depth multiplier for depthwise convolution. This is called the resolution multiplier in the MobileNet paper. Defaults to `1.0`.

`dropout` Dropout rate. Defaults to `0.001`.

`include_top` Boolean, whether to include the fully-connected layer at the top of the network. Defaults to `TRUE`.

`weights`	One of `NULL` (random initialization), `"imagenet"` (pre-training on ImageNet), or the path to the weights file to be loaded. Defaults to `"imagenet"`.
`input_tensor`	Optional Keras tensor (i.e. output of `layers.Input()`) to use as image input for the model. `input_tensor` is useful for sharing inputs between multiple different networks. Defaults to `NULL`.
`pooling`	Optional pooling mode for feature extraction when `include_top` is `FALSE`. • `NULL` (default) means that the output of the model will be the 4D tensor output of the last convolutional block. • `avg` means that global average pooling will be applied to the output of the last convolutional block, and thus the output of the model will be a 2D tensor. • `max` means that global max pooling will be applied.
`classes`	Optional number of classes to classify images into, only to be specified if `include_top` is `TRUE`, and if no `weights` argument is specified. Defaults to `1000`.
`classifier_activation`	A `str` or callable. The activation function to use on the "top" layer. Ignored unless `include_top=TRUE`. Set `classifier_activation=NULL` to return the logits of the "top" layer. When loading pretrained weights, `classifier_activation` can only be `NULL` or `"softmax"`.

Value

A model instance.

Reference

- MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications

This function returns a Keras image classification model, optionally loaded with weights pre-trained on ImageNet.

For image classification use cases, see this page for detailed examples.

For transfer learning use cases, make sure to read the guide to transfer learning & fine-tuning.

Note

Each Keras Application expects a specific kind of input preprocessing. For MobileNet, call `application_preprocess_inpu` on your inputs before passing them to the model. `application_preprocess_inputs()` will scale input pixels between `-1` and `1`.

See Also

- `https://keras.io/api/applications/mobilenet#mobilenet-function`

application_mobilenet_v2

Instantiates the MobileNetV2 architecture.

Description

MobileNetV2 is very similar to the original MobileNet, except that it uses inverted residual blocks with bottlenecking features. It has a drastically lower parameter count than the original MobileNet. MobileNets support any input size greater than 32 x 32, with larger image sizes offering better performance.

Usage

```
application_mobilenet_v2(
  input_shape = NULL,
  alpha = 1,
  include_top = TRUE,
  weights = "imagenet",
  input_tensor = NULL,
  pooling = NULL,
  classes = 1000L,
  classifier_activation = "softmax"
)
```

Arguments

`input_shape`	Optional shape tuple, only to be specified if `include_top` is `FALSE` (otherwise the input shape has to be `(224, 224, 3)` (with `"channels_last"` data format) or `(3, 224, 224)` (with `"channels_first"` data format). It should have exactly 3 inputs channels, and width and height should be no smaller than 32. E.g. `(200, 200, 3)` would be one valid value. Defaults to `NULL`. `input_shape` will be ignored if the `input_tensor` is provided.
`alpha`	Controls the width of the network. This is known as the width multiplier in the MobileNet paper. • If `alpha` < `1.0`, proportionally decreases the number of filters in each layer. • If `alpha` > `1.0`, proportionally increases the number of filters in each layer. • If `alpha` == `1`, default number of filters from the paper are used at each layer. Defaults to `1.0`.
`include_top`	Boolean, whether to include the fully-connected layer at the top of the network. Defaults to `TRUE`.
`weights`	One of `NULL` (random initialization), `"imagenet"` (pre-training on ImageNet), or the path to the weights file to be loaded. Defaults to `"imagenet"`.
`input_tensor`	Optional Keras tensor (i.e. output of `layers.Input()`) to use as image input for the model. `input_tensor` is useful for sharing inputs between multiple different networks. Defaults to `NULL`.

`pooling`	Optional pooling mode for feature extraction when `include_top` is `FALSE`. • `NULL` (default) means that the output of the model will be the 4D tensor output of the last convolutional block. • `avg` means that global average pooling will be applied to the output of the last convolutional block, and thus the output of the model will be a 2D tensor. • `max` means that global max pooling will be applied.
`classes`	Optional number of classes to classify images into, only to be specified if `include_top` is `TRUE`, and if no `weights` argument is specified. Defaults to `1000`.
`classifier_activation`	A `str` or callable. The activation function to use on the "top" layer. Ignored unless `include_top=TRUE`. Set `classifier_activation=NULL` to return the logits of the "top" layer. When loading pretrained weights, `classifier_activation` can only be `NULL` or `"softmax"`.

Value

A model instance.

Reference

- MobileNetV2: Inverted Residuals and Linear Bottlenecks (CVPR 2018)

This function returns a Keras image classification model, optionally loaded with weights pre-trained on ImageNet.

For image classification use cases, see this page for detailed examples.

For transfer learning use cases, make sure to read the guide to transfer learning & fine-tuning.

Note

Each Keras Application expects a specific kind of input preprocessing. For MobileNetV2, call `application_preprocess_inputs()` on your inputs before passing them to the model. `application_preprocess_inputs` will scale input pixels between `-1` and `1`.

See Also

- `https://keras.io/api/applications/mobilenet#mobilenetv2-function`

`application_mobilenet_v3_large`
Instantiates the MobileNetV3Large architecture.

Description

Instantiates the MobileNetV3Large architecture.

Usage

```
application_mobilenet_v3_large(
  input_shape = NULL,
  alpha = 1,
  minimalistic = FALSE,
  include_top = TRUE,
  weights = "imagenet",
  input_tensor = NULL,
  classes = 1000L,
  pooling = NULL,
  dropout_rate = 0.2,
  classifier_activation = "softmax",
  include_preprocessing = TRUE
)
```

Arguments

`input_shape`	Optional shape tuple, to be specified if you would like to use a model with an input image resolution that is not `(224, 224, 3)`. It should have exactly 3 inputs channels. You can also omit this option if you would like to infer input_shape from an input_tensor. If you choose to include both input_tensor and input_shape then input_shape will be used if they match, if the shapes do not match then we will throw an error. E.g. `(160, 160, 3)` would be one valid value.
`alpha`	controls the width of the network. This is known as the depth multiplier in the MobileNetV3 paper, but the name is kept for consistency with MobileNetV1 in Keras. • If `alpha` < `1.0`, proportionally decreases the number of filters in each layer. • If `alpha` > `1.0`, proportionally increases the number of filters in each layer. • If `alpha` == `1`, default number of filters from the paper are used at each layer.
`minimalistic`	In addition to large and small models this module also contains so-called minimalistic models, these models have the same per-layer dimensions characteristic as MobilenetV3 however, they don't utilize any of the advanced blocks (squeeze-and-excite units, hard-swish, and 5x5 convolutions). While these models are less efficient on CPU, they are much more performant on GPU/DSP.
`include_top`	Boolean, whether to include the fully-connected layer at the top of the network. Defaults to `TRUE`.
`weights`	String, one of `NULL` (random initialization), `"imagenet"` (pre-training on ImageNet), or the path to the weights file to be loaded.
`input_tensor`	Optional Keras tensor (i.e. output of `layers.Input()`) to use as image input for the model.
`classes`	Integer, optional number of classes to classify images into, only to be specified if `include_top` is `TRUE`, and if no `weights` argument is specified.
`pooling`	String, optional pooling mode for feature extraction when `include_top` is `FALSE`.

- `NULL` means that the output of the model will be the 4D tensor output of the last convolutional block.
- `avg` means that global average pooling will be applied to the output of the last convolutional block, and thus the output of the model will be a 2D tensor.
- `max` means that global max pooling will be applied.

`dropout_rate` fraction of the input units to drop on the last layer.

`classifier_activation`
A `str` or callable. The activation function to use on the "top" layer. Ignored unless `include_top=TRUE`. Set `classifier_activation=NULL` to return the logits of the "top" layer. When loading pretrained weights, `classifier_activation` can only be `NULL` or `"softmax"`.

`include_preprocessing`
Boolean, whether to include the preprocessing layer (`Rescaling`) at the bottom of the network. Defaults to `TRUE`.

Value

A model instance.

Reference

- Searching for MobileNetV3 (ICCV 2019)

The following table describes the performance of MobileNets v3::

MACs stands for Multiply Adds

Classification Checkpoint	MACs(M)	Parameters(M)	Top1 Accuracy	Pixel1 CPU(ms)
mobilenet_v3_large_1.0_224	217	5.4	75.6	51.2
mobilenet_v3_large_0.75_224	155	4.0	73.3	39.8
mobilenet_v3_large_minimalistic_1.0_224	209	3.9	72.3	44.1
mobilenet_v3_small_1.0_224	66	2.9	68.1	15.8
mobilenet_v3_small_0.75_224	44	2.4	65.4	12.8
mobilenet_v3_small_minimalistic_1.0_224	65	2.0	61.9	12.2

For image classification use cases, see this page for detailed examples.

For transfer learning use cases, make sure to read the guide to transfer learning & fine-tuning.

Note

Each Keras Application expects a specific kind of input preprocessing. For MobileNetV3, by default input preprocessing is included as a part of the model (as a `Rescaling` layer), and thus `application_preprocess_inputs()` is actually a pass-through function. In this use case, MobileNetV3 models expect their inputs to be float tensors of pixels with values in the `[0-255]` range. At the same time, preprocessing as a part of the model (i.e. `Rescaling` layer) can be disabled by setting `include_preprocessing` argument to `FALSE`. With preprocessing disabled MobileNetV3 models expect their inputs to be float tensors of pixels with values in the `[-1, 1]` range.

Call Arguments

- inputs: A floating point numpy.array or backend-native tensor, 4D with 3 color channels, with values in the range [0, 255] if include_preprocessing is TRUE and in the range [-1, 1] otherwise.

See Also

- https://keras.io/api/applications/mobilenet#mobilenetv3large-function

---

application_mobilenet_v3_small
Instantiates the MobileNetV3Small architecture.

Description

Instantiates the MobileNetV3Small architecture.

Usage

```
application_mobilenet_v3_small(
  input_shape = NULL,
  alpha = 1,
  minimalistic = FALSE,
  include_top = TRUE,
  weights = "imagenet",
  input_tensor = NULL,
  classes = 1000L,
  pooling = NULL,
  dropout_rate = 0.2,
  classifier_activation = "softmax",
  include_preprocessing = TRUE
)
```

Arguments

input_shape	Optional shape tuple, to be specified if you would like to use a model with an input image resolution that is not (224, 224, 3). It should have exactly 3 inputs channels. You can also omit this option if you would like to infer input_shape from an input_tensor. If you choose to include both input_tensor and input_shape then input_shape will be used if they match, if the shapes do not match then we will throw an error. E.g. (160, 160, 3) would be one valid value.
alpha	controls the width of the network. This is known as the depth multiplier in the MobileNetV3 paper, but the name is kept for consistency with MobileNetV1 in Keras. • If alpha < 1.0, proportionally decreases the number of filters in each layer.

- If `alpha` > `1.0`, proportionally increases the number of filters in each layer.
- If `alpha` == `1`, default number of filters from the paper are used at each layer.

`minimalistic`	In addition to large and small models this module also contains so-called minimalistic models, these models have the same per-layer dimensions characteristic as MobilenetV3 however, they don't utilize any of the advanced blocks (squeeze-and-excite units, hard-swish, and 5x5 convolutions). While these models are less efficient on CPU, they are much more performant on GPU/DSP.
`include_top`	Boolean, whether to include the fully-connected layer at the top of the network. Defaults to `TRUE`.
`weights`	String, one of `NULL` (random initialization), `"imagenet"` (pre-training on ImageNet), or the path to the weights file to be loaded.
`input_tensor`	Optional Keras tensor (i.e. output of `layers.Input()`) to use as image input for the model.
`classes`	Integer, optional number of classes to classify images into, only to be specified if `include_top` is `TRUE`, and if no `weights` argument is specified.
`pooling`	String, optional pooling mode for feature extraction when `include_top` is `FALSE`. • `NULL` means that the output of the model will be the 4D tensor output of the last convolutional block. • `avg` means that global average pooling will be applied to the output of the last convolutional block, and thus the output of the model will be a 2D tensor. • `max` means that global max pooling will be applied.
`dropout_rate`	fraction of the input units to drop on the last layer.
`classifier_activation`	A `str` or callable. The activation function to use on the "top" layer. Ignored unless `include_top=TRUE`. Set `classifier_activation=NULL` to return the logits of the "top" layer. When loading pretrained weights, `classifier_activation` can only be `NULL` or `"softmax"`.
`include_preprocessing`	Boolean, whether to include the preprocessing layer (`Rescaling`) at the bottom of the network. Defaults to `TRUE`.

Value

A model instance.

Reference

- Searching for MobileNetV3 (ICCV 2019)

The following table describes the performance of MobileNets v3::

MACs stands for Multiply Adds

Classification Checkpoint	MACs(M)	Parameters(M)	Top1 Accuracy	Pixel1 CPU(ms)
mobilenet_v3_large_1.0_224	217	5.4	75.6	51.2

mobilenet_v3_large_0.75_224	155	4.0	73.3	39.8
mobilenet_v3_large_minimalistic_1.0_224	209	3.9	72.3	44.1
mobilenet_v3_small_1.0_224	66	2.9	68.1	15.8
mobilenet_v3_small_0.75_224	44	2.4	65.4	12.8
mobilenet_v3_small_minimalistic_1.0_224	65	2.0	61.9	12.2

For image classification use cases, see this page for detailed examples.
For transfer learning use cases, make sure to read the guide to transfer learning & fine-tuning.

Note

Each Keras Application expects a specific kind of input preprocessing. For MobileNetV3, by default input preprocessing is included as a part of the model (as a `Rescaling` layer), and thus `application_preprocess_inputs()` is actually a pass-through function. In this use case, MobileNetV3 models expect their inputs to be float tensors of pixels with values in the `[0-255]` range. At the same time, preprocessing as a part of the model (i.e. `Rescaling` layer) can be disabled by setting `include_preprocessing` argument to `FALSE`. With preprocessing disabled MobileNetV3 models expect their inputs to be float tensors of pixels with values in the `[-1, 1]` range.

Call Arguments

- `inputs`: A floating point `numpy.array` or backend-native tensor, 4D with 3 color channels, with values in the range `[0, 255]` if `include_preprocessing` is `TRUE` and in the range `[-1, 1]` otherwise.

See Also

- `https://keras.io/api/applications/mobilenet#mobilenetv3small-function`

`application_nasnetlarge` *Instantiates a NASNet model in ImageNet mode.*

Description

Instantiates a NASNet model in ImageNet mode.

Usage

```
application_nasnetlarge(
  input_shape = NULL,
  include_top = TRUE,
  weights = "imagenet",
  input_tensor = NULL,
  pooling = NULL,
  classes = 1000L,
  classifier_activation = "softmax"
)
```

Arguments

`input_shape`	Optional shape tuple, only to be specified if `include_top` is `FALSE` (otherwise the input shape has to be `(331, 331, 3)` for NASNetLarge. It should have exactly 3 inputs channels, and width and height should be no smaller than 32. E.g. `(224, 224, 3)` would be one valid value.
`include_top`	Whether to include the fully-connected layer at the top of the network.
`weights`	`NULL` (random initialization) or `imagenet` (ImageNet weights). For loading `imagenet` weights, `input_shape` should be (331, 331, 3)
`input_tensor`	Optional Keras tensor (i.e. output of `layers.Input()`) to use as image input for the model.
`pooling`	Optional pooling mode for feature extraction when `include_top` is `FALSE`. • `NULL` means that the output of the model will be the 4D tensor output of the last convolutional layer. • `avg` means that global average pooling will be applied to the output of the last convolutional layer, and thus the output of the model will be a 2D tensor. • `max` means that global max pooling will be applied.
`classes`	Optional number of classes to classify images into, only to be specified if `include_top` is `TRUE`, and if no `weights` argument is specified.
`classifier_activation`	A `str` or callable. The activation function to use on the "top" layer. Ignored unless `include_top=TRUE`. Set `classifier_activation=NULL` to return the logits of the "top" layer. When loading pretrained weights, `classifier_activation` can only be `NULL` or `"softmax"`.

Value

A Keras model instance.

Reference

- Learning Transferable Architectures for Scalable Image Recognition (CVPR 2018)

Optionally loads weights pre-trained on ImageNet. Note that the data format convention used by the model is the one specified in your Keras config at `~/.keras/keras.json`.

Note

Each Keras Application expects a specific kind of input preprocessing. For NASNet, call `application_preprocess_inputs` on your inputs before passing them to the model.

See Also

- `https://keras.io/api/applications/nasnet#nasnetlarge-function`

application_nasnetmobile *Instantiates a Mobile NASNet model in ImageNet mode.*

Description

Instantiates a Mobile NASNet model in ImageNet mode.

Usage

```
application_nasnetmobile(
  input_shape = NULL,
  include_top = TRUE,
  weights = "imagenet",
  input_tensor = NULL,
  pooling = NULL,
  classes = 1000L,
  classifier_activation = "softmax"
)
```

Arguments

`input_shape`	Optional shape tuple, only to be specified if `include_top` is `FALSE` (otherwise the input shape has to be `(224, 224, 3)` for NASNetMobile It should have exactly 3 inputs channels, and width and height should be no smaller than 32. E.g. `(224, 224, 3)` would be one valid value.
`include_top`	Whether to include the fully-connected layer at the top of the network.
`weights`	`NULL` (random initialization) or `imagenet` (ImageNet weights). For loading `imagenet` weights, `input_shape` should be (224, 224, 3)
`input_tensor`	Optional Keras tensor (i.e. output of `layers.Input()`) to use as image input for the model.
`pooling`	Optional pooling mode for feature extraction when `include_top` is `FALSE`. • `NULL` means that the output of the model will be the 4D tensor output of the last convolutional layer. • `avg` means that global average pooling will be applied to the output of the last convolutional layer, and thus the output of the model will be a 2D tensor. • `max` means that global max pooling will be applied.
`classes`	Optional number of classes to classify images into, only to be specified if `include_top` is `TRUE`, and if no `weights` argument is specified.
`classifier_activation`	A `str` or callable. The activation function to use on the "top" layer. Ignored unless `include_top=TRUE`. Set `classifier_activation=NULL` to return the logits of the "top" layer. When loading pretrained weights, `classifier_activation` can only be `NULL` or `"softmax"`.

Value

A Keras model instance.

Reference

- Learning Transferable Architectures for Scalable Image Recognition (CVPR 2018)

Optionally loads weights pre-trained on ImageNet. Note that the data format convention used by the model is the one specified in your Keras config at `~/.keras/keras.json`.

Note

Each Keras Application expects a specific kind of input preprocessing. For NASNet, call `application_preprocess_inputs` on your inputs before passing them to the model.

See Also

- `https://keras.io/api/applications/nasnet#nasnetmobile-function`

`application_resnet101` *Instantiates the ResNet101 architecture.*

Description

Instantiates the ResNet101 architecture.

Usage

```
application_resnet101(
  include_top = TRUE,
  weights = "imagenet",
  input_tensor = NULL,
  input_shape = NULL,
  pooling = NULL,
  classes = 1000L,
  classifier_activation = "softmax"
)
```

Arguments

`include_top`	whether to include the fully-connected layer at the top of the network.
`weights`	one of `NULL` (random initialization), `"imagenet"` (pre-training on ImageNet), or the path to the weights file to be loaded.
`input_tensor`	optional Keras tensor (i.e. output of `layers.Input()`) to use as image input for the model.

`input_shape`	optional shape tuple, only to be specified if `include_top` is `FALSE` (otherwise the input shape has to be `(224, 224, 3)` (with `"channels_last"` data format) or `(3, 224, 224)` (with `"channels_first"` data format). It should have exactly 3 inputs channels, and width and height should be no smaller than 32. E.g. `(200, 200, 3)` would be one valid value.
`pooling`	Optional pooling mode for feature extraction when `include_top` is `FALSE`. • `NULL` means that the output of the model will be the 4D tensor output of the last convolutional block. • `avg` means that global average pooling will be applied to the output of the last convolutional block, and thus the output of the model will be a 2D tensor. • `max` means that global max pooling will be applied.
`classes`	optional number of classes to classify images into, only to be specified if `include_top` is `TRUE`, and if no `weights` argument is specified.
`classifier_activation`	A `str` or callable. The activation function to use on the "top" layer. Ignored unless `include_top=TRUE`. Set `classifier_activation=NULL` to return the logits of the "top" layer. When loading pretrained weights, `classifier_activation` can only be `NULL` or `"softmax"`.

Value

A Model instance.

Reference

- Deep Residual Learning for Image Recognition (CVPR 2015)

For image classification use cases, see this page for detailed examples.

For transfer learning use cases, make sure to read the guide to transfer learning & fine-tuning.

Note

Each Keras Application expects a specific kind of input preprocessing. For ResNet, call `application_preprocess_inputs(`
on your inputs before passing them to the model. `application_preprocess_inputs()` will convert the input images from RGB to BGR, then will zero-center each color channel with respect to the ImageNet dataset, without scaling.

See Also

- `https://keras.io/api/applications/resnet#resnet101-function`

application_resnet101_v2

Instantiates the ResNet101V2 architecture.

Description

Instantiates the ResNet101V2 architecture.

Usage

```
application_resnet101_v2(
  include_top = TRUE,
  weights = "imagenet",
  input_tensor = NULL,
  input_shape = NULL,
  pooling = NULL,
  classes = 1000L,
  classifier_activation = "softmax"
)
```

Arguments

`include_top`	whether to include the fully-connected layer at the top of the network.
`weights`	one of `NULL` (random initialization), `"imagenet"` (pre-training on ImageNet), or the path to the weights file to be loaded.
`input_tensor`	optional Keras tensor (i.e. output of `layers.Input()`) to use as image input for the model.
`input_shape`	optional shape tuple, only to be specified if `include_top` is `FALSE` (otherwise the input shape has to be `(224, 224, 3)` (with `"channels_last"` data format) or `(3, 224, 224)` (with `"channels_first"` data format). It should have exactly 3 inputs channels, and width and height should be no smaller than 32. E.g. `(200, 200, 3)` would be one valid value.
`pooling`	Optional pooling mode for feature extraction when `include_top` is `FALSE`. • `NULL` means that the output of the model will be the 4D tensor output of the last convolutional block. • `avg` means that global average pooling will be applied to the output of the last convolutional block, and thus the output of the model will be a 2D tensor. • `max` means that global max pooling will be applied.
`classes`	optional number of classes to classify images into, only to be specified if `include_top` is `TRUE`, and if no `weights` argument is specified.
`classifier_activation`	A `str` or callable. The activation function to use on the "top" layer. Ignored unless `include_top=TRUE`. Set `classifier_activation=NULL` to return the logits of the "top" layer. When loading pretrained weights, `classifier_activation` can only be `NULL` or `"softmax"`.

Value

A Model instance.

Reference

- Identity Mappings in Deep Residual Networks (CVPR 2016)

For image classification use cases, see this page for detailed examples.

For transfer learning use cases, make sure to read the guide to transfer learning & fine-tuning.

Note

Each Keras Application expects a specific kind of input preprocessing. For ResNet, call `application_preprocess_inputs(` on your inputs before passing them to the model. `application_preprocess_inputs()` will scale input pixels between -1 and 1.

See Also

- `https://keras.io/api/applications/resnet#resnet101v2-function`

`application_resnet152` *Instantiates the ResNet152 architecture.*

Description

Instantiates the ResNet152 architecture.

Usage

```
application_resnet152(
  include_top = TRUE,
  weights = "imagenet",
  input_tensor = NULL,
  input_shape = NULL,
  pooling = NULL,
  classes = 1000L,
  classifier_activation = "softmax"
)
```

Arguments

`include_top`	whether to include the fully-connected layer at the top of the network.
`weights`	one of `NULL` (random initialization), `"imagenet"` (pre-training on ImageNet), or the path to the weights file to be loaded.
`input_tensor`	optional Keras tensor (i.e. output of `layers.Input()`) to use as image input for the model.

`input_shape`	optional shape tuple, only to be specified if `include_top` is `FALSE` (otherwise the input shape has to be `(224, 224, 3)` (with `"channels_last"` data format) or `(3, 224, 224)` (with `"channels_first"` data format). It should have exactly 3 inputs channels, and width and height should be no smaller than 32. E.g. `(200, 200, 3)` would be one valid value.
`pooling`	Optional pooling mode for feature extraction when `include_top` is `FALSE`. • `NULL` means that the output of the model will be the 4D tensor output of the last convolutional block. • `avg` means that global average pooling will be applied to the output of the last convolutional block, and thus the output of the model will be a 2D tensor. • `max` means that global max pooling will be applied.
`classes`	optional number of classes to classify images into, only to be specified if `include_top` is `TRUE`, and if no `weights` argument is specified.
`classifier_activation`	A `str` or callable. The activation function to use on the "top" layer. Ignored unless `include_top=TRUE`. Set `classifier_activation=NULL` to return the logits of the "top" layer. When loading pretrained weights, `classifier_activation` can only be `NULL` or `"softmax"`.

Value

A Model instance.

Reference

- Deep Residual Learning for Image Recognition (CVPR 2015)

For image classification use cases, see this page for detailed examples.

For transfer learning use cases, make sure to read the guide to transfer learning & fine-tuning.

Note

Each Keras Application expects a specific kind of input preprocessing. For ResNet, call `application_preprocess_inputs(` on your inputs before passing them to the model. `application_preprocess_inputs()` will convert the input images from RGB to BGR, then will zero-center each color channel with respect to the ImageNet dataset, without scaling.

See Also

- `https://keras.io/api/applications/resnet#resnet152-function`

application_resnet152_v2

Instantiates the ResNet152V2 architecture.

Description

Instantiates the ResNet152V2 architecture.

Usage

```
application_resnet152_v2(
  include_top = TRUE,
  weights = "imagenet",
  input_tensor = NULL,
  input_shape = NULL,
  pooling = NULL,
  classes = 1000L,
  classifier_activation = "softmax"
)
```

Arguments

`include_top`	whether to include the fully-connected layer at the top of the network.
`weights`	one of `NULL` (random initialization), `"imagenet"` (pre-training on ImageNet), or the path to the weights file to be loaded.
`input_tensor`	optional Keras tensor (i.e. output of `layers.Input()`) to use as image input for the model.
`input_shape`	optional shape tuple, only to be specified if `include_top` is `FALSE` (otherwise the input shape has to be `(224, 224, 3)` (with `"channels_last"` data format) or `(3, 224, 224)` (with `"channels_first"` data format). It should have exactly 3 inputs channels, and width and height should be no smaller than 32. E.g. `(200, 200, 3)` would be one valid value.
`pooling`	Optional pooling mode for feature extraction when `include_top` is `FALSE`. • `NULL` means that the output of the model will be the 4D tensor output of the last convolutional block. • `avg` means that global average pooling will be applied to the output of the last convolutional block, and thus the output of the model will be a 2D tensor. • `max` means that global max pooling will be applied.
`classes`	optional number of classes to classify images into, only to be specified if `include_top` is `TRUE`, and if no `weights` argument is specified.
`classifier_activation`	A `str` or callable. The activation function to use on the "top" layer. Ignored unless `include_top=TRUE`. Set `classifier_activation=NULL` to return the logits of the "top" layer. When loading pretrained weights, `classifier_activation` can only be `NULL` or `"softmax"`.

Value

A Model instance.

Reference

- Identity Mappings in Deep Residual Networks (CVPR 2016)

For image classification use cases, see this page for detailed examples.

For transfer learning use cases, make sure to read the guide to transfer learning & fine-tuning.

Note

Each Keras Application expects a specific kind of input preprocessing. For ResNet, call `application_preprocess_inputs(` on your inputs before passing them to the model. `application_preprocess_inputs()` will scale input pixels between `-1` and `1`.

See Also

- `https://keras.io/api/applications/resnet#resnet152v2-function`

`application_resnet50` *Instantiates the ResNet50 architecture.*

Description

Instantiates the ResNet50 architecture.

Usage

```
application_resnet50(
  include_top = TRUE,
  weights = "imagenet",
  input_tensor = NULL,
  input_shape = NULL,
  pooling = NULL,
  classes = 1000L,
  classifier_activation = "softmax"
)
```

Arguments

`include_top`	whether to include the fully-connected layer at the top of the network.
`weights`	one of `NULL` (random initialization), `"imagenet"` (pre-training on ImageNet), or the path to the weights file to be loaded.
`input_tensor`	optional Keras tensor (i.e. output of `layers.Input()`) to use as image input for the model.

`input_shape`	optional shape tuple, only to be specified if `include_top` is `FALSE` (otherwise the input shape has to be `(224, 224, 3)` (with `"channels_last"` data format) or `(3, 224, 224)` (with `"channels_first"` data format). It should have exactly 3 inputs channels, and width and height should be no smaller than 32. E.g. `(200, 200, 3)` would be one valid value.
`pooling`	Optional pooling mode for feature extraction when `include_top` is `FALSE`. • `NULL` means that the output of the model will be the 4D tensor output of the last convolutional block. • `avg` means that global average pooling will be applied to the output of the last convolutional block, and thus the output of the model will be a 2D tensor. • `max` means that global max pooling will be applied.
`classes`	optional number of classes to classify images into, only to be specified if `include_top` is `TRUE`, and if no `weights` argument is specified.
`classifier_activation`	A `str` or callable. The activation function to use on the "top" layer. Ignored unless `include_top=TRUE`. Set `classifier_activation=NULL` to return the logits of the "top" layer. When loading pretrained weights, `classifier_activation` can only be `NULL` or `"softmax"`.

Value

A Model instance.

Reference

- Deep Residual Learning for Image Recognition (CVPR 2015)

For image classification use cases, see this page for detailed examples.

For transfer learning use cases, make sure to read the guide to transfer learning & fine-tuning.

Note

Each Keras Application expects a specific kind of input preprocessing. For ResNet, call `application_preprocess_inputs(` on your inputs before passing them to the model. `application_preprocess_inputs()` will convert the input images from RGB to BGR, then will zero-center each color channel with respect to the ImageNet dataset, without scaling.

See Also

- `https://keras.io/api/applications/resnet#resnet50-function`

`application_resnet50_v2` *Instantiates the ResNet50V2 architecture.*

Description

Instantiates the ResNet50V2 architecture.

Usage

```
application_resnet50_v2(
  include_top = TRUE,
  weights = "imagenet",
  input_tensor = NULL,
  input_shape = NULL,
  pooling = NULL,
  classes = 1000L,
  classifier_activation = "softmax"
)
```

Arguments

`include_top`	whether to include the fully-connected layer at the top of the network.
`weights`	one of `NULL` (random initialization), `"imagenet"` (pre-training on ImageNet), or the path to the weights file to be loaded.
`input_tensor`	optional Keras tensor (i.e. output of `layers.Input()`) to use as image input for the model.
`input_shape`	optional shape tuple, only to be specified if `include_top` is `FALSE` (otherwise the input shape has to be `(224, 224, 3)` (with `"channels_last"` data format) or `(3, 224, 224)` (with `"channels_first"` data format). It should have exactly 3 inputs channels, and width and height should be no smaller than 32. E.g. `(200, 200, 3)` would be one valid value.
`pooling`	Optional pooling mode for feature extraction when `include_top` is `FALSE`. • `NULL` means that the output of the model will be the 4D tensor output of the last convolutional block. • `avg` means that global average pooling will be applied to the output of the last convolutional block, and thus the output of the model will be a 2D tensor. • `max` means that global max pooling will be applied.
`classes`	optional number of classes to classify images into, only to be specified if `include_top` is `TRUE`, and if no `weights` argument is specified.
`classifier_activation`	A `str` or callable. The activation function to use on the "top" layer. Ignored unless `include_top=TRUE`. Set `classifier_activation=NULL` to return the logits of the "top" layer. When loading pretrained weights, `classifier_activation` can only be `NULL` or `"softmax"`.

Value

A Model instance.

Reference

- Identity Mappings in Deep Residual Networks (CVPR 2016)

For image classification use cases, see this page for detailed examples.

For transfer learning use cases, make sure to read the guide to transfer learning & fine-tuning.

Note

Each Keras Application expects a specific kind of input preprocessing. For ResNet, call `application_preprocess_inputs(` on your inputs before passing them to the model. `application_preprocess_inputs()` will scale input pixels between `-1` and `1`.

See Also

- `https://keras.io/api/applications/resnet#resnet50v2-function`

`application_vgg16` *Instantiates the VGG16 model.*

Description

Instantiates the VGG16 model.

Usage

```
application_vgg16(
  include_top = TRUE,
  weights = "imagenet",
  input_tensor = NULL,
  input_shape = NULL,
  pooling = NULL,
  classes = 1000L,
  classifier_activation = "softmax"
)
```

Arguments

`include_top`	whether to include the 3 fully-connected layers at the top of the network.
`weights`	one of `NULL` (random initialization), `"imagenet"` (pre-training on ImageNet), or the path to the weights file to be loaded.
`input_tensor`	optional Keras tensor (i.e. output of `layers.Input()`) to use as image input for the model.

`input_shape` optional shape tuple, only to be specified if `include_top` is `FALSE` (otherwise the input shape has to be `(224, 224, 3)` (with `channels_last` data format) or `(3, 224, 224)` (with `"channels_first"` data format). It should have exactly 3 input channels, and width and height should be no smaller than 32. E.g. `(200, 200, 3)` would be one valid value.

`pooling` Optional pooling mode for feature extraction when `include_top` is `FALSE`.

- `NULL` means that the output of the model will be the 4D tensor output of the last convolutional block.
- `avg` means that global average pooling will be applied to the output of the last convolutional block, and thus the output of the model will be a 2D tensor.
- `max` means that global max pooling will be applied.

`classes` optional number of classes to classify images into, only to be specified if `include_top` is `TRUE`, and if no `weights` argument is specified.

`classifier_activation`
A `str` or callable. The activation function to use on the "top" layer. Ignored unless `include_top=TRUE`. Set `classifier_activation=NULL` to return the logits of the "top" layer. When loading pretrained weights, `classifier_activation` can only be `NULL` or `"softmax"`.

Value

A model instance.

Reference

- Very Deep Convolutional Networks for Large-Scale Image Recognition (ICLR 2015)

For image classification use cases, see this page for detailed examples.

For transfer learning use cases, make sure to read the guide to transfer learning & fine-tuning.

The default input size for this model is 224x224.

Note

Each Keras Application expects a specific kind of input preprocessing. For VGG16, call `application_preprocess_inputs(` on your inputs before passing them to the model. `application_preprocess_inputs()` will convert the input images from RGB to BGR, then will zero-center each color channel with respect to the ImageNet dataset, without scaling.

See Also

- `https://keras.io/api/applications/vgg#vgg16-function`

application_vgg19 *Instantiates the VGG19 model.*

Description

Instantiates the VGG19 model.

Usage

```
application_vgg19(
  include_top = TRUE,
  weights = "imagenet",
  input_tensor = NULL,
  input_shape = NULL,
  pooling = NULL,
  classes = 1000L,
  classifier_activation = "softmax"
)
```

Arguments

`include_top`	whether to include the 3 fully-connected layers at the top of the network.
`weights`	one of `NULL` (random initialization), `"imagenet"` (pre-training on ImageNet), or the path to the weights file to be loaded.
`input_tensor`	optional Keras tensor (i.e. output of `layers.Input()`) to use as image input for the model.
`input_shape`	optional shape tuple, only to be specified if `include_top` is `FALSE` (otherwise the input shape has to be `(224, 224, 3)` (with `channels_last` data format) or `(3, 224, 224)` (with `"channels_first"` data format). It should have exactly 3 input channels, and width and height should be no smaller than 32. E.g. `(200, 200, 3)` would be one valid value.
`pooling`	Optional pooling mode for feature extraction when `include_top` is `FALSE`. • `NULL` means that the output of the model will be the 4D tensor output of the last convolutional block. • `avg` means that global average pooling will be applied to the output of the last convolutional block, and thus the output of the model will be a 2D tensor. • `max` means that global max pooling will be applied.
`classes`	optional number of classes to classify images into, only to be specified if `include_top` is `TRUE`, and if no `weights` argument is specified.
`classifier_activation`	A `str` or callable. The activation function to use on the "top" layer. Ignored unless `include_top=TRUE`. Set `classifier_activation=NULL` to return the logits of the "top" layer. When loading pretrained weights, `classifier_activation` can only be `NULL` or `"softmax"`.

Value

A model instance.

Reference

- Very Deep Convolutional Networks for Large-Scale Image Recognition (ICLR 2015)

For image classification use cases, see this page for detailed examples.

For transfer learning use cases, make sure to read the guide to transfer learning & fine-tuning.

The default input size for this model is 224x224.

Note

Each Keras Application expects a specific kind of input preprocessing. For VGG19, call `application_preprocess_inputs(` on your inputs before passing them to the model. `application_preprocess_inputs()` will convert the input images from RGB to BGR, then will zero-center each color channel with respect to the ImageNet dataset, without scaling.

See Also

- `https://keras.io/api/applications/vgg#vgg19-function`

`application_xception` *Instantiates the Xception architecture.*

Description

Instantiates the Xception architecture.

Usage

```
application_xception(
  include_top = TRUE,
  weights = "imagenet",
  input_tensor = NULL,
  input_shape = NULL,
  pooling = NULL,
  classes = 1000L,
  classifier_activation = "softmax"
)
```

Arguments

`include_top`	whether to include the 3 fully-connected layers at the top of the network.
`weights`	one of `NULL` (random initialization), `"imagenet"` (pre-training on ImageNet), or the path to the weights file to be loaded.
`input_tensor`	optional Keras tensor (i.e. output of `layers.Input()`) to use as image input for the model.
`input_shape`	optional shape tuple, only to be specified if `include_top` is `FALSE` (otherwise the input shape has to be `(299, 299, 3)`. It should have exactly 3 inputs channels, and width and height should be no smaller than 71. E.g. `(150, 150, 3)` would be one valid value.
`pooling`	Optional pooling mode for feature extraction when `include_top` is `FALSE`. • `NULL` means that the output of the model will be the 4D tensor output of the last convolutional block. • `avg` means that global average pooling will be applied to the output of the last convolutional block, and thus the output of the model will be a 2D tensor. • `max` means that global max pooling will be applied.
`classes`	optional number of classes to classify images into, only to be specified if `include_top` is `TRUE`, and if no `weights` argument is specified.
`classifier_activation`	A `str` or callable. The activation function to use on the "top" layer. Ignored unless `include_top=TRUE`. Set `classifier_activation=NULL` to return the logits of the "top" layer. When loading pretrained weights, `classifier_activation` can only be `NULL` or `"softmax"`.

Value

A model instance.

Reference

- Xception: Deep Learning with Depthwise Separable Convolutions (CVPR 2017)

For image classification use cases, see this page for detailed examples.

For transfer learning use cases, make sure to read the guide to transfer learning & fine-tuning.

The default input image size for this model is 299x299.

Note

Each Keras Application expects a specific kind of input preprocessing. For Xception, call `application_preprocess_inputs` on your inputs before passing them to the model. `application_preprocess_inputs()` will scale input pixels between `-1` and `1`.

See Also

- `https://keras.io/api/applications/xception#xception-function`

audio_dataset_from_directory

Generates a `tf.data.Dataset` from audio files in a directory.

Description

If your directory structure is:

```
main_directory/
...class_a/
......a_audio_1.wav
......a_audio_2.wav
...class_b/
......b_audio_1.wav
......b_audio_2.wav
```

Then calling `audio_dataset_from_directory(main_directory, labels = 'inferred')` will return a `tf.data.Dataset` that yields batches of audio files from the subdirectories `class_a` and `class_b`, together with labels 0 and 1 (0 corresponding to `class_a` and 1 corresponding to `class_b`).

Only `.wav` files are supported at this time.

Usage

```
audio_dataset_from_directory(
  directory,
  labels = "inferred",
  label_mode = "int",
  class_names = NULL,
  batch_size = 32L,
  sampling_rate = NULL,
  output_sequence_length = NULL,
  ragged = FALSE,
  shuffle = TRUE,
  seed = NULL,
  validation_split = NULL,
  subset = NULL,
  follow_links = FALSE,
  verbose = TRUE
)
```

Arguments

`directory`	Directory where the data is located. If `labels` is `"inferred"`, it should contain subdirectories, each containing audio files for a class. Otherwise, the directory structure is ignored.

`labels`	Either "inferred" (labels are generated from the directory structure), `NULL` (no labels), or a list/tuple of integer labels of the same size as the number of audio files found in the directory. Labels should be sorted according to the alphanumeric order of the audio file paths (obtained via `os.walk(directory)` in Python).
`label_mode`	String describing the encoding of `labels`. Options are: • `"int"`: means that the labels are encoded as integers (e.g. for `sparse_categorical_crossentropy` loss). • `"categorical"` means that the labels are encoded as a categorical vector (e.g. for `categorical_crossentropy` loss) • `"binary"` means that the labels (there can be only 2) are encoded as `float32` scalars with values 0 or 1 (e.g. for `binary_crossentropy`). • `NULL` (no labels).
`class_names`	Only valid if "labels" is `"inferred"`. This is the explicit list of class names (must match names of subdirectories). Used to control the order of the classes (otherwise alphanumerical order is used).
`batch_size`	Size of the batches of data. Default: 32. If `NULL`, the data will not be batched (the dataset will yield individual samples).
`sampling_rate`	Audio sampling rate (in samples per second).
`output_sequence_length`	Maximum length of an audio sequence. Audio files longer than this will be truncated to `output_sequence_length`. If set to `NULL`, then all sequences in the same batch will be padded to the length of the longest sequence in the batch.
`ragged`	Whether to return a Ragged dataset (where each sequence has its own length). Defaults to `FALSE`.
`shuffle`	Whether to shuffle the data. Defaults to `TRUE`. If set to `FALSE`, sorts the data in alphanumeric order.
`seed`	Optional random seed for shuffling and transformations.
`validation_split`	Optional float between 0 and 1, fraction of data to reserve for validation.
`subset`	Subset of the data to return. One of `"training"`, `"validation"` or `"both"`. Only used if `validation_split` is set.
`follow_links`	Whether to visits subdirectories pointed to by symlinks. Defaults to `FALSE`.
`verbose`	Whether to display number information on classes and number of files found. Defaults to `TRUE`.

Value

A `tf.data.Dataset` object.

- If `label_mode` is `NULL`, it yields `string` tensors of shape `(batch_size,)`, containing the contents of a batch of audio files.
- Otherwise, it yields a tuple `(audio, labels)`, where `audio` has shape `(batch_size, sequence_length, num_channe` and `labels` follows the format described below.

Rules regarding labels format:

- if label_mode is int, the labels are an int32 tensor of shape (batch_size,).
- if label_mode is binary, the labels are a float32 tensor of 1s and 0s of shape (batch_size, 1).
- if label_mode is categorical, the labels are a float32 tensor of shape (batch_size, num_classes), representing a one-hot encoding of the class index.

See Also

- https://keras.io/api/data_loading/audio#audiodatasetfromdirectory-function

Other dataset utils:
image_dataset_from_directory()
split_dataset()
text_dataset_from_directory()
timeseries_dataset_from_array()

Other utils:
clear_session()
config_disable_interactive_logging()
config_disable_traceback_filtering()
config_enable_interactive_logging()
config_enable_traceback_filtering()
config_is_interactive_logging_enabled()
config_is_traceback_filtering_enabled()
get_file()
get_source_inputs()
image_array_save()
image_dataset_from_directory()
image_from_array()
image_load()
image_smart_resize()
image_to_array()
layer_feature_space()
normalize()
pack_x_y_sample_weight()
pad_sequences()
set_random_seed()
split_dataset()
text_dataset_from_directory()
timeseries_dataset_from_array()
to_categorical()
unpack_x_y_sample_weight()
zip_lists()

`Callback`	*Define a custom* `Callback` *class*

Description

Callbacks can be passed to keras methods such as `fit()`, `evaluate()`, and `predict()` in order to hook into the various stages of the model training, evaluation, and inference lifecycle.

To create a custom callback, call `Callback()` and override the method associated with the stage of interest.

Usage

```
Callback(
  classname,
  on_epoch_begin = NULL,
  on_epoch_end = NULL,
  on_train_begin = NULL,
  on_train_end = NULL,
  on_train_batch_begin = NULL,
  on_train_batch_end = NULL,
  on_test_begin = NULL,
  on_test_end = NULL,
  on_test_batch_begin = NULL,
  on_test_batch_end = NULL,
  on_predict_begin = NULL,
  on_predict_end = NULL,
  on_predict_batch_begin = NULL,
  on_predict_batch_end = NULL,
  ...,
  public = list(),
  private = list(),
  inherit = NULL,
  parent_env = parent.frame()
)
```

Arguments

`classname` String, the name of the custom class. (Conventionally, CamelCase).

`on_epoch_begin` `\(epoch, logs = NULL)`

Called at the start of an epoch.

Subclasses should override for any actions to run. This function should only be called during TRAIN mode.

Args:

- `epoch`: Integer, index of epoch.
- `logs`: Named List. Currently no data is passed to this argument for this method but that may change in the future.

`on_epoch_end` `\(epoch, logs = NULL)`

Called at the end of an epoch.

Subclasses should override for any actions to run. This function should only be called during TRAIN mode.

Args:

- `epoch`: Integer, index of epoch.
- `logs`: Named List, metric results for this training epoch, and for the validation epoch if validation is performed. Validation result keys are prefixed with `val_`. For training epoch, the values of the `Model`'s metrics are returned. Example: `list(loss = 0.2, accuracy = 0.7)`.

`on_train_begin` `\(logs = NULL)`

Called at the beginning of training.

Subclasses should override for any actions to run.

Args:

- `logs`: Named list. Currently no data is passed to this argument for this method but that may change in the future.

`on_train_end` `\(logs = NULL)`

Called at the end of training.

Subclasses should override for any actions to run.

Args:

- `logs`: Named list. Currently the output of the last call to `on_epoch_end()` is passed to this argument for this method but that may change in the future.

`on_train_batch_begin`

`\(batch, logs = NULL)`

Called at the beginning of a training batch in `fit()` methods.

Subclasses should override for any actions to run.

Note that if the `steps_per_execution` argument to `compile` in `Model` is set to `N`, this method will only be called every `N` batches.

Args:

- `batch`: Integer, index of batch within the current epoch.
- `logs`: Named list. Currently no data is passed to this argument for this method but that may change in the future.

`on_train_batch_end`

`\(batch, logs=NULL)`

Called at the end of a training batch in `fit()` methods.

Subclasses should override for any actions to run.

Note that if the `steps_per_execution` argument to `compile` in `Model` is set to `N`, this method will only be called every `N` batches.

Args:

- `batch`: Integer, index of batch within the current epoch.
- `logs`: Named list. Aggregated metric results up until this batch.

`on_test_begin` `\(logs = NULL)`

Called at the beginning of evaluation or validation.

Subclasses should override for any actions to run.

Args:

- `logs`: Named list. Currently no data is passed to this argument for this method but that may change in the future.

`on_test_end` `\(logs = NULL)`

Called at the end of evaluation or validation.

Subclasses should override for any actions to run.

Args:

- `logs`: Named list. Currently the output of the last call to `on_test_batch_end()` is passed to this argument for this method but that may change in the future.

`on_test_batch_begin`

`\(batch, logs = NULL)`

Called at the beginning of a batch in `evaluate()` methods.

Also called at the beginning of a validation batch in the `fit()` methods, if validation data is provided.

Subclasses should override for any actions to run.

Note that if the `steps_per_execution` argument to `compile()` in `Model` is set to `N`, this method will only be called every `N` batches.

Args:

- `batch`: Integer, index of batch within the current epoch.
- `logs`: Named list. Currently no data is passed to this argument for this method but that may change in the future.

`on_test_batch_end`

`\(batch, logs = NULL)`

Called at the end of a batch in `evaluate()` methods.

Also called at the end of a validation batch in the `fit()` methods, if validation data is provided.

Subclasses should override for any actions to run.

Note that if the `steps_per_execution` argument to `compile()` in `Model` is set to `N`, this method will only be called every `N` batches.

Args:

- `batch`: Integer, index of batch within the current epoch.
- `logs`: Named list. Aggregated metric results up until this batch.

`on_predict_begin`

`\(logs = NULL)`

Called at the beginning of prediction.

Subclasses should override for any actions to run.

Args:

- `logs`: Named list. Currently no data is passed to this argument for this method but that may change in the future.

`on_predict_end` `\(logs = NULL)`

Called at the end of prediction.

Subclasses should override for any actions to run.

Args:

- `logs`: Named list. Currently no data is passed to this argument for this method but that may change in the future.

`on_predict_batch_begin`

`\(batch, logs = NULL)`

Called at the beginning of a batch in `predict()` methods.

Subclasses should override for any actions to run.

Note that if the `steps_per_execution` argument to `compile()` in `Model` is set to `N`, this method will only be called every `N` batches.

Args:

- `batch`: Integer, index of batch within the current epoch.
- `logs`: Named list. Currently no data is passed to this argument for this method but that may change in the future.

`on_predict_batch_end`

`\(batch, logs = NULL)`

Called at the end of a batch in `predict()` methods.

Subclasses should override for any actions to run.

Note that if the `steps_per_execution` argument to `compile` in `Model` is set to `N`, this method will only be called every `N` batches.

Args:

- `batch`: Integer, index of batch within the current epoch.
- `logs`: Named list. Aggregated metric results up until this batch.

`..., public` Additional methods or public members of the custom class.

`private` Named list of R objects (typically, functions) to include in instance private environments. `private` methods will have all the same symbols in scope as public methods (See section "Symbols in Scope"). Each instance will have it's own `private` environment. Any objects in `private` will be invisible from the Keras framework and the Python runtime.

`inherit` What the custom class will subclass. By default, the base keras class.

`parent_env` The R environment that all class methods will have as a grandparent.

Value

A function that returns the custom `Callback` instances, similar to the builtin callback functions.

Examples

```
training_finished <- FALSE
callback_mark_finished <- Callback("MarkFinished",
  on_train_end = function(logs = NULL) {
    training_finished <<- TRUE
  }
)

model <- keras_model_sequential(input_shape = c(1)) |>
  layer_dense(1)
model |> compile(loss = 'mean_squared_error')
model |> fit(op_ones(c(1, 1)), op_ones(c(1, 1)),
             callbacks = callback_mark_finished())
stopifnot(isTRUE(training_finished))
```

All R function custom methods (public and private) will have the following symbols in scope:

- `self`: the `Layer` instance.
- `super`: the `Layer` superclass.
- `private`: An R environment specific to the class instance. Any objects defined here will be invisible to the Keras framework.
- `__class__` the current class type object. This will also be available as an alias symbol, the value supplied to `Layer(classname = )`

Attributes (accessible via `self$`)

- `params`: Named list, Training parameters (e.g. verbosity, batch size, number of epochs, ...).
- `model`: Instance of `Model`. Reference of the model being trained.

The `logs` named list that callback methods take as argument will contain keys for quantities relevant to the current batch or epoch (see method-specific docstrings).

Symbols in scope

All R function custom methods (public and private) will have the following symbols in scope:

- `self`: The custom class instance.
- `super`: The custom class superclass.
- `private`: An R environment specific to the class instance. Any objects assigned here are invisible to the Keras framework.
- `__class__` and `as.symbol(classname)`: the custom class type object.

See Also

- `https://keras.io/api/callbacks/base_callback#callback-class`

Other callbacks:
`callback_backup_and_restore()`
`callback_csv_logger()`
`callback_early_stopping()`
`callback_lambda()`
`callback_learning_rate_scheduler()`
`callback_model_checkpoint()`
`callback_reduce_lr_on_plateau()`
`callback_remote_monitor()`

`callback_swap_ema_weights()`
`callback_tensorboard()`
`callback_terminate_on_nan()`

`callback_backup_and_restore`
Callback to back up and restore the training state.

Description

`callback_backup_and_restore()` callback is intended to recover training from an interruption that has happened in the middle of a `fit` execution, by backing up the training states in a temporary checkpoint file, at the end of each epoch. Each backup overwrites the previously written checkpoint file, so at any given time there is at most one such checkpoint file for backup/restoring purpose.

If training restarts before completion, the training state (which includes the model weights and epoch number) is restored to the most recently saved state at the beginning of a new `fit` run. At the completion of a `fit` run, the temporary checkpoint file is deleted.

Note that the user is responsible to bring jobs back after the interruption. This callback is important for the backup and restore mechanism for fault tolerance purpose, and the model to be restored from a previous checkpoint is expected to be the same as the one used to back up. If user changes arguments passed to `compile` or `fit`, the checkpoint saved for fault tolerance can become invalid.

Usage

```
callback_backup_and_restore(
  backup_dir,
  save_freq = "epoch",
  delete_checkpoint = TRUE
)
```

Arguments

`backup_dir`	String, path of directory where to store the data needed to restore the model. The directory cannot be reused elsewhere to store other files, e.g. by the `backup_and_restore` callback of another training run, or by another callback (e.g. `callback_model_checkpoint`) of the same training run.
`save_freq`	`"epoch"`, integer, or `FALSE`. When set to `"epoch"`, the callback saves the checkpoint at the end of each epoch. When set to an integer, the callback saves the checkpoint every `save_freq` batches. Set `save_freq = FALSE` only if using preemption checkpointing (i.e. with `save_before_preemption = TRUE`).
`delete_checkpoint`	Boolean, defaults to `TRUE`. This `backup_and_restore` callback works by saving a checkpoint to back up the training state. If `delete_checkpoint = TRUE`, the checkpoint will be deleted after training is finished. Use `FALSE` if you'd like to keep the checkpoint for future usage.

Value

A Callback instance that can be passed to fit.keras.src.models.model.Model().

Examples

```
callback_interrupting <- new_callback_class(
  "InterruptingCallback",
  on_epoch_begin = function(epoch, logs = NULL) {
    if (epoch == 4) {
      stop('Interrupting!')
    }
  }
)

backup_dir <- tempfile()
callback <- callback_backup_and_restore(backup_dir = backup_dir)
model <- keras_model_sequential() %>%
  layer_dense(10)
model %>% compile(optimizer = optimizer_sgd(), loss = 'mse')

tryCatch({
  model %>% fit(x = op_ones(c(5, 20)),
                y = op_zeros(5),
                epochs = 10, batch_size = 1,
                callbacks = list(callback, callback_interrupting()),
                verbose = 0)
}, python.builtin.RuntimeError = function(e) message("Interrupted!"))

## Interrupted!

model$history$epoch

## [1] 0 1 2


# model$history %>% keras3:::to_keras_training_history() %>% as.data.frame() %>% print()

history <- model %>% fit(x = op_ones(c(5, 20)),
                         y = op_zeros(5),
                         epochs = 10, batch_size = 1,
                         callbacks = list(callback),
                         verbose = 0)

# Only 6 more epochs are run, since first training got interrupted at
# zero-indexed epoch 4, second training will continue from 4 to 9.
nrow(as.data.frame(history))

## [1] 10
```

See Also

- https://keras.io/api/callbacks/backup_and_restore#backupandrestore-class

Other callbacks:
`Callback()`
`callback_csv_logger()`
`callback_early_stopping()`
`callback_lambda()`
`callback_learning_rate_scheduler()`
`callback_model_checkpoint()`
`callback_reduce_lr_on_plateau()`
`callback_remote_monitor()`
`callback_swap_ema_weights()`
`callback_tensorboard()`
`callback_terminate_on_nan()`

`callback_csv_logger` *Callback that streams epoch results to a CSV file.*

Description

Supports all values that can be represented as a string, including 1D iterables such as atomic vectors.

Usage

```
callback_csv_logger(filename, separator = ",", append = FALSE)
```

Arguments

`filename`	Filename of the CSV file, e.g. `'run/log.csv'`.
`separator`	String used to separate elements in the CSV file.
`append`	Boolean. `TRUE`: append if file exists (useful for continuing training). `FALSE`: overwrite existing file.

Value

A `Callback` instance that can be passed to `fit.keras.src.models.model.Model()`.

Examples

```
csv_logger <- callback_csv_logger('training.log')
model %>% fit(X_train, Y_train, callbacks = list(csv_logger))
```

See Also

- https://keras.io/api/callbacks/csv_logger#csvlogger-class

Other callbacks:
Callback()
callback_backup_and_restore()
callback_early_stopping()
callback_lambda()
callback_learning_rate_scheduler()
callback_model_checkpoint()
callback_reduce_lr_on_plateau()
callback_remote_monitor()
callback_swap_ema_weights()
callback_tensorboard()
callback_terminate_on_nan()

---

callback_early_stopping

Stop training when a monitored metric has stopped improving.

Description

Assuming the goal of a training is to minimize the loss. With this, the metric to be monitored would be 'loss', and mode would be 'min'. A model$fit() training loop will check at end of every epoch whether the loss is no longer decreasing, considering the min_delta and patience if applicable. Once it's found no longer decreasing, model$stop_training is marked TRUE and the training terminates.

The quantity to be monitored needs to be available in logs list. To make it so, pass the loss or metrics at model$compile().

Usage

```
callback_early_stopping(
  monitor = "val_loss",
  min_delta = 0L,
  patience = 0L,
  verbose = 0L,
  mode = "auto",
  baseline = NULL,
  restore_best_weights = FALSE,
  start_from_epoch = 0L
)
```

Arguments

`monitor`	Quantity to be monitored. Defaults to `"val_loss"`.
`min_delta`	Minimum change in the monitored quantity to qualify as an improvement, i.e. an absolute change of less than min_delta, will count as no improvement. Defaults to `0`.
`patience`	Number of epochs with no improvement after which training will be stopped. Defaults to `0`.
`verbose`	Verbosity mode, 0 or 1. Mode 0 is silent, and mode 1 displays messages when the callback takes an action. Defaults to `0`.
`mode`	One of `{"auto", "min", "max"}`. In `min` mode, training will stop when the quantity monitored has stopped decreasing; in `"max"` mode it will stop when the quantity monitored has stopped increasing; in `"auto"` mode, the direction is automatically inferred from the name of the monitored quantity. Defaults to `"auto"`.
`baseline`	Baseline value for the monitored quantity. If not `NULL`, training will stop if the model doesn't show improvement over the baseline. Defaults to `NULL`.
`restore_best_weights`	Whether to restore model weights from the epoch with the best value of the monitored quantity. If `FALSE`, the model weights obtained at the last step of training are used. An epoch will be restored regardless of the performance relative to the `baseline`. If no epoch improves on `baseline`, training will run for `patience` epochs and restore weights from the best epoch in that set. Defaults to `FALSE`.
`start_from_epoch`	Number of epochs to wait before starting to monitor improvement. This allows for a warm-up period in which no improvement is expected and thus training will not be stopped. Defaults to `0`.

Value

A `Callback` instance that can be passed to `fit.keras.src.models.model.Model()`.

Examples

```
callback <- callback_early_stopping(monitor = 'loss',
                                    patience = 3)
# This callback will stop the training when there is no improvement in
# the loss for three consecutive epochs.
model <- keras_model_sequential() %>%
  layer_dense(10)
model %>% compile(optimizer = optimizer_sgd(), loss = 'mse')
history <- model %>% fit(x = op_ones(c(5, 20)),
                         y = op_zeros(5),
                         epochs = 10, batch_size = 1,
                         callbacks = list(callback),
                         verbose = 0)
nrow(as.data.frame(history))  # Only 4 epochs are run.
```

```
## [1] 10
```

See Also

- https://keras.io/api/callbacks/early_stopping#earlystopping-class

Other callbacks:
`Callback()`
`callback_backup_and_restore()`
`callback_csv_logger()`
`callback_lambda()`
`callback_learning_rate_scheduler()`
`callback_model_checkpoint()`
`callback_reduce_lr_on_plateau()`
`callback_remote_monitor()`
`callback_swap_ema_weights()`
`callback_tensorboard()`
`callback_terminate_on_nan()`

`callback_lambda` *Callback for creating simple, custom callbacks on-the-fly.*

Description

This callback is constructed with anonymous functions that will be called at the appropriate time (during `Model.{fit | evaluate | predict}`). Note that the callbacks expects positional arguments, as:

- `on_epoch_begin` and `on_epoch_end` expect two positional arguments: `epoch`, `logs`
- `on_train_begin` and `on_train_end` expect one positional argument: `logs`
- `on_train_batch_begin` and `on_train_batch_end` expect two positional arguments: `batch`, `logs`
- See `Callback` class definition for the full list of functions and their expected arguments.

Usage

```
callback_lambda(
  on_epoch_begin = NULL,
  on_epoch_end = NULL,
  on_train_begin = NULL,
  on_train_end = NULL,
  on_train_batch_begin = NULL,
  on_train_batch_end = NULL,
  ...
)
```

Arguments

on_epoch_begin	called at the beginning of every epoch.
on_epoch_end	called at the end of every epoch.
on_train_begin	called at the beginning of model training.
on_train_end	called at the end of model training.
on_train_batch_begin	called at the beginning of every train batch.
on_train_batch_end	called at the end of every train batch.
...	Any function in Callback() that you want to override by passing function_name = function. For example, callback_lambda(.., on_train_end = train_end_fn). The custom function needs to have same arguments as the ones defined in Callback().

Value

A Callback instance that can be passed to fit.keras.src.models.model.Model().

Examples

```
# Print the batch number at the beginning of every batch.
batch_print_callback <- callback_lambda(
  on_train_batch_begin = function(batch, logs) {
    print(batch)
  }
)

# Stream the epoch loss to a file in new-line delimited JSON format
# (one valid JSON object per line)
json_log <- file('loss_log.json', open = 'wt')
json_logging_callback <- callback_lambda(
  on_epoch_end = function(epoch, logs) {
    jsonlite::write_json(
      list(epoch = epoch, loss = logs$loss),
      json_log,
      append = TRUE
    )
  },
  on_train_end = function(logs) {
    close(json_log)
  }
)

# Terminate some processes after having finished model training.
processes <- ...
cleanup_callback <- callback_lambda(
  on_train_end = function(logs) {
    for (p in processes) {
      if (is_alive(p)) {
        terminate(p)
      }
    }
  }
)

model %>% fit(
  ...,
  callbacks = list(
    batch_print_callback,
    json_logging_callback,
    cleanup_callback
  )
)
```

See Also

- https://keras.io/api/callbacks/lambda_callback#lambdacallback-class

Other callbacks:
`Callback()`
`callback_backup_and_restore()`
`callback_csv_logger()`
`callback_early_stopping()`
`callback_learning_rate_scheduler()`
`callback_model_checkpoint()`
`callback_reduce_lr_on_plateau()`
`callback_remote_monitor()`
`callback_swap_ema_weights()`
`callback_tensorboard()`
`callback_terminate_on_nan()`

---

callback_learning_rate_scheduler

Learning rate scheduler.

Description

At the beginning of every epoch, this callback gets the updated learning rate value from `schedule` function provided, with the current epoch and current learning rate, and applies the updated learning rate on the optimizer.

Usage

```
callback_learning_rate_scheduler(schedule, verbose = 0L)
```

Arguments

schedule	A function that takes an epoch index (integer, indexed from 0) and current learning rate (float) as inputs and returns a new learning rate as output (float).
verbose	Integer. 0: quiet, 1: log update messages.

Value

A Callback instance that can be passed to fit.keras.src.models.model.Model().

Examples

```
# This function keeps the initial learning rate steady for the first ten epochs
# and decreases it exponentially after that.
scheduler <- function(epoch, lr) {
  if (epoch < 10)
    return(lr)
  else
    return(lr * exp(-0.1))
}

model <- keras_model_sequential() |> layer_dense(units = 10)
model |> compile(optimizer = optimizer_sgd(), loss = 'mse')
model$optimizer$learning_rate |> as.array() |> round(5)

## [1] 0.01


callback <- callback_learning_rate_scheduler(schedule = scheduler)
history <- model |> fit(x = array(runif(100), c(5, 20)),
                        y = array(0, c(5, 1)),
                        epochs = 15, callbacks = list(callback), verbose = 0)
model$optimizer$learning_rate |> as.array() |> round(5)

## [1] 0.00607
```

See Also

- https://keras.io/api/callbacks/learning_rate_scheduler#learningratescheduler-class

Other callbacks:
Callback()
callback_backup_and_restore()
callback_csv_logger()
callback_early_stopping()
callback_lambda()
callback_model_checkpoint()
callback_reduce_lr_on_plateau()
callback_remote_monitor()

callback_swap_ema_weights()
callback_tensorboard()
callback_terminate_on_nan()

---

callback_model_checkpoint

Callback to save the Keras model or model weights at some frequency.

Description

`callback_model_checkpoint()` is used in conjunction with training using `model |> fit()` to save a model or weights (in a checkpoint file) at some interval, so the model or weights can be loaded later to continue the training from the state saved.

A few options this callback provides include:

- Whether to only keep the model that has achieved the "best performance" so far, or whether to save the model at the end of every epoch regardless of performance.
- Definition of "best"; which quantity to monitor and whether it should be maximized or minimized.
- The frequency it should save at. Currently, the callback supports saving at the end of every epoch, or after a fixed number of training batches.
- Whether only weights are saved, or the whole model is saved.

Usage

```
callback_model_checkpoint(
  filepath,
  monitor = "val_loss",
  verbose = 0L,
  save_best_only = FALSE,
  save_weights_only = FALSE,
  mode = "auto",
  save_freq = "epoch",
  initial_value_threshold = NULL
)
```

Arguments

`filepath`	string, path to save the model file. `filepath` can contain named formatting options, which will be filled the value of `epoch` and keys in `logs` (passed in `on_epoch_end`). The `filepath` name needs to end with `".weights.h5"` when `save_weights_only = TRUE` or should end with `".keras"` when checkpoint saving the whole model (default). For example: if `filepath` is `"{epoch:02d}-{val_loss:.2f}.keras"` then the model checkpoints will be saved with the epoch number and the validation loss in the filename. The directory of the filepath should not be reused by any other callbacks to avoid conflicts.

`monitor`	The metric name to monitor. Typically the metrics are set by the `model |> compile()` method. Note: • Prefix the name with `"val_"` to monitor validation metrics. • Use `"loss"` or `"val_loss"` to monitor the model's total loss. • If you specify metrics as strings, like `"accuracy"`, pass the same string (with or without the `"val_"` prefix). • If you pass `Metric` objects (created by one of `metric_*()`), `monitor` should be set to `metric$name`. • If you're not sure about the metric names you can check the contents of the `history$metrics` list returned by `history <- model |> fit()` • Multi-output models set additional prefixes on the metric names.
`verbose`	Verbosity mode, 0 or 1. Mode 0 is silent, and mode 1 displays messages when the callback takes an action.
`save_best_only`	if `save_best_only = TRUE`, it only saves when the model is considered the "best" and the latest best model according to the quantity monitored will not be overwritten. If `filepath` doesn't contain formatting options like `{epoch}` then `filepath` will be overwritten by each new better model.
`save_weights_only`	if TRUE, then only the model's weights will be saved (`model |> save_model_weights(filepath)`), else the full model is saved (`model |> save_model(filepath)`).
`mode`	one of {`"auto"`, `"min"`, `"max"`}. If `save_best_only = TRUE`, the decision to overwrite the current save file is made based on either the maximization or the minimization of the monitored quantity. For `val_acc`, this should be `"max"`, for `val_loss` this should be `"min"`, etc. In `"auto"` mode, the mode is set to `"max"` if the quantities monitored are `"acc"` or start with `"fmeasure"` and are set to `"min"` for the rest of the quantities.
`save_freq`	`"epoch"` or integer. When using `"epoch"`, the callback saves the model after each epoch. When using integer, the callback saves the model at end of this many batches. If the `Model` is compiled with `steps_per_execution = N`, then the saving criteria will be checked every Nth batch. Note that if the saving isn't aligned to epochs, the monitored metric may potentially be less reliable (it could reflect as little as 1 batch, since the metrics get reset every epoch). Defaults to `"epoch"`.
`initial_value_threshold`	Floating point initial "best" value of the metric to be monitored. Only applies if `save_best_value = TRUE`. Only overwrites the model weights already saved if the performance of current model is better than this value.

Value

A `Callback` instance that can be passed to `fit.keras.src.models.model.Model()`.

Examples

```
model <- keras_model_sequential(input_shape = c(10)) |>
  layer_dense(1, activation = "sigmoid") |>
  compile(loss = "binary_crossentropy", optimizer = "adam",
          metrics = c('accuracy'))

EPOCHS <- 10
checkpoint_filepath <- tempfile('checkpoint-model-', fileext = ".keras")
model_checkpoint_callback <- callback_model_checkpoint(
  filepath = checkpoint_filepath,
  monitor = 'val_accuracy',
  mode = 'max',
  save_best_only = TRUE
)

# Model is saved at the end of every epoch, if it's the best seen so far.
model |> fit(x = random_uniform(c(2, 10)), y = op_ones(2, 1),
             epochs = EPOCHS, validation_split = .5, verbose = 0,
             callbacks = list(model_checkpoint_callback))

# The model (that are considered the best) can be loaded as -
load_model(checkpoint_filepath)

## Model: "sequential"
## +---------------------------------+------------------------+---------------+
## | Layer (type)                    | Output Shape           |       Param # |
## +=================================+========================+===============+
## | dense (Dense)                   | (None, 1)              |            11 |
## +---------------------------------+------------------------+---------------+
##  Total params: 35 (144.00 B)
##  Trainable params: 11 (44.00 B)
##  Non-trainable params: 0 (0.00 B)
##  Optimizer params: 24 (100.00 B)


# Alternatively, one could checkpoint just the model weights as -
checkpoint_filepath <- tempfile('checkpoint-', fileext = ".weights.h5")
model_checkpoint_callback <- callback_model_checkpoint(
  filepath = checkpoint_filepath,
  save_weights_only = TRUE,
  monitor = 'val_accuracy',
  mode = 'max',
  save_best_only = TRUE
)

# Model weights are saved at the end of every epoch, if it's the best seen
# so far.
# same as above
model |> fit(x = random_uniform(c(2, 10)), y = op_ones(2, 1),
             epochs = EPOCHS, validation_split = .5, verbose = 0,
             callbacks = list(model_checkpoint_callback))
```

```
# The model weights (that are considered the best) can be loaded
model |> load_model_weights(checkpoint_filepath)
```

See Also

- https://keras.io/api/callbacks/model_checkpoint#modelcheckpoint-class

Other callbacks:
Callback()
callback_backup_and_restore()
callback_csv_logger()
callback_early_stopping()
callback_lambda()
callback_learning_rate_scheduler()
callback_reduce_lr_on_plateau()
callback_remote_monitor()
callback_swap_ema_weights()
callback_tensorboard()
callback_terminate_on_nan()

callback_reduce_lr_on_plateau

Reduce learning rate when a metric has stopped improving.

Description

Models often benefit from reducing the learning rate by a factor of 2-10 once learning stagnates. This callback monitors a quantity and if no improvement is seen for a 'patience' number of epochs, the learning rate is reduced.

Usage

```
callback_reduce_lr_on_plateau(
  monitor = "val_loss",
  factor = 0.1,
  patience = 10L,
  verbose = 0L,
  mode = "auto",
  min_delta = 1e-04,
  cooldown = 0L,
  min_lr = 0,
  ...
)
```

Arguments

monitor	String. Quantity to be monitored.
factor	Float. Factor by which the learning rate will be reduced. new_lr = lr * factor.
patience	Integer. Number of epochs with no improvement after which learning rate will be reduced.
verbose	Integer. 0: quiet, 1: update messages.
mode	String. One of {'auto', 'min', 'max'}. In 'min' mode, the learning rate will be reduced when the quantity monitored has stopped decreasing; in 'max' mode it will be reduced when the quantity monitored has stopped increasing; in 'auto' mode, the direction is automatically inferred from the name of the monitored quantity.
min_delta	Float. Threshold for measuring the new optimum, to only focus on significant changes.
cooldown	Integer. Number of epochs to wait before resuming normal operation after the learning rate has been reduced.
min_lr	Float. Lower bound on the learning rate.
...	For forward/backward compatability.

Value

A Callback instance that can be passed to fit.keras.src.models.model.Model().

Examples

```
reduce_lr <- callback_reduce_lr_on_plateau(monitor = 'val_loss', factor = 0.2,
                                           patience = 5, min_lr = 0.001)
model %>% fit(x_train, y_train, callbacks = list(reduce_lr))
```

See Also

- https://keras.io/api/callbacks/reduce_lr_on_plateau#reducelronplateau-class

Other callbacks:
Callback()
callback_backup_and_restore()
callback_csv_logger()
callback_early_stopping()
callback_lambda()
callback_learning_rate_scheduler()
callback_model_checkpoint()
callback_remote_monitor()
callback_swap_ema_weights()
callback_tensorboard()
callback_terminate_on_nan()

callback_remote_monitor

Callback used to stream events to a server.

Description

Requires the requests library. Events are sent to root + '/publish/epoch/end/' by default. Calls are HTTP POST, with a data argument which is a JSON-encoded named list of event data. If send_as_json = TRUE, the content type of the request will be "application/json". Otherwise the serialized JSON will be sent within a form.

Usage

```
callback_remote_monitor(
  root = "http://localhost:9000",
  path = "/publish/epoch/end/",
  field = "data",
  headers = NULL,
  send_as_json = FALSE
)
```

Arguments

root	String; root url of the target server.
path	String; path relative to root to which the events will be sent.
field	String; JSON field under which the data will be stored. The field is used only if the payload is sent within a form (i.e. when send_as_json = FALSE).
headers	Named list; optional custom HTTP headers.
send_as_json	Boolean; whether the request should be sent as "application/json".

Value

A Callback instance that can be passed to fit.keras.src.models.model.Model().

See Also

- https://keras.io/api/callbacks/remote_monitor#remotemonitor-class

Other callbacks:
Callback()
callback_backup_and_restore()
callback_csv_logger()
callback_early_stopping()
callback_lambda()
callback_learning_rate_scheduler()
callback_model_checkpoint()
callback_reduce_lr_on_plateau()

callback_swap_ema_weights()
callback_tensorboard()
callback_terminate_on_nan()

---

callback_swap_ema_weights *Swaps model weights and EMA weights before and after evaluation.*

Description

This callbacks replaces the model's weight values with the values of the optimizer's EMA weights (the exponential moving average of the past model weights values, implementing "Polyak averaging") before model evaluation, and restores the previous weights after evaluation.

The SwapEMAWeights callback is to be used in conjunction with an optimizer that sets use_ema = TRUE.

Note that the weights are swapped in-place in order to save memory. The behavior is undefined if you modify the EMA weights or model weights in other callbacks.

Usage

```
callback_swap_ema_weights(swap_on_epoch = FALSE)
```

Arguments

swap_on_epoch	Whether to perform swapping at on_epoch_begin() and on_epoch_end(). This is useful if you want to use EMA weights for other callbacks such as callback_model_checkpoint(). Defaults to FALSE.

Value

A Callback instance that can be passed to fit.keras.src.models.model.Model().

Examples

```
# Remember to set `use_ema=TRUE` in the optimizer
optimizer <- optimizer_sgd(use_ema = TRUE)
model |> compile(optimizer = optimizer, loss = ..., metrics = ...)

# Metrics will be computed with EMA weights
model |> fit(X_train, Y_train,
             callbacks = c(callback_swap_ema_weights()))

# If you want to save model checkpoint with EMA weights, you can set
# `swap_on_epoch=TRUE` and place ModelCheckpoint after SwapEMAWeights.
model |> fit(
  X_train, Y_train,
  callbacks = c(
    callback_swap_ema_weights(swap_on_epoch = TRUE),
    callback_model_checkpoint(...)
  )
)
```

See Also

Other callbacks:
`Callback()`
`callback_backup_and_restore()`
`callback_csv_logger()`
`callback_early_stopping()`
`callback_lambda()`
`callback_learning_rate_scheduler()`
`callback_model_checkpoint()`
`callback_reduce_lr_on_plateau()`
`callback_remote_monitor()`
`callback_tensorboard()`
`callback_terminate_on_nan()`

`callback_tensorboard` *Enable visualizations for TensorBoard.*

Description

TensorBoard is a visualization tool provided with TensorFlow. A TensorFlow installation is required to use this callback.

This callback logs events for TensorBoard, including:

- Metrics summary plots
- Training graph visualization
- Weight histograms
- Sampled profiling

When used in `model |> evaluate()` or regular validation in addition to epoch summaries, there will be a summary that records evaluation metrics vs `model$optimizer$iterations` written. The metric names will be prepended with `evaluation`, with `model$optimizer$iterations` being the step in the visualized TensorBoard.

If you have installed TensorFlow with `pip` or `reticulate::py_install()`, you should be able to launch TensorBoard from the command line:

```
tensorboard --logdir=path_to_your_logs
```

or from R with `tensorflow::tensorboard()`.

You can find more information about TensorBoard here.

Usage

```
callback_tensorboard(
  log_dir = "logs",
  histogram_freq = 0L,
  write_graph = TRUE,
  write_images = FALSE,
  write_steps_per_second = FALSE,
  update_freq = "epoch",
  profile_batch = 0L,
  embeddings_freq = 0L,
  embeddings_metadata = NULL
)
```

Arguments

`log_dir`	the path of the directory where to save the log files to be parsed by TensorBoard. e.g., `log_dir = file.path(working_dir, 'logs')`. This directory should not be reused by any other callbacks.
`histogram_freq`	frequency (in epochs) at which to compute weight histograms for the layers of the model. If set to 0, histograms won't be computed. Validation data (or split) must be specified for histogram visualizations.
`write_graph`	(Not supported at this time) Whether to visualize the graph in TensorBoard. Note that the log file can become quite large when `write_graph` is set to `TRUE`.
`write_images`	whether to write model weights to visualize as image in TensorBoard.
`write_steps_per_second`	whether to log the training steps per second into TensorBoard. This supports both epoch and batch frequency logging.
`update_freq`	`"batch"` or `"epoch"` or integer. When using `"epoch"`, writes the losses and metrics to TensorBoard after every epoch. If using an integer, let's say `1000`, all metrics and losses (including custom ones added by `Model.compile`) will be logged to TensorBoard every 1000 batches. `"batch"` is a synonym for 1, meaning that they will be written every batch. Note however that writing too frequently to TensorBoard can slow down your training, especially when used with distribution strategies as it will incur additional synchronization overhead. Batch-level summary writing is also available via `train_step` override. Please see TensorBoard Scalars tutorial # noqa: E501 for more details.
`profile_batch`	(Not supported at this time) Profile the batch(es) to sample compute characteristics. profile_batch must be a non-negative integer or a tuple of integers. A pair of positive integers signify a range of batches to profile. By default, profiling is disabled.
`embeddings_freq`	frequency (in epochs) at which embedding layers will be visualized. If set to 0, embeddings won't be visualized.
`embeddings_metadata`	Named list which maps embedding layer names to the filename of a file in which to save metadata for the embedding layer. In case the same metadata file is to be used for all embedding layers, a single filename can be passed.

Value

A Callback instance that can be passed to fit.keras.src.models.model.Model().

Examples

```
tensorboard_callback <- callback_tensorboard(log_dir = "./logs")
model %>% fit(x_train, y_train, epochs = 2, callbacks = list(tensorboard_callback))
# Then run the tensorboard command to view the visualizations.
```

Custom batch-level summaries in a subclassed Model:

```
MyModel <- new_model_class("MyModel",
  initialize = function() {
    self$dense <- layer_dense(units = 10)
  },
  call = function(x) {
    outputs <- x |> self$dense()
    tf$summary$histogram('outputs', outputs)
    outputs
  }
)

model <- MyModel()
model |> compile(optimizer = 'sgd', loss = 'mse')

# Make sure to set `update_freq = N` to log a batch-level summary every N
# batches. In addition to any `tf$summary` contained in `model$call()`,
# metrics added in `model |>compile` will be logged every N batches.
tb_callback <- callback_tensorboard(log_dir = './logs', update_freq = 1)
model |> fit(x_train, y_train, callbacks = list(tb_callback))
```

Custom batch-level summaries in a Functional API Model:

```
my_summary <- function(x) {
  tf$summary$histogram('x', x)
  x
}

inputs <- layer_input(10)
outputs <- inputs |>
  layer_dense(10) |>
  layer_lambda(my_summary)

model <- keras_model(inputs, outputs)
model |> compile(optimizer = 'sgd', loss = 'mse')

# Make sure to set `update_freq = N` to log a batch-level summary every N
# batches. In addition to any `tf.summary` contained in `Model.call`,
# metrics added in `Model.compile` will be logged every N batches.
```

```
tb_callback <- callback_tensorboard(log_dir = './logs', update_freq = 1)
model |> fit(x_train, y_train, callbacks = list(tb_callback))
```

Profiling:

```
# Profile a single batch, e.g. the 5th batch.
tensorboard_callback <- callback_tensorboard(
  log_dir = './logs', profile_batch = 5)
model |> fit(x_train, y_train, epochs = 2,
             callbacks = list(tensorboard_callback))

# Profile a range of batches, e.g. from 10 to 20.
tensorboard_callback <- callback_tensorboard(
  log_dir = './logs', profile_batch = c(10, 20))
model |> fit(x_train, y_train, epochs = 2,
             callbacks = list(tensorboard_callback))
```

See Also

- https://keras.io/api/callbacks/tensorboard#tensorboard-class

Other callbacks:
Callback()
callback_backup_and_restore()
callback_csv_logger()
callback_early_stopping()
callback_lambda()
callback_learning_rate_scheduler()
callback_model_checkpoint()
callback_reduce_lr_on_plateau()
callback_remote_monitor()
callback_swap_ema_weights()
callback_terminate_on_nan()

---

callback_terminate_on_nan
Callback that terminates training when a NaN loss is encountered.

Description

Callback that terminates training when a NaN loss is encountered.

Usage

```
callback_terminate_on_nan()
```

Value

A Callback instance that can be passed to fit.keras.src.models.model.Model().

See Also

- https://keras.io/api/callbacks/terminate_on_nan#terminateonnan-class

Other callbacks:
Callback()
callback_backup_and_restore()
callback_csv_logger()
callback_early_stopping()
callback_lambda()
callback_learning_rate_scheduler()
callback_model_checkpoint()
callback_reduce_lr_on_plateau()
callback_remote_monitor()
callback_swap_ema_weights()
callback_tensorboard()

---

clear_session *Resets all state generated by Keras.*

Description

Keras manages a global state, which it uses to implement the Functional model-building API and to uniquify autogenerated layer names.

If you are creating many models in a loop, this global state will consume an increasing amount of memory over time, and you may want to clear it. Calling clear_session() releases the global state: this helps avoid clutter from old models and layers, especially when memory is limited.

Example 1: calling clear_session() when creating models in a loop

```
for (i in 1:100) {
  # Without `clear_session()`, each iteration of this loop will
  # slightly increase the size of the global state managed by Keras
  model <- keras_model_sequential()
  for (j in 1:10) {
    model <- model |> layer_dense(units = 10)
  }
}

for (i in 1:100) {
  # With `clear_session()` called at the beginning,
  # Keras starts with a blank state at each iteration
  # and memory consumption is constant over time.
  clear_session()
  model <- keras_model_sequential()
  for (j in 1:10) {
    model <- model |> layer_dense(units = 10)
  }
}
```

Example 2: resetting the layer name generation counter

```
layers <- lapply(1:10, \(i) layer_dense(units = 10))

new_layer <- layer_dense(units = 10)
print(new_layer$name)

## [1] "dense_10"


clear_session()
new_layer <- layer_dense(units = 10)
print(new_layer$name)

## [1] "dense"
```

Usage

```
clear_session()
```

Value

`NULL`, invisibly, called for side effects.

See Also

- https://keras.io/api/utils/config_utils#clearsession-function

Other backend:
`config_backend()`
`config_epsilon()`
`config_floatx()`
`config_image_data_format()`
`config_set_epsilon()`
`config_set_floatx()`
`config_set_image_data_format()`

Other utils:
`audio_dataset_from_directory()`
`config_disable_interactive_logging()`
`config_disable_traceback_filtering()`

config_enable_interactive_logging()
config_enable_traceback_filtering()
config_is_interactive_logging_enabled()
config_is_traceback_filtering_enabled()
get_file()
get_source_inputs()
image_array_save()
image_dataset_from_directory()
image_from_array()
image_load()
image_smart_resize()
image_to_array()
layer_feature_space()
normalize()
pack_x_y_sample_weight()
pad_sequences()
set_random_seed()
split_dataset()
text_dataset_from_directory()
timeseries_dataset_from_array()
to_categorical()
unpack_x_y_sample_weight()
zip_lists()

---

clone_model *Clone a model instance.*

Description

Model cloning is similar to calling a model on new inputs, except that it creates new layers (and thus new weights) instead of sharing the weights of the existing layers.

Usage

```
clone_model(model, input_tensors = NULL, clone_function = NULL)
```

Arguments

`model`	Instance of Keras model (could be a functional model or a Sequential model).
`input_tensors`	Optional list of input tensors to build the model upon. If not provided, placeholders will be created.
`clone_function`	Callable to be used to clone each layer in the target model (except `InputLayer` instances). It takes as argument the layer instance to be cloned, and returns the corresponding layer instance to be used in the model copy. If unspecified, this callable defaults to the following serialization/deserialization function: `function(layer) layer$`__class__`$from_config(layer$get_config())`

By passing a custom callable, you can customize your copy of the model, e.g. by wrapping certain layers of interest (you might want to replace all LSTM instances with equivalent `Bidirectional(LSTM(...))` instances, for example).

Value

A new model instance.

`compile.keras.src.models.model.Model`
Configure a model for training.

Description

Configure a model for training.

Usage

```
## S3 method for class 'keras.src.models.model.Model'
compile(
  object,
  optimizer = "rmsprop",
  loss = NULL,
  metrics = NULL,
  ...,
  loss_weights = NULL,
  weighted_metrics = NULL,
  run_eagerly = FALSE,
  steps_per_execution = 1L,
  jit_compile = "auto",
  auto_scale_loss = TRUE
)
```

Arguments

`object`	Keras model object
`optimizer`	String (name of optimizer) or optimizer instance. See `optimizer_*` family.
`loss`	Loss function. May be: • a string (name of builtin loss function), • a custom function, or • a `Loss` instance (returned by the `loss_*` family of functions). A loss function is any callable with the signature `loss = fn(y_true, y_pred)`, where `y_true` are the ground truth values, and `y_pred` are the model's predictions. `y_true` should have shape `(batch_size, d1, .. dN)` (except in the case of sparse loss functions such as sparse categorical crossentropy which expects integer arrays of shape `(batch_size, d1, .. dN-1)`). `y_pred` should

have shape `(batch_size, d1, .. dN)`. The loss function should return a float tensor.

`metrics` List of metrics to be evaluated by the model during training and testing. Each of these can be:

- a string (name of a built-in function),
- a function, optionally with a `"name"` attribute or
- a `Metric()` instance. See the `metric_*` family of functions.

Typically you will use `metrics = c('accuracy')`. A function is any callable with the signature `result = fn(y_true, y_pred)`. To specify different metrics for different outputs of a multi-output model, you could also pass a named list, such as `metrics = list(a = 'accuracy', b = c('accuracy', 'mse'))`. You can also pass a list to specify a metric or a list of metrics for each output, such as `metrics = list(c('accuracy'), c('accuracy', 'mse'))` or `metrics = list('accuracy', c('accuracy', 'mse'))`. When you pass the strings `'accuracy'` or `'acc'`, we convert this to one of `metric_binary_accuracy()`, `metric_categorical_accuracy()`, `metric_sparse_categorical_accuracy()` based on the shapes of the targets and of the model output. A similar conversion is done for the strings `"crossentropy"` and `"ce"` as well. The metrics passed here are evaluated without sample weighting; if you would like sample weighting to apply, you can specify your metrics via the `weighted_metrics` argument instead.

If providing an anonymous R function, you can customize the printed name during training by assigning `attr(<fn>, "name") <- "my_custom_metric_name"`, or by calling `custom_metric("my_custom_metric_name", <fn>)`

`...` Additional arguments passed on to the `compile()` model method.

`loss_weights` Optional list (named or unnamed) specifying scalar coefficients (R numerics) to weight the loss contributions of different model outputs. The loss value that will be minimized by the model will then be the *weighted sum* of all individual losses, weighted by the `loss_weights` coefficients. If an unnamed list, it is expected to have a 1:1 mapping to the model's outputs. If a named list, it is expected to map output names (strings) to scalar coefficients.

`weighted_metrics`
List of metrics to be evaluated and weighted by `sample_weight` or `class_weight` during training and testing.

`run_eagerly` Bool. If `TRUE`, this model's forward pass will never be compiled. It is recommended to leave this as `FALSE` when training (for best performance), and to set it to `TRUE` when debugging.

`steps_per_execution`
Int. The number of batches to run during each a single compiled function call. Running multiple batches inside a single compiled function call can greatly improve performance on TPUs or small models with a large R/Python overhead. At most, one full epoch will be run each execution. If a number larger than the size of the epoch is passed, the execution will be truncated to the size of the epoch. Note that if `steps_per_execution` is set to `N`, `Callback$on_batch_begin` and `Callback$on_batch_end` methods will only be called every `N` batches (i.e. before/after each compiled function execution). Not supported with the PyTorch backend.

`jit_compile`	Bool or `"auto"`. Whether to use XLA compilation when compiling a model. For `jax` and `tensorflow` backends, `jit_compile="auto"` enables XLA compilation if the model supports it, and disabled otherwise. For `torch` backend, `"auto"` will default to eager execution and `jit_compile=True` will run with `torch.compile` with the `"inductor"` backend.
`auto_scale_loss`	Bool. If `TRUE` and the model dtype policy is `"mixed_float16"`, the passed optimizer will be automatically wrapped in a `LossScaleOptimizer`, which will dynamically scale the loss to prevent underflow.

Value

This is called primarily for the side effect of modifying `object` in-place. The first argument `object` is also returned, invisibly, to enable usage with the pipe.

Examples

```
model |> compile(
  optimizer = optimizer_adam(learning_rate = 1e-3),
  loss = loss_binary_crossentropy(),
  metrics = c(metric_binary_accuracy(),
              metric_false_negatives())
)
```

See Also

- https://keras.io/api/models/model_training_apis#compile-method

Other model training:
`evaluate.keras.src.models.model.Model()`
`predict.keras.src.models.model.Model()`
`predict_on_batch()`
`test_on_batch()`
`train_on_batch()`

`config_backend` *Publicly accessible method for determining the current backend.*

Description

Publicly accessible method for determining the current backend.

Usage

```
config_backend()
```

Value

String, the name of the backend Keras is currently using. One of "tensorflow", "torch", or "jax".

Examples

```
config_backend()

## [1] "tensorflow"
```

See Also

use_backend()

Other config backend:
config_epsilon()
config_floatx()
config_image_data_format()
config_set_epsilon()
config_set_floatx()
config_set_image_data_format()

Other backend:
clear_session()
config_epsilon()
config_floatx()
config_image_data_format()
config_set_epsilon()
config_set_floatx()
config_set_image_data_format()

Other config:
config_disable_interactive_logging()
config_disable_traceback_filtering()
config_enable_interactive_logging()
config_enable_traceback_filtering()
config_enable_unsafe_deserialization()
config_epsilon()
config_floatx()
config_image_data_format()
config_is_interactive_logging_enabled()
config_is_traceback_filtering_enabled()
config_set_backend()
config_set_epsilon()
config_set_floatx()
config_set_image_data_format()

config_disable_interactive_logging
Turn off interactive logging.

Description

When interactive logging is disabled, Keras sends logs to `absl.logging`. This is the best option when using Keras in a non-interactive way, such as running a training or inference job on a server.

Usage

```
config_disable_interactive_logging()
```

Value

No return value, called for side effects.

See Also

Other io utils:
config_enable_interactive_logging()
config_is_interactive_logging_enabled()

Other utils:
audio_dataset_from_directory()
clear_session()
config_disable_traceback_filtering()
config_enable_interactive_logging()
config_enable_traceback_filtering()
config_is_interactive_logging_enabled()
config_is_traceback_filtering_enabled()
get_file()
get_source_inputs()
image_array_save()
image_dataset_from_directory()
image_from_array()
image_load()
image_smart_resize()
image_to_array()
layer_feature_space()
normalize()
pack_x_y_sample_weight()
pad_sequences()
set_random_seed()
split_dataset()
text_dataset_from_directory()
timeseries_dataset_from_array()

to_categorical()
unpack_x_y_sample_weight()
zip_lists()

Other config:
config_backend()
config_disable_traceback_filtering()
config_enable_interactive_logging()
config_enable_traceback_filtering()
config_enable_unsafe_deserialization()
config_epsilon()
config_floatx()
config_image_data_format()
config_is_interactive_logging_enabled()
config_is_traceback_filtering_enabled()
config_set_backend()
config_set_epsilon()
config_set_floatx()
config_set_image_data_format()

`config_disable_traceback_filtering`
Turn off traceback filtering.

Description

Raw Keras tracebacks (also known as stack traces) involve many internal frames, which can be challenging to read through, while not being actionable for end users. By default, Keras filters internal frames in most exceptions that it raises, to keep traceback short, readable, and focused on what's actionable for you (your own code).

See also `config_enable_traceback_filtering()` and `config_is_traceback_filtering_enabled()`.

If you have previously disabled traceback filtering via `config_disable_traceback_filtering()`, you can re-enable it via `config_enable_traceback_filtering()`.

Usage

```
config_disable_traceback_filtering()
```

Value

No return value, called for side effects.

See Also

Other traceback utils:
`config_enable_traceback_filtering()`
`config_is_traceback_filtering_enabled()`

Other utils:
`audio_dataset_from_directory()`
`clear_session()`
`config_disable_interactive_logging()`
`config_enable_interactive_logging()`
`config_enable_traceback_filtering()`
`config_is_interactive_logging_enabled()`
`config_is_traceback_filtering_enabled()`
`get_file()`
`get_source_inputs()`
`image_array_save()`
`image_dataset_from_directory()`
`image_from_array()`
`image_load()`
`image_smart_resize()`
`image_to_array()`
`layer_feature_space()`
`normalize()`
`pack_x_y_sample_weight()`
`pad_sequences()`
`set_random_seed()`
`split_dataset()`
`text_dataset_from_directory()`
`timeseries_dataset_from_array()`
`to_categorical()`
`unpack_x_y_sample_weight()`
`zip_lists()`

Other config:
`config_backend()`
`config_disable_interactive_logging()`
`config_enable_interactive_logging()`
`config_enable_traceback_filtering()`
`config_enable_unsafe_deserialization()`
`config_epsilon()`
`config_floatx()`
`config_image_data_format()`
`config_is_interactive_logging_enabled()`
`config_is_traceback_filtering_enabled()`
`config_set_backend()`
`config_set_epsilon()`
`config_set_floatx()`

config_set_image_data_format()

config_dtype_policy *Returns the current default dtype policy object.*

Description

Returns the current default dtype policy object.

Usage

```
config_dtype_policy()
```

Value

A DTypePolicy object.

config_enable_interactive_logging *Turn on interactive logging.*

Description

When interactive logging is enabled, Keras displays logs via stdout. This provides the best experience when using Keras in an interactive environment such as a shell or a notebook.

Usage

```
config_enable_interactive_logging()
```

Value

No return value, called for side effects.

See Also

Other io utils:
config_disable_interactive_logging()
config_is_interactive_logging_enabled()

Other utils:
audio_dataset_from_directory()
clear_session()
config_disable_interactive_logging()
config_disable_traceback_filtering()

config_enable_traceback_filtering()
config_is_interactive_logging_enabled()
config_is_traceback_filtering_enabled()
get_file()
get_source_inputs()
image_array_save()
image_dataset_from_directory()
image_from_array()
image_load()
image_smart_resize()
image_to_array()
layer_feature_space()
normalize()
pack_x_y_sample_weight()
pad_sequences()
set_random_seed()
split_dataset()
text_dataset_from_directory()
timeseries_dataset_from_array()
to_categorical()
unpack_x_y_sample_weight()
zip_lists()

Other config:
config_backend()
config_disable_interactive_logging()
config_disable_traceback_filtering()
config_enable_traceback_filtering()
config_enable_unsafe_deserialization()
config_epsilon()
config_floatx()
config_image_data_format()
config_is_interactive_logging_enabled()
config_is_traceback_filtering_enabled()
config_set_backend()
config_set_epsilon()
config_set_floatx()
config_set_image_data_format()

---

config_enable_traceback_filtering

Turn on traceback filtering.

Description

Raw Keras tracebacks (also known as stack traces) involve many internal frames, which can be challenging to read through, while not being actionable for end users. By default, Keras filters internal frames in most exceptions that it raises, to keep traceback short, readable, and focused on what's actionable for you (your own code).

See also `config_disable_traceback_filtering()` and `config_is_traceback_filtering_enabled()`.

If you have previously disabled traceback filtering via `config_disable_traceback_filtering()`, you can re-enable it via `config_enable_traceback_filtering()`.

Usage

```
config_enable_traceback_filtering()
```

Value

No return value, called for side effects.

See Also

Other traceback utils:
`config_disable_traceback_filtering()`
`config_is_traceback_filtering_enabled()`

Other utils:
`audio_dataset_from_directory()`
`clear_session()`
`config_disable_interactive_logging()`
`config_disable_traceback_filtering()`
`config_enable_interactive_logging()`
`config_is_interactive_logging_enabled()`
`config_is_traceback_filtering_enabled()`
`get_file()`
`get_source_inputs()`
`image_array_save()`
`image_dataset_from_directory()`
`image_from_array()`
`image_load()`
`image_smart_resize()`
`image_to_array()`
`layer_feature_space()`
`normalize()`
`pack_x_y_sample_weight()`
`pad_sequences()`
`set_random_seed()`
`split_dataset()`
`text_dataset_from_directory()`
`timeseries_dataset_from_array()`
`to_categorical()`

unpack_x_y_sample_weight()
zip_lists()

Other config:
config_backend()
config_disable_interactive_logging()
config_disable_traceback_filtering()
config_enable_interactive_logging()
config_enable_unsafe_deserialization()
config_epsilon()
config_floatx()
config_image_data_format()
config_is_interactive_logging_enabled()
config_is_traceback_filtering_enabled()
config_set_backend()
config_set_epsilon()
config_set_floatx()
config_set_image_data_format()

---

config_enable_unsafe_deserialization

Disables safe mode globally, allowing deserialization of lambdas.

Description

Disables safe mode globally, allowing deserialization of lambdas.

Usage

```
config_enable_unsafe_deserialization()
```

Value

No return value, called for side effects.

See Also

Other config:
config_backend()
config_disable_interactive_logging()
config_disable_traceback_filtering()
config_enable_interactive_logging()
config_enable_traceback_filtering()
config_epsilon()
config_floatx()
config_image_data_format()

config_is_interactive_logging_enabled()
config_is_traceback_filtering_enabled()
config_set_backend()
config_set_epsilon()
config_set_floatx()
config_set_image_data_format()

config_epsilon *Return the value of the fuzz factor used in numeric expressions.*

Description

Return the value of the fuzz factor used in numeric expressions.

Usage

```
config_epsilon()
```

Value

A float.

Examples

```
config_epsilon()

## [1] 1e-07
```

See Also

- https://keras.io/api/utils/config_utils#epsilon-function

Other config backend:
config_backend()
config_floatx()
config_image_data_format()
config_set_epsilon()
config_set_floatx()
config_set_image_data_format()

Other backend:
clear_session()
config_backend()
config_floatx()
config_image_data_format()
config_set_epsilon()

config_set_floatx()
config_set_image_data_format()

Other config:
config_backend()
config_disable_interactive_logging()
config_disable_traceback_filtering()
config_enable_interactive_logging()
config_enable_traceback_filtering()
config_enable_unsafe_deserialization()
config_floatx()
config_image_data_format()
config_is_interactive_logging_enabled()
config_is_traceback_filtering_enabled()
config_set_backend()
config_set_epsilon()
config_set_floatx()
config_set_image_data_format()

config_floatx *Return the default float type, as a string.*

Description

E.g. 'bfloat16' 'float16', 'float32', 'float64'.

Usage

```
config_floatx()
```

Value

String, the current default float type.

Examples

```
keras3::config_floatx()
```

```
## [1] "float32"
```

See Also

- https://keras.io/api/utils/config_utils#floatx-function

Other config backend:
config_backend()
config_epsilon()
config_image_data_format()
config_set_epsilon()
config_set_floatx()
config_set_image_data_format()

Other backend:
clear_session()
config_backend()
config_epsilon()
config_image_data_format()
config_set_epsilon()
config_set_floatx()
config_set_image_data_format()

Other config:
config_backend()
config_disable_interactive_logging()
config_disable_traceback_filtering()
config_enable_interactive_logging()
config_enable_traceback_filtering()
config_enable_unsafe_deserialization()
config_epsilon()
config_image_data_format()
config_is_interactive_logging_enabled()
config_is_traceback_filtering_enabled()
config_set_backend()
config_set_epsilon()
config_set_floatx()
config_set_image_data_format()

---

config_image_data_format

Return the default image data format convention.

Description

Return the default image data format convention.

Usage

```
config_image_data_format()
```

Value

A string, either 'channels_first' or 'channels_last'.

Examples

```
config_image_data_format()

## [1] "channels_last"
```

See Also

- https://keras.io/api/utils/config_utils#imagedataformat-function

Other config backend:
config_backend()
config_epsilon()
config_floatx()
config_set_epsilon()
config_set_floatx()
config_set_image_data_format()

Other backend:
clear_session()
config_backend()
config_epsilon()
config_floatx()
config_set_epsilon()
config_set_floatx()
config_set_image_data_format()

Other config:
config_backend()
config_disable_interactive_logging()
config_disable_traceback_filtering()
config_enable_interactive_logging()
config_enable_traceback_filtering()
config_enable_unsafe_deserialization()
config_epsilon()
config_floatx()
config_is_interactive_logging_enabled()
config_is_traceback_filtering_enabled()
config_set_backend()
config_set_epsilon()
config_set_floatx()

config_set_image_data_format()

config_is_interactive_logging_enabled
Check if interactive logging is enabled.

Description

To switch between writing logs to stdout and absl.logging, you may use config_enable_interactive_logging() and config_disable_interactive_logging().

Usage

```
config_is_interactive_logging_enabled()
```

Value

Boolean, TRUE if interactive logging is enabled, and FALSE otherwise.

See Also

Other io utils:
config_disable_interactive_logging()
config_enable_interactive_logging()

Other utils:
audio_dataset_from_directory()
clear_session()
config_disable_interactive_logging()
config_disable_traceback_filtering()
config_enable_interactive_logging()
config_enable_traceback_filtering()
config_is_traceback_filtering_enabled()
get_file()
get_source_inputs()
image_array_save()
image_dataset_from_directory()
image_from_array()
image_load()
image_smart_resize()
image_to_array()
layer_feature_space()
normalize()
pack_x_y_sample_weight()
pad_sequences()
set_random_seed()

`split_dataset()`
`text_dataset_from_directory()`
`timeseries_dataset_from_array()`
`to_categorical()`
`unpack_x_y_sample_weight()`
`zip_lists()`

Other config:
`config_backend()`
`config_disable_interactive_logging()`
`config_disable_traceback_filtering()`
`config_enable_interactive_logging()`
`config_enable_traceback_filtering()`
`config_enable_unsafe_deserialization()`
`config_epsilon()`
`config_floatx()`
`config_image_data_format()`
`config_is_traceback_filtering_enabled()`
`config_set_backend()`
`config_set_epsilon()`
`config_set_floatx()`
`config_set_image_data_format()`

`config_is_traceback_filtering_enabled`

Check if traceback filtering is enabled.

Description

Raw Keras tracebacks (also known as stack traces) involve many internal frames, which can be challenging to read through, while not being actionable for end users. By default, Keras filters internal frames in most exceptions that it raises, to keep traceback short, readable, and focused on what's actionable for you (your own code).

See also `config_enable_traceback_filtering()` and `config_disable_traceback_filtering()`.

If you have previously disabled traceback filtering via `config_disable_traceback_filtering()`, you can re-enable it via `config_enable_traceback_filtering()`.

Usage

```
config_is_traceback_filtering_enabled()
```

Value

Boolean, `TRUE` if traceback filtering is enabled, and `FALSE` otherwise.

See Also

Other traceback utils:
`config_disable_traceback_filtering()`
`config_enable_traceback_filtering()`

Other utils:
`audio_dataset_from_directory()`
`clear_session()`
`config_disable_interactive_logging()`
`config_disable_traceback_filtering()`
`config_enable_interactive_logging()`
`config_enable_traceback_filtering()`
`config_is_interactive_logging_enabled()`
`get_file()`
`get_source_inputs()`
`image_array_save()`
`image_dataset_from_directory()`
`image_from_array()`
`image_load()`
`image_smart_resize()`
`image_to_array()`
`layer_feature_space()`
`normalize()`
`pack_x_y_sample_weight()`
`pad_sequences()`
`set_random_seed()`
`split_dataset()`
`text_dataset_from_directory()`
`timeseries_dataset_from_array()`
`to_categorical()`
`unpack_x_y_sample_weight()`
`zip_lists()`

Other config:
`config_backend()`
`config_disable_interactive_logging()`
`config_disable_traceback_filtering()`
`config_enable_interactive_logging()`
`config_enable_traceback_filtering()`
`config_enable_unsafe_deserialization()`
`config_epsilon()`
`config_floatx()`
`config_image_data_format()`
`config_is_interactive_logging_enabled()`
`config_set_backend()`
`config_set_epsilon()`
`config_set_floatx()`

config_set_image_data_format()

---

config_set_backend *Reload the backend (and the Keras package).*

Description

Reload the backend (and the Keras package).

Usage

```
config_set_backend(backend)
```

Arguments

backend	String

Value

Nothing, this function is called for its side effect.

Examples

```
config_set_backend("jax")
```

WARNING

Using this function is dangerous and should be done carefully. Changing the backend will **NOT** convert the type of any already-instantiated objects. Thus, any layers / tensors / etc. already created will no longer be usable without errors. It is strongly recommended **not** to keep around **any** Keras-originated objects instances created before calling `config_set_backend()`.

This includes any function or class instance that uses any Keras functionality. All such code needs to be re-executed after calling `config_set_backend()`.

See Also

Other config:
config_backend()
config_disable_interactive_logging()
config_disable_traceback_filtering()
config_enable_interactive_logging()
config_enable_traceback_filtering()
config_enable_unsafe_deserialization()
config_epsilon()
config_floatx()
config_image_data_format()
config_is_interactive_logging_enabled()

config_is_traceback_filtering_enabled()
config_set_epsilon()
config_set_floatx()
config_set_image_data_format()

`config_set_dtype_policy` *Sets the default dtype policy globally.*

Description

Sets the default dtype policy globally.

Usage

```
config_set_dtype_policy(policy)
```

Arguments

`policy`	A string or `DTypePolicy` object.

Value

No return value, called for side effects.

Examples

```
config_set_dtype_policy("mixed_float16")
```

`config_set_epsilon` *Set the value of the fuzz factor used in numeric expressions.*

Description

Set the value of the fuzz factor used in numeric expressions.

Usage

```
config_set_epsilon(value)
```

Arguments

`value`	float. New value of epsilon.

Value

No return value, called for side effects.

Examples

```
config_epsilon()

## [1] 1e-07


config_set_epsilon(1e-5)
config_epsilon()

## [1] 1e-05


# Set it back to the default value.
config_set_epsilon(1e-7)
```

See Also

- https://keras.io/api/utils/config_utils#setepsilon-function

Other config backend:
config_backend()
config_epsilon()
config_floatx()
config_image_data_format()
config_set_floatx()
config_set_image_data_format()

Other backend:
clear_session()
config_backend()
config_epsilon()
config_floatx()
config_image_data_format()
config_set_floatx()
config_set_image_data_format()

Other config:
config_backend()
config_disable_interactive_logging()
config_disable_traceback_filtering()
config_enable_interactive_logging()
config_enable_traceback_filtering()
config_enable_unsafe_deserialization()
config_epsilon()

config_floatx()
config_image_data_format()
config_is_interactive_logging_enabled()
config_is_traceback_filtering_enabled()
config_set_backend()
config_set_floatx()
config_set_image_data_format()

---

config_set_floatx — *Set the default float dtype.*

Description

Set the default float dtype.

Usage

```
config_set_floatx(value)
```

Arguments

`value`	String; `'bfloat16'`, `'float16'`, `'float32'`, or `'float64'`.

Value

No return value, called for side effects.

Note

It is not recommended to set this to `"float16"` for training, as this will likely cause numeric stability issues. Instead, mixed precision, which leverages a mix of `float16` and `float32`. It can be configured by calling `keras3::keras$mixed_precision$set_dtype_policy('mixed_float16')`.

Examples

```
config_floatx()

## [1] "float32"


config_set_floatx('float64')
config_floatx()

## [1] "float64"


# Set it back to float32
config_set_floatx('float32')
```

Raises

ValueError: In case of invalid value.

See Also

- https://keras.io/api/utils/config_utils#setfloatx-function

Other config backend:
config_backend()
config_epsilon()
config_floatx()
config_image_data_format()
config_set_epsilon()
config_set_image_data_format()

Other backend:
clear_session()
config_backend()
config_epsilon()
config_floatx()
config_image_data_format()
config_set_epsilon()
config_set_image_data_format()

Other config:
config_backend()
config_disable_interactive_logging()
config_disable_traceback_filtering()
config_enable_interactive_logging()
config_enable_traceback_filtering()
config_enable_unsafe_deserialization()
config_epsilon()
config_floatx()
config_image_data_format()
config_is_interactive_logging_enabled()
config_is_traceback_filtering_enabled()
config_set_backend()
config_set_epsilon()
config_set_image_data_format()

---

config_set_image_data_format

Set the value of the image data format convention.

Description

Set the value of the image data format convention.

Usage

```
config_set_image_data_format(data_format)
```

Arguments

`data_format`	string. 'channels_first' or 'channels_last'.

Value

No return value, called for side effects.

Examples

```
config_image_data_format()

## [1] "channels_last"


# 'channels_last'

keras3::config_set_image_data_format('channels_first')
config_image_data_format()

## [1] "channels_first"


# Set it back to `'channels_last'`
keras3::config_set_image_data_format('channels_last')
```

See Also

- https://keras.io/api/utils/config_utils#setimagedataformat-function

Other config backend:
`config_backend()`
`config_epsilon()`
`config_floatx()`
`config_image_data_format()`
`config_set_epsilon()`
`config_set_floatx()`

Other backend:
`clear_session()`
`config_backend()`
`config_epsilon()`

config_floatx()
config_image_data_format()
config_set_epsilon()
config_set_floatx()

Other config:
config_backend()
config_disable_interactive_logging()
config_disable_traceback_filtering()
config_enable_interactive_logging()
config_enable_traceback_filtering()
config_enable_unsafe_deserialization()
config_epsilon()
config_floatx()
config_image_data_format()
config_is_interactive_logging_enabled()
config_is_traceback_filtering_enabled()
config_set_backend()
config_set_epsilon()
config_set_floatx()

Constraint *Define a custom* Constraint *class*

Description

Base class for weight constraints.

A Constraint() instance works like a stateless function. Users who subclass the Constraint class should override the call() method, which takes a single weight parameter and return a projected version of that parameter (e.g. normalized or clipped). Constraints can be used with various Keras layers via the kernel_constraint or bias_constraint arguments.

Here's a simple example of a non-negative weight constraint:

```
constraint_nonnegative <- Constraint("NonNegative",
  call = function(w) {
    w * op_cast(w >= 0, dtype = w$dtype)
  }
)
weight <- op_convert_to_tensor(c(-1, 1))
constraint_nonnegative()(weight)

## tf.Tensor([-0.  1.], shape=(2), dtype=float32)
```

Usage in a layer:

```
layer_dense(units = 4, kernel_constraint = constraint_nonnegative())

## <Dense name=dense, built=False>
```

Usage

```
Constraint(
  classname,
  call = NULL,
  get_config = NULL,
  ...,
  public = list(),
  private = list(),
  inherit = NULL,
  parent_env = parent.frame()
)
```

Arguments

`classname`	String, the name of the custom class. (Conventionally, CamelCase).
`call`	`\(w)` Applies the constraint to the input weight variable. By default, the inputs weight variable is not modified. Users should override this method to implement their own projection function. Args: • `w`: Input weight variable. Returns: Projected variable (by default, returns unmodified inputs).
`get_config`	`\()` Function that returns a named list of the object config. A constraint config is a named list (JSON-serializable) that can be used to reinstantiate the same object (via `do.call(<constraint_class>, <config>)`).
`..., public`	Additional methods or public members of the custom class.
`private`	Named list of R objects (typically, functions) to include in instance private environments. `private` methods will have all the same symbols in scope as public methods (See section "Symbols in Scope"). Each instance will have it's own `private` environment. Any objects in `private` will be invisible from the Keras framework and the Python runtime.
`inherit`	What the custom class will subclass. By default, the base keras class.
`parent_env`	The R environment that all class methods will have as a grandparent.

Value

A function that returns `Constraint` instances, similar to the builtin constraint functions like `constraint_maxnorm()`.

Symbols in scope

All R function custom methods (public and private) will have the following symbols in scope:

- `self`: The custom class instance.
- `super`: The custom class superclass.
- `private`: An R environment specific to the class instance. Any objects assigned here are invisible to the Keras framework.
- `__class__` and `as.symbol(classname)`: the custom class type object.

See Also

Other constraints:
`constraint_maxnorm()`
`constraint_minmaxnorm()`
`constraint_nonneg()`
`constraint_unitnorm()`

---

constraint_maxnorm *MaxNorm weight constraint.*

Description

Constrains the weights incident to each hidden unit to have a norm less than or equal to a desired value.

Usage

```
constraint_maxnorm(max_value = 2L, axis = 1L)
```

Arguments

`max_value`	the maximum norm value for the incoming weights.
`axis`	integer, axis along which to calculate weight norms. For instance, in a `Dense` layer the weight matrix has shape `(input_dim, output_dim)`, set `axis` to `0` to constrain each weight vector of length `(input_dim,)`. In a `Conv2D` layer with `data_format = "channels_last"`, the weight tensor has shape `(rows, cols, input_depth, output_d` set `axis` to `[0, 1, 2]` to constrain the weights of each filter tensor of size `(rows, cols, input_depth)`.

Value

A `Constraint` instance, a callable that can be passed to layer constructors or used directly by calling it with tensors.

See Also

- https://keras.io/api/layers/constraints#maxnorm-class

Other constraints:
Constraint()
constraint_minmaxnorm()
constraint_nonneg()
constraint_unitnorm()

---

constraint_minmaxnorm *MinMaxNorm weight constraint.*

Description

Constrains the weights incident to each hidden unit to have the norm between a lower bound and an upper bound.

Usage

```
constraint_minmaxnorm(min_value = 0, max_value = 1, rate = 1, axis = 1L)
```

Arguments

min_value	the minimum norm for the incoming weights.
max_value	the maximum norm for the incoming weights.
rate	rate for enforcing the constraint: weights will be rescaled to yield op_clip? (1 - rate) * norm + rate * op_clip(norm, min_value, max_value). Effectively, this means that rate = 1.0 stands for strict enforcement of the constraint, while rate<1.0 means that weights will be rescaled at each step to slowly move towards a value inside the desired interval.
axis	integer, axis along which to calculate weight norms. For instance, in a Dense layer the weight matrix has shape (input_dim, output_dim), set axis to 0 to constrain each weight vector of length (input_dim,). In a Conv2D layer with data_format = "channels_last", the weight tensor has shape (rows, cols, input_depth, output_d set axis to [0, 1, 2] to constrain the weights of each filter tensor of size (rows, cols, input_depth).

Value

A Constraint instance, a callable that can be passed to layer constructors or used directly by calling it with tensors.

See Also

- https://keras.io/api/layers/constraints#minmaxnorm-class

Other constraints:
`Constraint()`
`constraint_maxnorm()`
`constraint_nonneg()`
`constraint_unitnorm()`

constraint_nonneg *Constrains the weights to be non-negative.*

Description

Constrains the weights to be non-negative.

Usage

```
constraint_nonneg()
```

Value

A `Constraint` instance, a callable that can be passed to layer constructors or used directly by calling it with tensors.

See Also

- https://keras.io/api/layers/constraints#nonneg-class

Other constraints:
`Constraint()`
`constraint_maxnorm()`
`constraint_minmaxnorm()`
`constraint_unitnorm()`

constraint_unitnorm *Constrains the weights incident to each hidden unit to have unit norm.*

Description

Constrains the weights incident to each hidden unit to have unit norm.

Usage

```
constraint_unitnorm(axis = 1L)
```

Arguments

`axis`	integer, axis along which to calculate weight norms. For instance, in a `Dense` layer the weight matrix has shape `(input_dim, output_dim)`, set `axis` to `0` to constrain each weight vector of length `(input_dim,)`. In a `Conv2D` layer with `data_format = "channels_last"`, the weight tensor has shape `(rows, cols, input_depth, output_d` set `axis` to `[0, 1, 2]` to constrain the weights of each filter tensor of size `(rows, cols, input_depth)`.

Value

A `Constraint` instance, a callable that can be passed to layer constructors or used directly by calling it with tensors.

See Also

- `https://keras.io/api/layers/constraints#unitnorm-class`

Other constraints:
`Constraint()`
`constraint_maxnorm()`
`constraint_minmaxnorm()`
`constraint_nonneg()`

`count_params` *Count the total number of scalars composing the weights.*

Description

Count the total number of scalars composing the weights.

Usage

```
count_params(object)
```

Arguments

`object`	Layer or model object

Value

An integer count

See Also

Other layer methods:
get_config()
get_weights()
quantize_weights()
reset_state()

---

custom_metric *Custom metric function*

Description

Custom metric function

Usage

```
custom_metric(name, metric_fn)
```

Arguments

name	name used to show training progress output
metric_fn	An R function with signature function(y_true, y_pred) that accepts tensors.

Details

You can provide an arbitrary R function as a custom metric. Note that the y_true and y_pred parameters are tensors, so computations on them should use op_* tensor functions.

Use the custom_metric() function to define a custom metric. Note that a name ('mean_pred') is provided for the custom metric function: this name is used within training progress output.

If you want to save and load a model with custom metrics, you should also call register_keras_serializable(), or specify the metric in the call the load_model(). For example: load_model("my_model.keras", c('mean_pred' = metric_mean_pred)).

Alternatively, you can wrap all of your code in a call to with_custom_object_scope() which will allow you to refer to the metric by name just like you do with built in keras metrics.

Alternative ways of supplying custom metrics:

- custom_metric(): Arbitrary R function.
- metric_mean_wrapper(): Wrap an arbitrary R function in a Metric instance.
- Create a custom Metric() subclass.

Value

A callable function with a __name__ attribute.

See Also

Other metrics:
Metric()
metric_auc()
metric_binary_accuracy()
metric_binary_crossentropy()
metric_binary_focal_crossentropy()
metric_binary_iou()
metric_categorical_accuracy()
metric_categorical_crossentropy()
metric_categorical_focal_crossentropy()
metric_categorical_hinge()
metric_cosine_similarity()
metric_f1_score()
metric_false_negatives()
metric_false_positives()
metric_fbeta_score()
metric_hinge()
metric_huber()
metric_iou()
metric_kl_divergence()
metric_log_cosh()
metric_log_cosh_error()
metric_mean()
metric_mean_absolute_error()
metric_mean_absolute_percentage_error()
metric_mean_iou()
metric_mean_squared_error()
metric_mean_squared_logarithmic_error()
metric_mean_wrapper()
metric_one_hot_iou()
metric_one_hot_mean_iou()
metric_poisson()
metric_precision()
metric_precision_at_recall()
metric_r2_score()
metric_recall()
metric_recall_at_precision()
metric_root_mean_squared_error()
metric_sensitivity_at_specificity()
metric_sparse_categorical_accuracy()
metric_sparse_categorical_crossentropy()
metric_sparse_top_k_categorical_accuracy()
metric_specificity_at_sensitivity()
metric_squared_hinge()
metric_sum()
metric_top_k_categorical_accuracy()
metric_true_negatives()

metric_true_positives()

dataset_boston_housing

Boston housing price regression dataset

Description

Dataset taken from the StatLib library which is maintained at Carnegie Mellon University.

Usage

```
dataset_boston_housing(
  path = "boston_housing.npz",
  test_split = 0.2,
  seed = 113L
)
```

Arguments

path	Path where to cache the dataset locally (relative to ~/.keras/datasets).
test_split	fraction of the data to reserve as test set.
seed	Random seed for shuffling the data before computing the test split.

Value

Lists of training and test data: `train$x, train$y, test$x, test$y`.

Samples contain 13 attributes of houses at different locations around the Boston suburbs in the late 1970s. Targets are the median values of the houses at a location (in k$).

See Also

Other datasets:
dataset_cifar10()
dataset_cifar100()
dataset_fashion_mnist()
dataset_imdb()
dataset_mnist()
dataset_reuters()

`dataset_cifar10` *CIFAR10 small image classification*

Description

Dataset of 50,000 32x32 color training images, labeled over 10 categories, and 10,000 test images.

Usage

```
dataset_cifar10()
```

Value

Lists of training and test data: `train$x, train$y, test$x, test$y`.

The `x` data is an array of RGB image data with shape (num_samples, 3, 32, 32).

The `y` data is an array of category labels (integers in range 0-9) with shape (num_samples).

See Also

Other datasets:
`dataset_boston_housing()`
`dataset_cifar100()`
`dataset_fashion_mnist()`
`dataset_imdb()`
`dataset_mnist()`
`dataset_reuters()`

`dataset_cifar100` *CIFAR100 small image classification*

Description

Dataset of 50,000 32x32 color training images, labeled over 100 categories, and 10,000 test images.

Usage

```
dataset_cifar100(label_mode = c("fine", "coarse"))
```

Arguments

`label_mode`	one of "fine", "coarse".

Value

Lists of training and test data: `train$x, train$y, test$x, test$y`.

The `x` data is an array of RGB image data with shape (num_samples, 3, 32, 32).

The `y` data is an array of category labels with shape (num_samples).

See Also

Other datasets:
`dataset_boston_housing()`
`dataset_cifar10()`
`dataset_fashion_mnist()`
`dataset_imdb()`
`dataset_mnist()`
`dataset_reuters()`

`dataset_fashion_mnist` *Fashion-MNIST database of fashion articles*

Description

Dataset of 60,000 28x28 grayscale images of the 10 fashion article classes, along with a test set of 10,000 images. This dataset can be used as a drop-in replacement for MNIST. The class labels are encoded as integers from 0-9 which correspond to T-shirt/top, Trouser, Pullover, Dress, Coat, Sandal, Shirt,

Usage

```
dataset_fashion_mnist()
```

Details

Dataset of 60,000 28x28 grayscale images of 10 fashion categories, along with a test set of 10,000 images. This dataset can be used as a drop-in replacement for MNIST. The class labels are:

- 0 - T-shirt/top
- 1 - Trouser
- 2 - Pullover
- 3 - Dress
- 4 - Coat
- 5 - Sandal
- 6 - Shirt
- 7 - Sneaker
- 8 - Bag
- 9 - Ankle boot

Value

Lists of training and test data: train$x, train$y, test$x, test$y, where x is an array of grayscale image data with shape (num_samples, 28, 28) and y is an array of article labels (integers in range 0-9) with shape (num_samples).

See Also

Other datasets:
dataset_boston_housing()
dataset_cifar10()
dataset_cifar100()
dataset_imdb()
dataset_mnist()
dataset_reuters()

---

dataset_imdb *IMDB Movie reviews sentiment classification*

Description

Dataset of 25,000 movies reviews from IMDB, labeled by sentiment (positive/negative). Reviews have been preprocessed, and each review is encoded as a sequence of word indexes (integers). For convenience, words are indexed by overall frequency in the dataset, so that for instance the integer "3" encodes the 3rd most frequent word in the data. This allows for quick filtering operations such as: "only consider the top 10,000 most common words, but eliminate the top 20 most common words".

Usage

```
dataset_imdb(
  path = "imdb.npz",
  num_words = NULL,
  skip_top = 0L,
  maxlen = NULL,
  seed = 113L,
  start_char = 1L,
  oov_char = 2L,
  index_from = 3L
)

dataset_imdb_word_index(path = "imdb_word_index.json")
```

Arguments

`path`	Where to cache the data (relative to `~/.keras/dataset`).
`num_words`	Max number of words to include. Words are ranked by how often they occur (in the training set) and only the most frequent words are kept
`skip_top`	Skip the top N most frequently occuring words (which may not be informative).
`maxlen`	sequences longer than this will be filtered out.
`seed`	random seed for sample shuffling.
`start_char`	The start of a sequence will be marked with this character. Set to 1 because 0 is usually the padding character.
`oov_char`	Words that were cut out because of the `num_words` or `skip_top` limit will be replaced with this character.
`index_from`	Index actual words with this index and higher.

Details

As a convention, "0" does not stand for a specific word, but instead is used to encode any unknown word.

Value

Lists of training and test data: `train$x, train$y, test$x, test$y`.

The `x` data includes integer sequences. If the `num_words` argument was specific, the maximum possible index value is `num_words-1`. If the `maxlen` argument was specified, the largest possible sequence length is `maxlen`.

The `y` data includes a set of integer labels (0 or 1).

The `dataset_imdb_word_index()` function returns a list where the names are words and the values are integer.

See Also

Other datasets:
`dataset_boston_housing()`
`dataset_cifar10()`
`dataset_cifar100()`
`dataset_fashion_mnist()`
`dataset_mnist()`
`dataset_reuters()`

`dataset_mnist` *MNIST database of handwritten digits*

Description

Dataset of 60,000 28x28 grayscale images of the 10 digits, along with a test set of 10,000 images.

Usage

```
dataset_mnist(path = "mnist.npz")
```

Arguments

`path`	Path where to cache the dataset locally (relative to ~/.keras/datasets).

Value

Lists of training and test data: `train$x, train$y, test$x, test$y`, where `x` is an array of grayscale image data with shape (num_samples, 28, 28) and `y` is an array of digit labels (integers in range 0-9) with shape (num_samples).

See Also

Other datasets:
`dataset_boston_housing()`
`dataset_cifar10()`
`dataset_cifar100()`
`dataset_fashion_mnist()`
`dataset_imdb()`
`dataset_reuters()`

`dataset_reuters` *Reuters newswire topics classification*

Description

Dataset of 11,228 newswires from Reuters, labeled over 46 topics. As with `dataset_imdb()` , each wire is encoded as a sequence of word indexes (same conventions).

Usage

```
dataset_reuters(
  path = "reuters.npz",
  num_words = NULL,
  skip_top = 0L,
  maxlen = NULL,
  test_split = 0.2,
  seed = 113L,
  start_char = 1L,
  oov_char = 2L,
  index_from = 3L
)

dataset_reuters_word_index(path = "reuters_word_index.pkl")
```

Arguments

path	Where to cache the data (relative to `~/.keras/dataset`).
num_words	Max number of words to include. Words are ranked by how often they occur (in the training set) and only the most frequent words are kept
skip_top	Skip the top N most frequently occuring words (which may not be informative).
maxlen	Truncate sequences after this length.
test_split	Fraction of the dataset to be used as test data.
seed	Random seed for sample shuffling.
start_char	The start of a sequence will be marked with this character. Set to 1 because 0 is usually the padding character.
oov_char	words that were cut out because of the `num_words` or `skip_top` limit will be replaced with this character.
index_from	index actual words with this index and higher.

Value

Lists of training and test data: `train$x, train$y, test$x, test$y` with same format as `dataset_imdb()`. The `dataset_reuters_word_index()` function returns a list where the names are words and the values are integer. e.g. `word_index[["giraffe"]]` might return `1234`.

See Also

Other datasets:
`dataset_boston_housing()`
`dataset_cifar10()`
`dataset_cifar100()`
`dataset_fashion_mnist()`
`dataset_imdb()`
`dataset_mnist()`

deserialize_keras_object

Retrieve the object by deserializing the config dict.

Description

The config dict is a Python dictionary that consists of a set of key-value pairs, and represents a Keras object, such as an `Optimizer`, `Layer`, `Metrics`, etc. The saving and loading library uses the following keys to record information of a Keras object:

- `class_name`: String. This is the name of the class, as exactly defined in the source code, such as "LossesContainer".
- `config`: Named List. Library-defined or user-defined key-value pairs that store the configuration of the object, as obtained by `object$get_config()`.
- `module`: String. The path of the python module. Built-in Keras classes expect to have prefix `keras`.
- `registered_name`: String. The key the class is registered under via `register_keras_serializable(package, name)` API. The key has the format of `'{package}>{name}'`, where `package` and `name` are the arguments passed to `register_keras_serializable()`. If `name` is not provided, it uses the class name. If `registered_name` successfully resolves to a class (that was registered), the `class_name` and `config` values in the config dict will not be used. `registered_name` is only used for non-built-in classes.

For example, the following config list represents the built-in Adam optimizer with the relevant config:

```
config <- list(
  class_name = "Adam",
  config = list(
    amsgrad = FALSE,
    beta_1 = 0.8999999761581421,
    beta_2 = 0.9990000128746033,
    epsilon = 1e-07,
    learning_rate = 0.0010000000474974513,
    name = "Adam"
  ),
  module = "keras.optimizers",
  registered_name = NULL
)
# Returns an `Adam` instance identical to the original one.
deserialize_keras_object(config)

## <keras.src.optimizers.adam.Adam object>
```

If the class does not have an exported Keras namespace, the library tracks it by its `module` and `class_name`. For example:

```
config <- list(
  class_name = "MetricsList",
  config =  list(
    ...
  ),
  module = "keras.trainers.compile_utils",
  registered_name = "MetricsList"
)

# Returns a `MetricsList` instance identical to the original one.
deserialize_keras_object(config)
```

And the following config represents a user-customized `MeanSquaredError` loss:

```
# define a custom object
loss_modified_mse <- Loss(
  "ModifiedMeanSquaredError",
  inherit = loss_mean_squared_error)

# register the custom object
register_keras_serializable(loss_modified_mse)

# confirm object is registered
get_custom_objects()

## $`keras3>ModifiedMeanSquaredError`
## <class '<r-namespace:keras3>.ModifiedMeanSquaredError'>


get_registered_name(loss_modified_mse)

## [1] "keras3>ModifiedMeanSquaredError"


# now custom object instances can be serialized
full_config <- serialize_keras_object(loss_modified_mse())

# the `config` arguments will be passed to loss_modified_mse()
str(full_config)

## List of 4
##  $ module         : chr "<r-namespace:keras3>"
##  $ class_name     : chr "ModifiedMeanSquaredError"
##  $ config         :List of 2
##   ..$ name     : chr "mean_squared_error"
##   ..$ reduction: chr "sum_over_batch_size"
##  $ registered_name: chr "keras3>ModifiedMeanSquaredError"
```

```
# and custom object instances can be deserialized
deserialize_keras_object(full_config)

## <<r-namespace:keras3>.ModifiedMeanSquaredError object>


# Returns the `ModifiedMeanSquaredError` object
```

Usage

```
deserialize_keras_object(config, custom_objects = NULL, safe_mode = TRUE, ...)
```

Arguments

`config`	Named list describing the object.
`custom_objects`	Named list containing a mapping between custom object names the corresponding classes or functions.
`safe_mode`	Boolean, whether to disallow unsafe `lambda` deserialization. When `safe_mode=FALSE`, loading an object has the potential to trigger arbitrary code execution. This argument is only applicable to the Keras v3 model format. Defaults to `TRUE`.
`...`	For forward/backward compatability.

Value

The object described by the `config` dictionary.

See Also

- `https://keras.io/api/models/model_saving_apis/serialization_utils#deserializekerasobject-functi`

Other serialization utilities:
`get_custom_objects()`
`get_registered_name()`
`get_registered_object()`
`register_keras_serializable()`
`serialize_keras_object()`
`with_custom_object_scope()`

`evaluate.keras.src.models.model.Model`
Evaluate a Keras Model

Description

This functions returns the loss value and metrics values for the model in test mode. Computation is done in batches (see the `batch_size` arg.)

Usage

```
## S3 method for class 'keras.src.models.model.Model'
evaluate(
  object,
  x = NULL,
  y = NULL,
  ...,
  batch_size = NULL,
  verbose = getOption("keras.verbose", default = "auto"),
  sample_weight = NULL,
  steps = NULL,
  callbacks = NULL
)
```

Arguments

`object`	Keras model object
`x`	Input data. It could be: • An R array (or array-like), or a list of arrays (in case the model has multiple inputs). • A tensor, or a list of tensors (in case the model has multiple inputs). • A named list mapping input names to the corresponding array/tensors, if the model has named inputs. • A `tf.data.Dataset`. Should return a tuple of either `(inputs, targets)` or `(inputs, targets, sample_weights)`. • A generator returning `(inputs, targets)` or `(inputs, targets, sample_weights)`.
`y`	Target data. Like the input data `x`, it could be either R array(s) or backend-native tensor(s). If `x` is a `tf.data.Dataset` or generator function, `y` should not be specified (since targets will be obtained from the iterator/dataset).
`...`	For forward/backward compatability.
`batch_size`	Integer or `NULL`. Number of samples per batch of computation. If unspecified, `batch_size` will default to `32`. Do not specify the `batch_size` if your data is in the form of a a tf dataset or generator (since they generate batches).
`verbose`	`"auto"`, `0`, `1`, or `2`. Verbosity mode. `0` = silent, `1` = progress bar, `2` = single line. `"auto"` becomes `1` for most cases, `2` if in a knitr render or running on a distributed training server. Note that the progress bar is not particularly useful when logged to a file, so `verbose=2` is recommended when not running interactively (e.g. in a production environment). Defaults to `"auto"`.
`sample_weight`	Optional array of weights for the test samples, used for weighting the loss function. You can either pass a flat (1D) R array with the same length as the input samples (1:1 mapping between weights and samples), or in the case of temporal data, you can pass a 2D array with shape `(samples, sequence_length)`, to apply a different weight to every timestep of every sample. This argument is not supported when `x` is a tfdataset, instead pass sample weights as the third element of `x`.

`steps`	Integer or `NULL`. Total number of steps (batches of samples) before declaring the evaluation round finished. Ignored with the default value of `NULL`. If `x` is a `tf.data.Dataset` and `steps` is `NULL`, evaluation will run until the dataset is exhausted.
`callbacks`	List of `Callback` instances. List of callbacks to apply during evaluation.

Value

Scalar test loss (if the model has a single output and no metrics) or list of scalars (if the model has multiple outputs and/or metrics). The attribute `model$metrics_names` will give you the display labels for the scalar outputs.

See Also

- `https://keras.io/api/models/model_training_apis#evaluate-method`

Other model training:
`compile.keras.src.models.model.Model()`
`predict.keras.src.models.model.Model()`
`predict_on_batch()`
`test_on_batch()`
`train_on_batch()`

`export_savedmodel.keras.src.models.model.Model`
Create a TF SavedModel artifact for inference (e.g. via TF-Serving).

Description

(e.g. via TF-Serving).

Note: This can currently only be used with the TensorFlow or JAX backends.

This method lets you export a model to a lightweight SavedModel artifact that contains the model's forward pass only (its `call()` method) and can be served via e.g. TF-Serving. The forward pass is registered under the name `serve()` (see example below).

The original code of the model (including any custom layers you may have used) is *no longer* necessary to reload the artifact – it is entirely standalone.

Usage

```
## S3 method for class 'keras.src.models.model.Model'
export_savedmodel(object, export_dir_base, ...)
```

Arguments

`object`	A keras model.
`export_dir_base`	string, file path where to save the artifact.
`...`	For forward/backward compatability.

Value

This is called primarily for the side effect of exporting `object`. The first argument, `object` is also returned, invisibly, to enable usage with the pipe.

Examples

```
# Create the artifact
model |> tensorflow::export_savedmodel("path/to/location")

# Later, in a different process / environment...
library(tensorflow)
reloaded_artifact <- tf$saved_model$load("path/to/location")
predictions <- reloaded_artifact$serve(input_data)

# see tfdeploy::serve_savedmodel() for serving a model over a local web api.
```

See Also

Other saving and loading functions:
`layer_tfsm()`
`load_model()`
`load_model_weights()`
`register_keras_serializable()`
`save_model()`
`save_model_config()`
`save_model_weights()`
`with_custom_object_scope()`

`fit.keras.src.models.model.Model` *Train a model for a fixed number of epochs (dataset iterations).*

Description

Train a model for a fixed number of epochs (dataset iterations).

Usage

```
## S3 method for class 'keras.src.models.model.Model'
fit(
  object,
  x = NULL,
  y = NULL,
  ...,
  batch_size = NULL,
  epochs = 1L,
  callbacks = NULL,
  validation_split = 0,
  validation_data = NULL,
  shuffle = TRUE,
  class_weight = NULL,
  sample_weight = NULL,
  initial_epoch = 1L,
  steps_per_epoch = NULL,
  validation_steps = NULL,
  validation_batch_size = NULL,
  validation_freq = 1L,
  verbose = getOption("keras.verbose", default = "auto"),
  view_metrics = getOption("keras.view_metrics", default = "auto")
)
```

Arguments

`object`	Keras model object
`x`	Input data. It could be: • An array (or array-like), or a list of arrays (in case the model has multiple inputs). • A tensor, or a list of tensors (in case the model has multiple inputs). • A named list mapping input names to the corresponding array/tensors, if the model has named inputs. • A `tf.data.Dataset`. Should return a tuple of either `(inputs, targets)` or `(inputs, targets, sample_weights)`. • A generator returning `(inputs, targets)` or `(inputs, targets, sample_weights)`.
`y`	Target data. Like the input data `x`, it could be either array(s) or backend-native tensor(s). If `x` is a TF Dataset or generator, `y` should not be specified (since targets will be obtained from `x`).
`...`	Additional arguments passed on to the model `fit()` method.
`batch_size`	Integer or `NULL`. Number of samples per gradient update. If unspecified, `batch_size` will default to `32`. Do not specify the `batch_size` if your data is in the form of TF Datasets or generators, (since they generate batches).
`epochs`	Integer. Number of epochs to train the model. An epoch is an iteration over the entire `x` and `y` data provided (unless the `steps_per_epoch` flag is set to something other than `NULL`). Note that in conjunction with `initial_epoch`, `epochs` is to be understood as "final epoch". The model is not trained for a number of iterations given by `epochs`, but merely until the epoch of index `epochs` is reached.
`callbacks`	List of `Callback()` instances. List of callbacks to apply during training. See `callback_*`.
`validation_split`	Float between 0 and 1. Fraction of the training data to be used as validation data. The model will set apart this fraction of the training data, will not train on it, and will evaluate the loss and any model metrics on this data at the end of

each epoch. The validation data is selected from the last samples in the x and y data provided, before shuffling. This argument is not supported when x is a TF Dataset or generator. If both validation_data and validation_split are provided, validation_data will override validation_split.

validation_data
: Data on which to evaluate the loss and any model metrics at the end of each epoch. The model will not be trained on this data. Thus, note the fact that the validation loss of data provided using validation_split or validation_data is not affected by regularization layers like noise and dropout. validation_data will override validation_split. It could be:

- A tuple (x_val, y_val) of arrays or tensors.
- A tuple (x_val, y_val, val_sample_weights) of arrays.
- A generator returning (inputs, targets) or (inputs, targets, sample_weights).

shuffle
: Boolean, whether to shuffle the training data before each epoch. This argument is ignored when x is a generator or a TF Dataset.

class_weight
: Optional named list mapping class indices (integers, 0-based) to a weight (float) value, used for weighting the loss function (during training only). This can be useful to tell the model to "pay more attention" to samples from an under-represented class. When class_weight is specified and targets have a rank of 2 or greater, either y must be one-hot encoded, or an explicit final dimension of 1 must be included for sparse class labels.

sample_weight
: Optional array of weights for the training samples, used for weighting the loss function (during training only). You can either pass a flat (1D) array/vector with the same length as the input samples (1:1 mapping between weights and samples), or in the case of temporal data, you can pass a 2D array (matrix) with shape (samples, sequence_length), to apply a different weight to every timestep of every sample. This argument is not supported when x is a TF Dataset or generator, instead provide the sample_weights as the third element of x. Note that sample weighting does not apply to metrics specified via the metrics argument in compile(). To apply sample weighting to your metrics, you can specify them via the weighted_metrics in compile() instead.

initial_epoch
: Integer. Epoch at which to start training (useful for resuming a previous training run).

steps_per_epoch
: Integer or NULL. Total number of steps (batches of samples) before declaring one epoch finished and starting the next epoch. When training with input tensors such as backend-native tensors, the default NULL is equal to the number of samples in your dataset divided by the batch size, or 1 if that cannot be determined. If x is a TF Dataset, and steps_per_epoch is NULL, the epoch will run until the input dataset is exhausted. When passing an infinitely repeating dataset, you must specify the steps_per_epoch argument. If steps_per_epoch = -1 the training will run indefinitely with an infinitely repeating dataset.

validation_steps
: Only relevant if validation_data is provided. Total number of steps (batches of samples) to draw before stopping when performing validation at the end of every epoch. If validation_steps is NULL, validation will run until the

	`validation_data` dataset is exhausted. In the case of an infinitely repeated dataset, it will run into an infinite loop. If `validation_steps` is specified and only part of the dataset will be consumed, the evaluation will start from the beginning of the dataset at each epoch. This ensures that the same validation samples are used every time.
`validation_batch_size`	Integer or `NULL`. Number of samples per validation batch. If unspecified, will default to `batch_size`. Do not specify the `validation_batch_size` if your data is in the form of TF Datasets or generator instances (since they generate batches).
`validation_freq`	Only relevant if validation data is provided. Specifies how many training epochs to run before a new validation run is performed, e.g. `validation_freq=2` runs validation every 2 epochs.
`verbose`	`"auto"`, `0`, `1`, or `2`. Verbosity mode. `0` = silent, `1` = progress bar, `2` = one line per epoch. `"auto"` becomes 1 for most cases, 2 if in a knitr render or running on a distributed training server. Note that the progress bar is not particularly useful when logged to a file, so `verbose=2` is recommended when not running interactively (e.g., in a production environment). Defaults to `"auto"`.
`view_metrics`	View realtime plot of training metrics (by epoch). The default (`"auto"`) will display the plot when running within RStudio, `metrics` were specified during model `compile()`, `epochs > 1` and `verbose > 0`. Set the global `options(keras.view_metrics = )` option to establish a different default.

Details

Unpacking behavior for iterator-like inputs:

A common pattern is to pass an iterator like object such as a `tf.data.Dataset` or a generator to `fit()`, which will in fact yield not only features (`x`) but optionally targets (`y`) and sample weights (`sample_weight`). Keras requires that the output of such iterator-likes be unambiguous. The iterator should return a `tuple()` of length 1, 2, or 3, where the optional second and third elements will be used for `y` and `sample_weight` respectively. Any other type provided will be wrapped in a length-one `tuple()`, effectively treating everything as `x`. When yielding named lists, they should still adhere to the top-level tuple structure, e.g. `tuple(list(x0 = x0, x = x1), y)`. Keras will not attempt to separate features, targets, and weights from the keys of a single dict.

Value

A `keras_training_history` object, which is a named list: `list(params = <params>, metrics = <metrics>")`, with S3 methods `print()`, `plot()`, and `as.data.frame()`. The metrics field is a record of training loss values and metrics values at successive epochs, as well as validation loss values and validation metrics values (if applicable).

See Also

- `https://keras.io/api/models/model_training_apis#fit-method` `freeze_weights` *Freeze and unfreeze weights*

Description

Freeze weights in a model or layer so that they are no longer trainable.

Usage

```
freeze_weights(object, from = NULL, to = NULL, which = NULL)

unfreeze_weights(object, from = NULL, to = NULL, which = NULL)
```

Arguments

`object`	Keras model or layer object
`from`	Layer instance, layer name, or layer index within model
`to`	Layer instance, layer name, or layer index within model
`which`	layer names, integer positions, layers, logical vector (of `length(object$layers)`), or a function returning a logical vector.

Value

The input `object` with frozen weights is returned, invisibly. Note, `object` is modified in place, and the return value is only provided to make usage with the pipe convenient.

Examples

```
# instantiate a VGG16 model
conv_base <- application_vgg16(
  weights = "imagenet",
  include_top = FALSE,
  input_shape = c(150, 150, 3)
)

# freeze it's weights
freeze_weights(conv_base)

# Note the "Trainable" column
conv_base


## Model: "vgg16"
## +-----------------------------+-----------------------+------------+-------+
## | Layer (type)               | Output Shape         |   Param # | Trai. . . |
## +=============================+=======================+============+=======+
## | input_layer (InputLayer)    | (None, 150, 150, 3)   |          0 |   -   |
## +-----------------------------+----------------------+-----------+-------+
## | block1_conv1 (Conv2D)       | (None, 150, 150, 64)  |     1,792 |   N   |
## +-----------------------------+----------------------+-----------+-------+
## | block1_conv2 (Conv2D)       | (None, 150, 150, 64)  |    36,928 |   N   |
## +-----------------------------+----------------------+-----------+-------+
## | block1_pool (MaxPooling2D)  | (None, 75, 75, 64)    |         0 |   -   |
## +-----------------------------+----------------------+-----------+-------+
## | block2_conv1 (Conv2D)       | (None, 75, 75, 128)   |    73,856 |   N   |
## +-----------------------------+----------------------+-----------+-------+
## | block2_conv2 (Conv2D)       | (None, 75, 75, 128)   |   147,584 |   N   |
## +-----------------------------+----------------------+-----------+-------+
## | block2_pool (MaxPooling2D)  | (None, 37, 37, 128)   |         0 |   -   |
## +-----------------------------+----------------------+-----------+-------+
## | block3_conv1 (Conv2D)       | (None, 37, 37, 256)   |   295,168 |   N   |
## +-----------------------------+----------------------+-----------+-------+
## | block3_conv2 (Conv2D)       | (None, 37, 37, 256)   |   590,080 |   N   |
## +-----------------------------+----------------------+-----------+-------+
## | block3_conv3 (Conv2D)       | (None, 37, 37, 256)   |   590,080 |   N   |
## +-----------------------------+----------------------+-----------+-------+
## | block3_pool (MaxPooling2D)  | (None, 18, 18, 256)   |         0 |   -   |
## +-----------------------------+----------------------+-----------+-------+
## | block4_conv1 (Conv2D)       | (None, 18, 18, 512)   | 1,180,160 |   N   |
## +-----------------------------+----------------------+-----------+-------+
## | block4_conv2 (Conv2D)       | (None, 18, 18, 512)   | 2,359,808 |   N   |
## +-----------------------------+----------------------+-----------+-------+
## | block4_conv3 (Conv2D)       | (None, 18, 18, 512)   | 2,359,808 |   N   |
## +-----------------------------+----------------------+-----------+-------+
## | block4_pool (MaxPooling2D)  | (None, 9, 9, 512)     |         0 |   -   |
## +-----------------------------+----------------------+-----------+-------+
## | block5_conv1 (Conv2D)       | (None, 9, 9, 512)     | 2,359,808 |   N   |
## +-----------------------------+----------------------+-----------+-------+
## | block5_conv2 (Conv2D)       | (None, 9, 9, 512)     | 2,359,808 |   N   |
## +-----------------------------+----------------------+-----------+-------+
## | block5_conv3 (Conv2D)       | (None, 9, 9, 512)     | 2,359,808 |   N   |
## +-----------------------------+----------------------+-----------+-------+
## | block5_pool (MaxPooling2D)  | (None, 4, 4, 512)     |         0 |   -   |
## +-----------------------------+----------------------+-----------+-------+
##  Total params: 14,714,688 (56.13 MB)
##  Trainable params: 0 (0.00 B)
##  Non-trainable params: 14,714,688 (56.13 MB)
```

```
# create a composite model that includes the base + more layers
model <- keras_model_sequential(input_batch_shape = shape(conv_base$input)) |>
  conv_base() |>
  layer_flatten() |>
  layer_dense(units = 256, activation = "relu") |>
  layer_dense(units = 1, activation = "sigmoid")
```

```
# compile
model |> compile(
  loss = "binary_crossentropy",
  optimizer = optimizer_rmsprop(learning_rate = 2e-5),
  metrics = c("accuracy")
)

model

## Model: "sequential"
## +-----------------------------+----------------------+-----------+-------+
## | Layer (type)                | Output Shape         |   Param # | Trai… |
## +=============================+======================+===========+=======+
## | vgg16 (Functional)          | (None, 4, 4, 512)    | 14,714,688 |   N   |
## +-----------------------------+----------------------+-----------+-------+
## | flatten (Flatten)           | (None, 8192)         |         0 |   -   |
## +-----------------------------+----------------------+-----------+-------+
## | dense_1 (Dense)             | (None, 256)          | 2,097,408 |   Y   |
## +-----------------------------+----------------------+-----------+-------+
## | dense (Dense)               | (None, 1)            |       257 |   Y   |
## +-----------------------------+----------------------+-----------+-------+
##  Total params: 16,812,353 (64.13 MB)
##  Trainable params: 2,097,665 (8.00 MB)
##  Non-trainable params: 14,714,688 (56.13 MB)


print(model, expand_nested = TRUE)

## Model: "sequential"
## +-----------------------------+----------------------+-----------+-------+
## | Layer (type)                | Output Shape         |   Param # | Trai… |
## +=============================+======================+===========+=======+
## | vgg16 (Functional)          | (None, 4, 4, 512)    | 14,714,688 |   N   |
## +-----------------------------+----------------------+-----------+-------+
## |    > input_layer            | (None, 150, 150, 3)  |         0 |   -   |
## | (InputLayer)                |                      |           |       |
## +-----------------------------+----------------------+-----------+-------+
## |    > block1_conv1 (Conv2D)  | (None, 150, 150, 64) |     1,792 |   N   |
## +-----------------------------+----------------------+-----------+-------+
## |    > block1_conv2 (Conv2D)  | (None, 150, 150, 64) |    36,928 |   N   |
## +-----------------------------+----------------------+-----------+-------+
## |    > block1_pool            | (None, 75, 75, 64)   |         0 |   -   |
## | (MaxPooling2D)              |                      |           |       |
## +-----------------------------+----------------------+-----------+-------+
## |    > block2_conv1 (Conv2D)  | (None, 75, 75, 128)  |    73,856 |   N   |
## +-----------------------------+----------------------+-----------+-------+
## |    > block2_conv2 (Conv2D)  | (None, 75, 75, 128)  |   147,584 |   N   |
## +-----------------------------+----------------------+-----------+-------+
## |    > block2_pool           | (None, 37, 37, 128)   |          0 |   -   |
## | (MaxPooling2D)             |                       |            |       |
## +----------------------------+-----------------------+------------+-------+
## |    > block3_conv1 (Conv2D) | (None, 37, 37, 256)   |    295,168 |   N   |
## +----------------------------+-----------------------+------------+-------+
## |    > block3_conv2 (Conv2D) | (None, 37, 37, 256)   |    590,080 |   N   |
## +----------------------------+-----------------------+------------+-------+
## |    > block3_conv3 (Conv2D) | (None, 37, 37, 256)   |    590,080 |   N   |
## +----------------------------+-----------------------+------------+-------+
## |    > block3_pool           | (None, 18, 18, 256)   |          0 |   -   |
## | (MaxPooling2D)             |                       |            |       |
## +----------------------------+-----------------------+------------+-------+
## |    > block4_conv1 (Conv2D) | (None, 18, 18, 512)   |  1,180,160 |   N   |
## +----------------------------+-----------------------+------------+-------+
## |    > block4_conv2 (Conv2D) | (None, 18, 18, 512)   |  2,359,808 |   N   |
## +----------------------------+-----------------------+------------+-------+
## |    > block4_conv3 (Conv2D) | (None, 18, 18, 512)   |  2,359,808 |   N   |
## +----------------------------+-----------------------+------------+-------+
## |    > block4_pool           | (None, 9, 9, 512)     |          0 |   -   |
## | (MaxPooling2D)             |                       |            |       |
## +----------------------------+-----------------------+------------+-------+
## |    > block5_conv1 (Conv2D) | (None, 9, 9, 512)     |  2,359,808 |   N   |
## +----------------------------+-----------------------+------------+-------+
## |    > block5_conv2 (Conv2D) | (None, 9, 9, 512)     |  2,359,808 |   N   |
## +----------------------------+-----------------------+------------+-------+
## |    > block5_conv3 (Conv2D) | (None, 9, 9, 512)     |  2,359,808 |   N   |
## +----------------------------+-----------------------+------------+-------+
## |    > block5_pool           | (None, 4, 4, 512)     |          0 |   -   |
## | (MaxPooling2D)             |                       |            |       |
## +----------------------------+-----------------------+------------+-------+
## | flatten (Flatten)          | (None, 8192)          |          0 |   -   |
## +----------------------------+-----------------------+------------+-------+
## | dense_1 (Dense)            | (None, 256)           |  2,097,408 |   Y   |
## +----------------------------+-----------------------+------------+-------+
## | dense (Dense)              | (None, 1)             |        257 |   Y   |
## +----------------------------+-----------------------+------------+-------+
##  Total params: 16,812,353 (64.13 MB)
##  Trainable params: 2,097,665 (8.00 MB)
##  Non-trainable params: 14,714,688 (56.13 MB)



# unfreeze weights from "block5_conv1" on
unfreeze_weights(conv_base, from = "block5_conv1")

# compile again since we froze or unfroze weights
model |> compile(
  loss = "binary_crossentropy",
  optimizer = optimizer_rmsprop(learning_rate = 2e-5),
  metrics = c("accuracy")
)

conv_base

## Model: "vgg16"
## +-----------------------------+----------------------+-----------+-------+
## | Layer (type)               | Output Shape         |    Param # | Trai. . . |
## +=============================+======================+===========+=======+
## | input_layer (InputLayer)    | (None, 150, 150, 3)   |          0 |   -   |
## +-----------------------------+----------------------+-----------+-------+
## | block1_conv1 (Conv2D)       | (None, 150, 150, 64)  |      1,792 |   N   |
## +-----------------------------+----------------------+-----------+-------+
## | block1_conv2 (Conv2D)       | (None, 150, 150, 64)  |     36,928 |   N   |
## +-----------------------------+----------------------+-----------+-------+
## | block1_pool (MaxPooling2D)  | (None, 75, 75, 64)    |          0 |   -   |
## +-----------------------------+----------------------+-----------+-------+
## | block2_conv1 (Conv2D)       | (None, 75, 75, 128)   |     73,856 |   N   |
## +-----------------------------+----------------------+-----------+-------+
## | block2_conv2 (Conv2D)       | (None, 75, 75, 128)   |    147,584 |   N   |
## +-----------------------------+----------------------+-----------+-------+
## | block2_pool (MaxPooling2D)  | (None, 37, 37, 128)   |          0 |   -   |
## +-----------------------------+----------------------+-----------+-------+
## | block3_conv1 (Conv2D)       | (None, 37, 37, 256)   |    295,168 |   N   |
## +-----------------------------+----------------------+-----------+-------+
## | block3_conv2 (Conv2D)       | (None, 37, 37, 256)   |    590,080 |   N   |
## +-----------------------------+----------------------+-----------+-------+
## | block3_conv3 (Conv2D)       | (None, 37, 37, 256)   |    590,080 |   N   |
## +-----------------------------+----------------------+-----------+-------+
## | block3_pool (MaxPooling2D)  | (None, 18, 18, 256)   |          0 |   -   |
## +-----------------------------+----------------------+-----------+-------+
## | block4_conv1 (Conv2D)       | (None, 18, 18, 512)   |  1,180,160 |   N   |
## +-----------------------------+----------------------+-----------+-------+
## | block4_conv2 (Conv2D)       | (None, 18, 18, 512)   |  2,359,808 |   N   |
## +-----------------------------+----------------------+-----------+-------+
## | block4_conv3 (Conv2D)       | (None, 18, 18, 512)   |  2,359,808 |   N   |
## +-----------------------------+----------------------+-----------+-------+
## | block4_pool (MaxPooling2D)  | (None, 9, 9, 512)     |          0 |   -   |
## +-----------------------------+----------------------+-----------+-------+
## | block5_conv1 (Conv2D)       | (None, 9, 9, 512)     |  2,359,808 |   Y   |
## +-----------------------------+----------------------+-----------+-------+
## | block5_conv2 (Conv2D)       | (None, 9, 9, 512)     |  2,359,808 |   Y   |
## +-----------------------------+----------------------+-----------+-------+
## | block5_conv3 (Conv2D)       | (None, 9, 9, 512)     |  2,359,808 |   Y   |
## +-----------------------------+----------------------+-----------+-------+
## | block5_pool (MaxPooling2D)  | (None, 4, 4, 512)     |          0 |   -   |
## +-----------------------------+----------------------+-----------+-------+
##  Total params: 14,714,688 (56.13 MB)
##  Trainable params: 7,079,424 (27.01 MB)
##  Non-trainable params: 7,635,264 (29.13 MB)
```

```
print(model, expand_nested = TRUE)
```

```
## Model: "sequential"
## +-----------------------------+----------------------+-----------+-------+
## | Layer (type)                | Output Shape         |   Param # | Trai... |
## +=============================+======================+===========+=======+
## | vgg16 (Functional)          | (None, 4, 4, 512)    | 14,714,688 |   Y   |
## +-----------------------------+----------------------+-----------+-------+
## |    > input_layer            | (None, 150, 150, 3)  |         0 |   -   |
## | (InputLayer)                |                      |           |       |
## +-----------------------------+----------------------+-----------+-------+
## |    > block1_conv1 (Conv2D)  | (None, 150, 150, 64) |     1,792 |   N   |
## +-----------------------------+----------------------+-----------+-------+
## |    > block1_conv2 (Conv2D)  | (None, 150, 150, 64) |    36,928 |   N   |
## +-----------------------------+----------------------+-----------+-------+
## |    > block1_pool            | (None, 75, 75, 64)   |         0 |   -   |
## | (MaxPooling2D)              |                      |           |       |
## +-----------------------------+----------------------+-----------+-------+
## |    > block2_conv1 (Conv2D)  | (None, 75, 75, 128)  |    73,856 |   N   |
## +-----------------------------+----------------------+-----------+-------+
## |    > block2_conv2 (Conv2D)  | (None, 75, 75, 128)  |   147,584 |   N   |
## +-----------------------------+----------------------+-----------+-------+
## |    > block2_pool            | (None, 37, 37, 128)  |         0 |   -   |
## | (MaxPooling2D)              |                      |           |       |
## +-----------------------------+----------------------+-----------+-------+
## |    > block3_conv1 (Conv2D)  | (None, 37, 37, 256)  |   295,168 |   N   |
## +-----------------------------+----------------------+-----------+-------+
## |    > block3_conv2 (Conv2D)  | (None, 37, 37, 256)  |   590,080 |   N   |
## +-----------------------------+----------------------+-----------+-------+
## |    > block3_conv3 (Conv2D)  | (None, 37, 37, 256)  |   590,080 |   N   |
## +-----------------------------+----------------------+-----------+-------+
## |    > block3_pool            | (None, 18, 18, 256)  |         0 |   -   |
## | (MaxPooling2D)              |                      |           |       |
## +-----------------------------+----------------------+-----------+-------+
## |    > block4_conv1 (Conv2D)  | (None, 18, 18, 512)  | 1,180,160 |   N   |
## +-----------------------------+----------------------+-----------+-------+
## |    > block4_conv2 (Conv2D)  | (None, 18, 18, 512)  | 2,359,808 |   N   |
## +-----------------------------+----------------------+-----------+-------+
## |    > block4_conv3 (Conv2D)  | (None, 18, 18, 512)  | 2,359,808 |   N   |
## +-----------------------------+----------------------+-----------+-------+
## |    > block4_pool            | (None, 9, 9, 512)    |         0 |   -   |
## | (MaxPooling2D)              |                      |           |       |
## +-----------------------------+----------------------+-----------+-------+
## |    > block5_conv1 (Conv2D)  | (None, 9, 9, 512)    | 2,359,808 |   Y   |
## +-----------------------------+----------------------+-----------+-------+
## |    > block5_conv2 (Conv2D)  | (None, 9, 9, 512)    | 2,359,808 |   Y   |
## +-----------------------------+----------------------+-----------+-------+
## |    > block5_conv3 (Conv2D)  | (None, 9, 9, 512)    | 2,359,808 |   Y   |
## +-----------------------------+----------------------+-----------+-------+
## |    > block5_pool            | (None, 4, 4, 512)    |         0 |   -   |
## | (MaxPooling2D)              |                      |           |       |
## +-----------------------------+----------------------+-----------+-------+
## | flatten (Flatten)           | (None, 8192)         |         0 |   -   |
## +-----------------------------+----------------------+-----------+-------+
## | dense_1 (Dense)             | (None, 256)          | 2,097,408 |   Y   |
## +-----------------------------+----------------------+-----------+-------+
## | dense (Dense)               | (None, 1)            |       257 |   Y   |
## +-----------------------------+----------------------+-----------+-------+
##  Total params: 16,812,353 (64.13 MB)
##  Trainable params: 9,177,089 (35.01 MB)
##  Non-trainable params: 7,635,264 (29.13 MB)
```

```
# freeze only the last 5 layers
freeze_weights(conv_base, from = -5)
conv_base
```

```
## Model: "vgg16"
## +-----------------------------+----------------------+-----------+-------+
## | Layer (type)               | Output Shape         |   Param # | Trai... |
## +=============================+======================+===========+=======+
## | input_layer (InputLayer)    | (None, 150, 150, 3)  |         0 |   -   |
## +-----------------------------+----------------------+-----------+-------+
## | block1_conv1 (Conv2D)       | (None, 150, 150, 64) |     1,792 |   Y   |
## +-----------------------------+----------------------+-----------+-------+
## | block1_conv2 (Conv2D)       | (None, 150, 150, 64) |    36,928 |   Y   |
## +-----------------------------+----------------------+-----------+-------+
## | block1_pool (MaxPooling2D)  | (None, 75, 75, 64)   |         0 |   -   |
## +-----------------------------+----------------------+-----------+-------+
## | block2_conv1 (Conv2D)       | (None, 75, 75, 128)  |    73,856 |   Y   |
## +-----------------------------+----------------------+-----------+-------+
## | block2_conv2 (Conv2D)       | (None, 75, 75, 128)  |   147,584 |   Y   |
## +-----------------------------+----------------------+-----------+-------+
## | block2_pool (MaxPooling2D)  | (None, 37, 37, 128)  |         0 |   -   |
## +-----------------------------+----------------------+-----------+-------+
## | block3_conv1 (Conv2D)       | (None, 37, 37, 256)  |   295,168 |   Y   |
## +-----------------------------+----------------------+-----------+-------+
## | block3_conv2 (Conv2D)       | (None, 37, 37, 256)  |   590,080 |   Y   |
## +-----------------------------+----------------------+-----------+-------+
## | block3_conv3 (Conv2D)       | (None, 37, 37, 256)  |   590,080 |   Y   |
## +-----------------------------+----------------------+-----------+-------+
## | block3_pool (MaxPooling2D)  | (None, 18, 18, 256)  |         0 |   -   |
## +-----------------------------+----------------------+-----------+-------+
## | block4_conv1 (Conv2D)       | (None, 18, 18, 512)  | 1,180,160 |   Y   |
## +-----------------------------+----------------------+-----------+-------+
## | block4_conv2 (Conv2D)       | (None, 18, 18, 512)   |  2,359,808 |   Y   |
## +-----------------------------+-----------------------+-----------+-------+
## | block4_conv3 (Conv2D)       | (None, 18, 18, 512)   |  2,359,808 |   Y   |
## +-----------------------------+-----------------------+-----------+-------+
## | block4_pool (MaxPooling2D)  | (None, 9, 9, 512)     |         0 |   -   |
## +-----------------------------+-----------------------+-----------+-------+
## | block5_conv1 (Conv2D)       | (None, 9, 9, 512)     |  2,359,808 |   N   |
## +-----------------------------+-----------------------+-----------+-------+
## | block5_conv2 (Conv2D)       | (None, 9, 9, 512)     |  2,359,808 |   N   |
## +-----------------------------+-----------------------+-----------+-------+
## | block5_conv3 (Conv2D)       | (None, 9, 9, 512)     |  2,359,808 |   N   |
## +-----------------------------+-----------------------+-----------+-------+
## | block5_pool (MaxPooling2D)  | (None, 4, 4, 512)     |         0 |   -   |
## +-----------------------------+-----------------------+-----------+-------+
##  Total params: 14,714,688 (56.13 MB)
##  Trainable params: 7,635,264 (29.13 MB)
##  Non-trainable params: 7,079,424 (27.01 MB)


# freeze only the last 5 layers, a different way
unfreeze_weights(conv_base, to = -6)
conv_base

## Model: "vgg16"
## +-----------------------------+-----------------------+-----------+-------+
## | Layer (type)               | Output Shape          |    Param # | Trai. . . |
## +=============================+=======================+===========+=======+
## | input_layer (InputLayer)    | (None, 150, 150, 3)   |         0 |   -   |
## +-----------------------------+-----------------------+-----------+-------+
## | block1_conv1 (Conv2D)       | (None, 150, 150, 64)  |      1,792 |   Y   |
## +-----------------------------+-----------------------+-----------+-------+
## | block1_conv2 (Conv2D)       | (None, 150, 150, 64)  |     36,928 |   Y   |
## +-----------------------------+-----------------------+-----------+-------+
## | block1_pool (MaxPooling2D)  | (None, 75, 75, 64)    |         0 |   -   |
## +-----------------------------+-----------------------+-----------+-------+
## | block2_conv1 (Conv2D)       | (None, 75, 75, 128)   |     73,856 |   Y   |
## +-----------------------------+-----------------------+-----------+-------+
## | block2_conv2 (Conv2D)       | (None, 75, 75, 128)   |    147,584 |   Y   |
## +-----------------------------+-----------------------+-----------+-------+
## | block2_pool (MaxPooling2D)  | (None, 37, 37, 128)   |         0 |   -   |
## +-----------------------------+-----------------------+-----------+-------+
## | block3_conv1 (Conv2D)       | (None, 37, 37, 256)   |    295,168 |   Y   |
## +-----------------------------+-----------------------+-----------+-------+
## | block3_conv2 (Conv2D)       | (None, 37, 37, 256)   |    590,080 |   Y   |
## +-----------------------------+-----------------------+-----------+-------+
## | block3_conv3 (Conv2D)       | (None, 37, 37, 256)   |    590,080 |   Y   |
## +-----------------------------+-----------------------+-----------+-------+
## | block3_pool (MaxPooling2D)  | (None, 18, 18, 256)   |         0 |   -   |
## +-----------------------------+-----------------------+-----------+-------+
## | block4_conv1 (Conv2D)       | (None, 18, 18, 512)   |  1,180,160 |   Y   |
## +-----------------------------+-----------------------+------------+-------+
## | block4_conv2 (Conv2D)       | (None, 18, 18, 512)   |  2,359,808 |   Y   |
## +-----------------------------+-----------------------+------------+-------+
## | block4_conv3 (Conv2D)       | (None, 18, 18, 512)   |  2,359,808 |   Y   |
## +-----------------------------+-----------------------+------------+-------+
## | block4_pool (MaxPooling2D)  | (None, 9, 9, 512)     |          0 |   -   |
## +-----------------------------+-----------------------+------------+-------+
## | block5_conv1 (Conv2D)       | (None, 9, 9, 512)     |  2,359,808 |   N   |
## +-----------------------------+-----------------------+------------+-------+
## | block5_conv2 (Conv2D)       | (None, 9, 9, 512)     |  2,359,808 |   N   |
## +-----------------------------+-----------------------+------------+-------+
## | block5_conv3 (Conv2D)       | (None, 9, 9, 512)     |  2,359,808 |   N   |
## +-----------------------------+-----------------------+------------+-------+
## | block5_pool (MaxPooling2D)  | (None, 4, 4, 512)     |          0 |   -   |
## +-----------------------------+-----------------------+------------+-------+
##  Total params: 14,714,688 (56.13 MB)
##  Trainable params: 7,635,264 (29.13 MB)
##  Non-trainable params: 7,079,424 (27.01 MB)


# Freeze only layers of a certain type, e.g, BatchNorm layers
batch_norm_layer_class_name <- class(layer_batch_normalization())[1]
is_batch_norm_layer <- function(x) inherits(x, batch_norm_layer_class_name)

model <- application_efficientnet_b0()
freeze_weights(model, which = is_batch_norm_layer)
# print(model)

# equivalent to:
for(layer in model$layers) {
  if(is_batch_norm_layer(layer))
    layer$trainable <- FALSE
  else
    layer$trainable <- TRUE
}
```

Note

The `from` and `to` layer arguments are both inclusive.

When applied to a model, the freeze or unfreeze is a global operation over all layers in the model (i.e. layers not within the specified range will be set to the opposite value, e.g. unfrozen for a call to freeze).

Models must be compiled again after weights are frozen or unfrozen.

get_config — *Layer/Model configuration*

Description

A layer config is an object returned from `get_config()` that contains the configuration of a layer or model. The same layer or model can be reinstantiated later (without its trained weights) from this configuration using `from_config()`. The config does not include connectivity information, nor the class name (those are handled externally).

Usage

```
get_config(object)

from_config(config, custom_objects = NULL)
```

Arguments

`object`	Layer or model object
`config`	Object with layer or model configuration
`custom_objects`	list of custom objects needed to instantiate the layer, e.g., custom layers defined by `new_layer_class()` or similar.

Value

`get_config()` returns an object with the configuration, `from_config()` returns a re-instantiation of the object.

Note

Objects returned from `get_config()` are not serializable via RDS. If you want to save and restore a model across sessions, you can use `save_model_config()` (for model configuration only, not weights) or `save_model()` to save the model configuration and weights to the filesystem.

See Also

Other model functions:
`get_layer()`
`keras_model()`
`keras_model_sequential()`
`pop_layer()`
`summary.keras.src.models.model.Model()`

Other layer methods:
`count_params()`
`get_weights()`
`quantize_weights()`

reset_state()

---

get_custom_objects *Get/set the currently registered custom objects.*

Description

Custom objects set using `custom_object_scope()` are not added to the global list of custom objects, and will not appear in the returned list.

Usage

```
get_custom_objects()

set_custom_objects(objects = named_list(), clear = TRUE)
```

Arguments

`objects`	A named list of custom objects, as returned by `get_custom_objects()` and `set_custom_objects()`.
`clear`	bool, whether to clear the custom object registry before populating it with `objects`.

Value

An R named list mapping registered names to registered objects. `set_custom_objects()` returns the registry values before updating, invisibly.

Examples

```
get_custom_objects()
```

You can use `set_custom_objects()` to restore a previous registry state.

```
# within a function, if you want to temporarily modify the registry,
function() {
  orig_objects <- set_custom_objects(clear = TRUE)
  on.exit(set_custom_objects(orig_objects))

  ## temporarily modify the global registry
  # register_keras_serializable(....)
  # ....  <do work>
  # on.exit(), the previous registry state is restored.
}
```

Note

`register_keras_serializable()` is preferred over `set_custom_objects()` for registering new objects.

See Also

Other serialization utilities:
deserialize_keras_object()
get_registered_name()
get_registered_object()
register_keras_serializable()
serialize_keras_object()
with_custom_object_scope()

---

get_file *Downloads a file from a URL if it not already in the cache.*

Description

By default the file at the url origin is downloaded to the cache_dir ~/.keras, placed in the cache_subdir datasets, and given the filename fname. The final location of a file example.txt would therefore be ~/.keras/datasets/example.txt. Files in .tar, .tar.gz, .tar.bz, and .zip formats can also be extracted.

Passing a hash will verify the file after download. The command line programs shasum and sha256sum can compute the hash.

Usage

```
get_file(
  fname = NULL,
  origin = NULL,
  ...,
  file_hash = NULL,
  cache_subdir = "datasets",
  hash_algorithm = "auto",
  extract = FALSE,
  archive_format = "auto",
  cache_dir = NULL,
  force_download = FALSE
)
```

Arguments

fname	Name of the file. If an absolute path, e.g. "/path/to/file.txt" is specified, the file will be saved at that location. If NULL, the name of the file at origin will be used.
origin	Original URL of the file.
...	For forward/backward compatability.
file_hash	The expected hash string of the file after download. The sha256 and md5 hash algorithms are both supported.

`cache_subdir`	Subdirectory under the Keras cache dir where the file is saved. If an absolute path, e.g. `"/path/to/folder"` is specified, the file will be saved at that location.
`hash_algorithm`	Select the hash algorithm to verify the file. options are `"md5'`, `"sha256'`, and `"auto'`. The default 'auto' detects the hash algorithm in use.
`extract`	`TRUE` tries extracting the file as an Archive, like tar or zip.
`archive_format`	Archive format to try for extracting the file. Options are `"auto'`, `"tar'`, `"zip'`, and `NULL`. `"tar"` includes tar, tar.gz, and tar.bz files. The default `"auto"` corresponds to `c("tar", "zip")`. `NULL` or an empty list will return no matches found.
`cache_dir`	Location to store cached files, when `NULL` it defaults to `Sys.getenv("KERAS_HOME", "~/.keras/")`.
`force_download`	If `TRUE`, the file will always be re-downloaded regardless of the cache state.

Value

Path to the downloaded file.

** Warning on malicious downloads **

Downloading something from the Internet carries a risk. NEVER download a file/archive if you do not trust the source. We recommend that you specify the `file_hash` argument (if the hash of the source file is known) to make sure that the file you are getting is the one you expect.

Examples

```
path_to_downloaded_file <- get_file(
   origin = "https://storage.googleapis.com/download.tensorflow.org/example_images/flower_photos.tgz
     extract = TRUE
)
```

See Also

- https://keras.io/api/utils/python_utils#getfile-function

Other utils:
`audio_dataset_from_directory()`
`clear_session()`
`config_disable_interactive_logging()`
`config_disable_traceback_filtering()`
`config_enable_interactive_logging()`
`config_enable_traceback_filtering()`
`config_is_interactive_logging_enabled()`
`config_is_traceback_filtering_enabled()`
`get_source_inputs()`
`image_array_save()`
`image_dataset_from_directory()`
`image_from_array()`
`image_load()`
`image_smart_resize()`
`image_to_array()`

`layer_feature_space()`
`normalize()`
`pack_x_y_sample_weight()`
`pad_sequences()`
`set_random_seed()`
`split_dataset()`
`text_dataset_from_directory()`
`timeseries_dataset_from_array()`
`to_categorical()`
`unpack_x_y_sample_weight()`
`zip_lists()`

`get_layer` *Retrieves a layer based on either its name (unique) or index.*

Description

Indices are based on order of horizontal graph traversal (bottom-up) and are 1-based. If `name` and `index` are both provided, `index` will take precedence.

Usage

```
get_layer(object, name = NULL, index = NULL)
```

Arguments

`object`	Keras model object
`name`	String, name of layer.
`index`	Integer, index of layer (1-based). Also valid are negative values, which count from the end of model.

Value

A layer instance.

See Also

Other model functions:
`get_config()`
`keras_model()`
`keras_model_sequential()`
`pop_layer()`
`summary.keras.src.models.model.Model()`

`get_registered_name` *Returns the name registered to an object within the Keras framework.*

Description

This function is part of the Keras serialization and deserialization framework. It maps objects to the string names associated with those objects for serialization/deserialization.

Usage

```
get_registered_name(obj)
```

Arguments

`obj`	The object to look up.

Value

The name associated with the object, or the default name if the object is not registered.

See Also

Other serialization utilities:
`deserialize_keras_object()`
`get_custom_objects()`
`get_registered_object()`
`register_keras_serializable()`
`serialize_keras_object()`
`with_custom_object_scope()`

`get_registered_object` *Returns the class associated with* `name` *if it is registered with Keras.*

Description

This function is part of the Keras serialization and deserialization framework. It maps strings to the objects associated with them for serialization/deserialization.

Usage

```
get_registered_object(name, custom_objects = NULL, module_objects = NULL)
```

Arguments

`name`	The name to look up.
`custom_objects`	A named list of custom objects to look the name up in. Generally, custom_objects is provided by the user.
`module_objects`	A named list of custom objects to look the name up in. Generally, `module_objects` is provided by midlevel library implementers.

Value

An instantiable class associated with `name`, or `NULL` if no such class exists.

Examples

```
from_config <- function(cls, config, custom_objects = NULL) {
  if ('my_custom_object_name' \%in\% names(config)) {
    config$hidden_cls <- get_registered_object(
      config$my_custom_object_name,
      custom_objects = custom_objects)
  }
}
```

See Also

Other serialization utilities:
`deserialize_keras_object()`
`get_custom_objects()`
`get_registered_name()`
`register_keras_serializable()`
`serialize_keras_object()`
`with_custom_object_scope()`

`get_source_inputs` *Returns the list of input tensors necessary to compute* `tensor`.

Description

Output will always be a list of tensors (potentially with 1 element).

Usage

```
get_source_inputs(tensor)
```

Arguments

`tensor`	The tensor to start from.

Value

List of input tensors.

Example

```
input <- keras_input(c(3))
output <- input |> layer_dense(4) |> op_multiply(5)
reticulate::py_id(get_source_inputs(output)[[1]]) ==
reticulate::py_id(input)
```

```
## [1] TRUE
```

See Also

Other utils:
audio_dataset_from_directory()
clear_session()
config_disable_interactive_logging()
config_disable_traceback_filtering()
config_enable_interactive_logging()
config_enable_traceback_filtering()
config_is_interactive_logging_enabled()
config_is_traceback_filtering_enabled()
get_file()
image_array_save()
image_dataset_from_directory()
image_from_array()
image_load()
image_smart_resize()
image_to_array()
layer_feature_space()
normalize()
pack_x_y_sample_weight()
pad_sequences()
set_random_seed()
split_dataset()
text_dataset_from_directory()
timeseries_dataset_from_array()
to_categorical()
unpack_x_y_sample_weight()
zip_lists()

get_weights *Layer/Model weights as R arrays*

Description

Layer/Model weights as R arrays

Usage

```
get_weights(object, trainable = NA)

set_weights(object, weights)
```

Arguments

`object`	Layer or model object
`trainable`	if `NA` (the default), all weights are returned. If `TRUE`, only weights of trainable variables are returned. If `FALSE`, only weights of non-trainable variables are returned.
`weights`	Weights as R array

Value

A list of R arrays.

Note

You can access the Layer/Model as `KerasVariables` (which are also backend-native tensors like `tf.Variable`) at `object$weights`, `object$trainable_weights`, or `object$non_trainable_weights`

See Also

Other layer methods:
`count_params()`
`get_config()`
`quantize_weights()`
`reset_state()`

image_array_save *Saves an image stored as an array to a path or file object.*

Description

Saves an image stored as an array to a path or file object.

Usage

```
image_array_save(
  x,
  path,
  data_format = NULL,
  file_format = NULL,
  scale = TRUE,
  ...
)
```

Arguments

x	An array.
path	Path or file object.
data_format	Image data format, either "channels_first" or "channels_last".
file_format	Optional file format override. If omitted, the format to use is determined from the filename extension. If a file object was used instead of a filename, this parameter should always be used.
scale	Whether to rescale image values to be within [0, 255].
...	Additional keyword arguments passed to PIL.Image.save().

Value

Called primarily for side effects. The input x is returned, invisibly, to enable usage with the pipe.

See Also

- https://keras.io/api/data_loading/image#saveimg-function

Other image utils:
image_from_array()
image_load()
image_smart_resize()
image_to_array()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()

op_image_pad()
op_image_resize()

Other utils:
audio_dataset_from_directory()
clear_session()
config_disable_interactive_logging()
config_disable_traceback_filtering()
config_enable_interactive_logging()
config_enable_traceback_filtering()
config_is_interactive_logging_enabled()
config_is_traceback_filtering_enabled()
get_file()
get_source_inputs()
image_dataset_from_directory()
image_from_array()
image_load()
image_smart_resize()
image_to_array()
layer_feature_space()
normalize()
pack_x_y_sample_weight()
pad_sequences()
set_random_seed()
split_dataset()
text_dataset_from_directory()
timeseries_dataset_from_array()
to_categorical()
unpack_x_y_sample_weight()
zip_lists()

---

image_dataset_from_directory

Generates a `tf.data.Dataset` *from image files in a directory.*

Description

If your directory structure is:

```
main_directory/
...class_a/
......a_image_1.jpg
......a_image_2.jpg
...class_b/
......b_image_1.jpg
......b_image_2.jpg
```

Then calling `image_dataset_from_directory(main_directory, labels = 'inferred')` will return a `tf.data.Dataset` that yields batches of images from the subdirectories `class_a` and `class_b`, together with labels 0 and 1 (0 corresponding to `class_a` and 1 corresponding to `class_b`).

Supported image formats: `.jpeg`, `.jpg`, `.png`, `.bmp`, `.gif`. Animated gifs are truncated to the first frame.

Usage

```
image_dataset_from_directory(
  directory,
  labels = "inferred",
  label_mode = "int",
  class_names = NULL,
  color_mode = "rgb",
  batch_size = 32L,
  image_size = c(256L, 256L),
  shuffle = TRUE,
  seed = NULL,
  validation_split = NULL,
  subset = NULL,
  interpolation = "bilinear",
  follow_links = FALSE,
  crop_to_aspect_ratio = FALSE,
  pad_to_aspect_ratio = FALSE,
  data_format = NULL,
  verbose = TRUE
)
```

Arguments

`directory`	Directory where the data is located. If `labels` is `"inferred"`, it should contain subdirectories, each containing images for a class. Otherwise, the directory structure is ignored.
`labels`	Either `"inferred"` (labels are generated from the directory structure), `NULL` (no labels), or a list/tuple of integer labels of the same size as the number of image files found in the directory. Labels should be sorted according to the alphanumeric order of the image file paths (obtained via `os.walk(directory)` in Python).
`label_mode`	String describing the encoding of `labels`. Options are: • `"int"`: means that the labels are encoded as integers (e.g. for `sparse_categorical_crossentropy` loss). • `"categorical"` means that the labels are encoded as a categorical vector (e.g. for `categorical_crossentropy` loss). • `"binary"` means that the labels (there can be only 2) are encoded as `float32` scalars with values 0 or 1 (e.g. for `binary_crossentropy`). • `NULL` (no labels).

`class_names`	Only valid if `labels` is `"inferred"`. This is the explicit list of class names (must match names of subdirectories). Used to control the order of the classes (otherwise alphanumerical order is used).
`color_mode`	One of `"grayscale"`, `"rgb"`, `"rgba"`. Defaults to `"rgb"`. Whether the images will be converted to have 1, 3, or 4 channels.
`batch_size`	Size of the batches of data. Defaults to 32. If `NULL`, the data will not be batched (the dataset will yield individual samples).
`image_size`	Size to resize images to after they are read from disk, specified as `(height, width)`. Defaults to `(256, 256)`. Since the pipeline processes batches of images that must all have the same size, this must be provided.
`shuffle`	Whether to shuffle the data. Defaults to `TRUE`. If set to `FALSE`, sorts the data in alphanumeric order.
`seed`	Optional random seed for shuffling and transformations.
`validation_split`	Optional float between 0 and 1, fraction of data to reserve for validation.
`subset`	Subset of the data to return. One of `"training"`, `"validation"`, or `"both"`. Only used if `validation_split` is set. When `subset = "both"`, the utility returns a tuple of two datasets (the training and validation datasets respectively).
`interpolation`	String, the interpolation method used when resizing images. Defaults to `"bilinear"`. Supports `"bilinear"`, `"nearest"`, `"bicubic"`, `"area"`, `"lanczos3"`, `"lanczos5"`, `"gaussian"`, `"mitchellcubic"`.
`follow_links`	Whether to visit subdirectories pointed to by symlinks. Defaults to `FALSE`.
`crop_to_aspect_ratio`	If `TRUE`, resize the images without aspect ratio distortion. When the original aspect ratio differs from the target aspect ratio, the output image will be cropped so as to return the largest possible window in the image (of size `image_size`) that matches the target aspect ratio. By default (`crop_to_aspect_ratio = FALSE`), aspect ratio may not be preserved.
`pad_to_aspect_ratio`	If `TRUE`, resize the images without aspect ratio distortion. When the original aspect ratio differs from the target aspect ratio, the output image will be padded so as to return the largest possible window in the image (of size `image_size`) that matches the target aspect ratio. By default (`pad_to_aspect_ratio=FALSE`), aspect ratio may not be preserved.
`data_format`	If `NULL` uses `config_image_data_format()` otherwise either `'channel_last'` or `'channel_first'`.
`verbose`	Whether to display number information on classes and number of files found. Defaults to `TRUE`.

Value

A `tf.data.Dataset` object.

- If `label_mode` is `NULL`, it yields `float32` tensors of shape `(batch_size, image_size[1], image_size[2], num_cha` encoding images (see below for rules regarding `num_channels`).

- Otherwise, it yields a tuple `(images, labels)`, where `images` has shape `(batch_size, image_size[1], image_size`
  and `labels` follows the format described below.

Rules regarding labels format:

- if `label_mode` is `"int"`, the labels are an `int32` tensor of shape `(batch_size,)`.
- if `label_mode` is `"binary"`, the labels are a `float32` tensor of 1s and 0s of shape `(batch_size, 1)`.
- if `label_mode` is `"categorical"`, the labels are a `float32` tensor of shape `(batch_size, num_classes)`, representing a one-hot encoding of the class index.

Rules regarding number of channels in the yielded images:

- if `color_mode` is `"grayscale"`, there's 1 channel in the image tensors.
- if `color_mode` is `"rgb"`, there are 3 channels in the image tensors.
- if `color_mode` is `"rgba"`, there are 4 channels in the image tensors.

See Also

- https://keras.io/api/data_loading/image#imagedatasetfromdirectory-function

Other dataset utils:
`audio_dataset_from_directory()`
`split_dataset()`
`text_dataset_from_directory()`
`timeseries_dataset_from_array()`

Other utils:
`audio_dataset_from_directory()`
`clear_session()`
`config_disable_interactive_logging()`
`config_disable_traceback_filtering()`
`config_enable_interactive_logging()`
`config_enable_traceback_filtering()`
`config_is_interactive_logging_enabled()`
`config_is_traceback_filtering_enabled()`
`get_file()`
`get_source_inputs()`
`image_array_save()`
`image_from_array()`
`image_load()`
`image_smart_resize()`
`image_to_array()`
`layer_feature_space()`
`normalize()`
`pack_x_y_sample_weight()`
`pad_sequences()`
`set_random_seed()`
`split_dataset()`
`text_dataset_from_directory()`

timeseries_dataset_from_array()
to_categorical()
unpack_x_y_sample_weight()
zip_lists()

Other preprocessing:
image_smart_resize()
text_dataset_from_directory()
timeseries_dataset_from_array()

---

image_from_array *Converts a 3D array to a PIL Image instance.*

Description

Converts a 3D array to a PIL Image instance.

Usage

```
image_from_array(x, data_format = NULL, scale = TRUE, dtype = NULL)
```

Arguments

x	Input data, in any form that can be converted to an array.
data_format	Image data format, can be either "channels_first" or "channels_last". Defaults to NULL, in which case the global setting config_image_data_format() is used (unless you changed it, it defaults to "channels_last").
scale	Whether to rescale the image such that minimum and maximum values are 0 and 255 respectively. Defaults to TRUE.
dtype	Dtype to use. NULL means the global setting config_floatx() is used (unless you changed it, it defaults to "float32"). Defaults to NULL.

Value

A PIL Image instance.

Example

```
img <- array(runif(30000), dim = c(100, 100, 3))
pil_img <- image_from_array(img)
pil_img

## <PIL.Image.Image image mode=RGB size=100x100>
```

See Also

Other image utils:
image_array_save()
image_load()
image_smart_resize()
image_to_array()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()

Other utils:
audio_dataset_from_directory()
clear_session()
config_disable_interactive_logging()
config_disable_traceback_filtering()
config_enable_interactive_logging()
config_enable_traceback_filtering()
config_is_interactive_logging_enabled()
config_is_traceback_filtering_enabled()
get_file()
get_source_inputs()
image_array_save()
image_dataset_from_directory()
image_load()
image_smart_resize()
image_to_array()
layer_feature_space()
normalize()
pack_x_y_sample_weight()
pad_sequences()
set_random_seed()
split_dataset()
text_dataset_from_directory()
timeseries_dataset_from_array()
to_categorical()
unpack_x_y_sample_weight()
zip_lists()

---

image_load *Loads an image into PIL format.*

Description

Loads an image into PIL format.

Usage

```
image_load(
  path,
  color_mode = "rgb",
  target_size = NULL,
  interpolation = "nearest",
  keep_aspect_ratio = FALSE
)
```

Arguments

`path`	Path to image file.
`color_mode`	One of `"grayscale"`, `"rgb"`, `"rgba"`. Default: `"rgb"`. The desired image format.
`target_size`	Either `NULL` (default to original size) or tuple of ints `(img_height, img_width)`.
`interpolation`	Interpolation method used to resample the image if the target size is different from that of the loaded image. Supported methods are `"nearest"`, `"bilinear"`, and `"bicubic"`. If PIL version 1.1.3 or newer is installed, `"lanczos"` is also supported. If PIL version 3.4.0 or newer is installed, `"box"` and `"hamming"` are also supported. By default, `"nearest"` is used.
`keep_aspect_ratio`	Boolean, whether to resize images to a target size without aspect ratio distortion. The image is cropped in the center with target aspect ratio before resizing.

Value

A PIL Image instance.

Example

```
image_path <- get_file(origin = "https://www.r-project.org/logo/Rlogo.png")
(image <- image_load(image_path))

## <PIL.Image.Image image mode=RGB size=724x561>


input_arr <- image_to_array(image)
str(input_arr)

##  num [1:561, 1:724, 1:3] 0 0 0 0 0 0 0 0 0 0 ...


input_arr %<>% array_reshape(dim = c(1, dim(input_arr))) # Convert single image to a batch.

model |> predict(input_arr)
```

See Also

- https://keras.io/api/data_loading/image#loadimg-function

Other image utils:
image_array_save()
image_from_array()
image_smart_resize()
image_to_array()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()

Other utils:
audio_dataset_from_directory()
clear_session()
config_disable_interactive_logging()
config_disable_traceback_filtering()
config_enable_interactive_logging()
config_enable_traceback_filtering()
config_is_interactive_logging_enabled()
config_is_traceback_filtering_enabled()
get_file()
get_source_inputs()
image_array_save()
image_dataset_from_directory()
image_from_array()
image_smart_resize()
image_to_array()
layer_feature_space()
normalize()
pack_x_y_sample_weight()
pad_sequences()
set_random_seed()
split_dataset()
text_dataset_from_directory()
timeseries_dataset_from_array()
to_categorical()
unpack_x_y_sample_weight()
zip_lists()

---

image_smart_resize *Resize images to a target size without aspect ratio distortion.*

Description

Image datasets typically yield images that have each a different size. However, these images need to be batched before they can be processed by Keras layers. To be batched, images need to share the same height and width.

You could simply do, in TF (or JAX equivalent):

```
size <- c(200, 200)
ds <- ds$map(\(img) tf$image$resize(img, size))
```

However, if you do this, you distort the aspect ratio of your images, since in general they do not all have the same aspect ratio as `size`. This is fine in many cases, but not always (e.g. for image generation models this can be a problem).

Note that passing the argument `preserve_aspect_ratio = TRUE` to `tf$image$resize()` will preserve the aspect ratio, but at the cost of no longer respecting the provided target size.

This calls for:

```
size <- c(200, 200)
ds <- ds$map(\(img) image_smart_resize(img, size))
```

Your output images will actually be `(200, 200)`, and will not be distorted. Instead, the parts of the image that do not fit within the target size get cropped out.

The resizing process is:

1. Take the largest centered crop of the image that has the same aspect ratio as the target size. For instance, if `size = c(200, 200)` and the input image has size `(340, 500)`, we take a crop of `(340, 340)` centered along the width.
2. Resize the cropped image to the target size. In the example above, we resize the `(340, 340)` crop to `(200, 200)`.

Usage

```
image_smart_resize(
  x,
  size,
  interpolation = "bilinear",
  data_format = "channels_last",
  backend_module = NULL
)
```

Arguments

`x`	Input image or batch of images (as a tensor or array). Must be in format `(height, width, channels)` or `(batch_size, height, width, channels)`.
`size`	Tuple of `(height, width)` integer. Target size.
`interpolation`	String, interpolation to use for resizing. Defaults to `'bilinear'`. Supports `bilinear`, `nearest`, `bicubic`, `lanczos3`, `lanczos5`.
`data_format`	`"channels_last"` or `"channels_first"`.
`backend_module`	Backend module to use (if different from the default backend).

Value

Array with shape `(size[1], size[2], channels)`. If the input image was an array, the output is an array, and if it was a backend-native tensor, the output is a backend-native tensor.

See Also

Other image utils:
`image_array_save()`
`image_from_array()`
`image_load()`
`image_to_array()`
`op_image_affine_transform()`
`op_image_crop()`
`op_image_extract_patches()`
`op_image_map_coordinates()`
`op_image_pad()`
`op_image_resize()`

Other utils:
`audio_dataset_from_directory()`
`clear_session()`
`config_disable_interactive_logging()`
`config_disable_traceback_filtering()`
`config_enable_interactive_logging()`
`config_enable_traceback_filtering()`
`config_is_interactive_logging_enabled()`
`config_is_traceback_filtering_enabled()`
`get_file()`
`get_source_inputs()`
`image_array_save()`
`image_dataset_from_directory()`
`image_from_array()`
`image_load()`
`image_to_array()`
`layer_feature_space()`
`normalize()`
`pack_x_y_sample_weight()`
`pad_sequences()`
`set_random_seed()`
`split_dataset()`
`text_dataset_from_directory()`
`timeseries_dataset_from_array()`
`to_categorical()`
`unpack_x_y_sample_weight()`
`zip_lists()`

Other preprocessing:
`image_dataset_from_directory()`

text_dataset_from_directory()
timeseries_dataset_from_array()

`image_to_array` *Converts a PIL Image instance to a matrix.*

Description

Converts a PIL Image instance to a matrix.

Usage

```
image_to_array(img, data_format = NULL, dtype = NULL)
```

Arguments

`img`	Input PIL Image instance.
`data_format`	Image data format, can be either `"channels_first"` or `"channels_last"`. Defaults to `NULL`, in which case the global setting `config_image_data_format()` is used (unless you changed it, it defaults to `"channels_last"`).
`dtype`	Dtype to use. `NULL` means the global setting `config_floatx()` is used (unless you changed it, it defaults to `"float32"`).

Value

A 3D array.

Example

```
image_path <- get_file(origin = "https://www.r-project.org/logo/Rlogo.png")
(img <- image_load(image_path))
```

```
## <PIL.Image.Image image mode=RGB size=724x561>
```

```
array <- image_to_array(img)
str(array)
```

```
##  num [1:561, 1:724, 1:3] 0 0 0 0 0 0 0 0 0 0 ...
```

See Also

- https://keras.io/api/data_loading/image#imgtoarray-function

Other image utils:
image_array_save()
image_from_array()
image_load()
image_smart_resize()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()

Other utils:
audio_dataset_from_directory()
clear_session()
config_disable_interactive_logging()
config_disable_traceback_filtering()
config_enable_interactive_logging()
config_enable_traceback_filtering()
config_is_interactive_logging_enabled()
config_is_traceback_filtering_enabled()
get_file()
get_source_inputs()
image_array_save()
image_dataset_from_directory()
image_from_array()
image_load()
image_smart_resize()
layer_feature_space()
normalize()
pack_x_y_sample_weight()
pad_sequences()
set_random_seed()
split_dataset()
text_dataset_from_directory()
timeseries_dataset_from_array()
to_categorical()
unpack_x_y_sample_weight()
zip_lists()

---

initializer_constant *Initializer that generates tensors with constant values.*

Description

Only scalar values are allowed. The constant value provided must be convertible to the dtype requested when calling the initializer.

Usage

```
initializer_constant(value = 0)
```

Arguments

`value`	A numeric scalar.

Value

An `Initializer` instance that can be passed to layer or variable constructors, or called directly with a shape to return a Tensor.

Examples

```
# Standalone usage:
initializer <- initializer_constant(10)
values <- initializer(shape = c(2, 2))

# Usage in a Keras layer:
initializer <- initializer_constant(10)
layer <- layer_dense(units = 3, kernel_initializer = initializer)
```

See Also

- `https://keras.io/api/layers/initializers#constant-class`

Other constant initializers:
`initializer_identity()`
`initializer_ones()`
`initializer_zeros()`

Other initializers:
`initializer_glorot_normal()`
`initializer_glorot_uniform()`
`initializer_he_normal()`
`initializer_he_uniform()`
`initializer_identity()`
`initializer_lecun_normal()`
`initializer_lecun_uniform()`
`initializer_ones()`
`initializer_orthogonal()`
`initializer_random_normal()`
`initializer_random_uniform()`
`initializer_truncated_normal()`

initializer_variance_scaling()
initializer_zeros()

`initializer_glorot_normal`

The Glorot normal initializer, also called Xavier normal initializer.

Description

Draws samples from a truncated normal distribution centered on 0 with `stddev = sqrt(2 / (fan_in + fan_out))` where `fan_in` is the number of input units in the weight tensor and `fan_out` is the number of output units in the weight tensor.

Usage

```
initializer_glorot_normal(seed = NULL)
```

Arguments

seed	An integer or instance of `random_seed_generator()`. Used to make the behavior of the initializer deterministic. Note that an initializer seeded with an integer or `NULL` (unseeded) will produce the same random values across multiple calls. To get different random values across multiple calls, use as seed an instance of `random_seed_generator()`.

Value

An `Initializer` instance that can be passed to layer or variable constructors, or called directly with a shape to return a Tensor.

Examples

```
# Standalone usage:
initializer <- initializer_glorot_normal()
values <- initializer(shape = c(2, 2))

# Usage in a Keras layer:
initializer <- initializer_glorot_normal()
layer <- layer_dense(units = 3, kernel_initializer = initializer)
```

Reference

- Glorot et al., 2010

See Also

- https://keras.io/api/layers/initializers#glorotnormal-class

Other random initializers:
initializer_glorot_uniform()
initializer_he_normal()
initializer_he_uniform()
initializer_lecun_normal()
initializer_lecun_uniform()
initializer_orthogonal()
initializer_random_normal()
initializer_random_uniform()
initializer_truncated_normal()
initializer_variance_scaling()

Other initializers:
initializer_constant()
initializer_glorot_uniform()
initializer_he_normal()
initializer_he_uniform()
initializer_identity()
initializer_lecun_normal()
initializer_lecun_uniform()
initializer_ones()
initializer_orthogonal()
initializer_random_normal()
initializer_random_uniform()
initializer_truncated_normal()
initializer_variance_scaling()
initializer_zeros()

`initializer_glorot_uniform` *The Glorot uniform initializer, also called Xavier uniform initializer.*

Description

Draws samples from a uniform distribution within `[-limit, limit]`, where `limit` = `sqrt(6 / (fan_in + fan_out))` (`fan_in` is the number of input units in the weight tensor and `fan_out` is the number of output units).

Usage

```
initializer_glorot_uniform(seed = NULL)
```

Arguments

`seed`	An integer or instance of `random_seed_generator()`. Used to make the behavior of the initializer deterministic. Note that an initializer seeded with an integer or `NULL` (unseeded) will produce the same random values across multiple calls. To get different random values across multiple calls, use as seed an instance of `random_seed_generator()`.

Value

An `Initializer` instance that can be passed to layer or variable constructors, or called directly with a shape to return a Tensor.

Examples

```
# Standalone usage:
initializer <- initializer_glorot_uniform()
values <- initializer(shape = c(2, 2))

# Usage in a Keras layer:
initializer <- initializer_glorot_uniform()
layer <- layer_dense(units = 3, kernel_initializer = initializer)
```

Reference

- Glorot et al., 2010

See Also

- `https://keras.io/api/layers/initializers#glorotuniform-class`

Other random initializers:
`initializer_glorot_normal()`
`initializer_he_normal()`
`initializer_he_uniform()`
`initializer_lecun_normal()`
`initializer_lecun_uniform()`
`initializer_orthogonal()`
`initializer_random_normal()`
`initializer_random_uniform()`
`initializer_truncated_normal()`
`initializer_variance_scaling()`

Other initializers:
`initializer_constant()`
`initializer_glorot_normal()`
`initializer_he_normal()`
`initializer_he_uniform()`
`initializer_identity()`
`initializer_lecun_normal()`

initializer_lecun_uniform()
initializer_ones()
initializer_orthogonal()
initializer_random_normal()
initializer_random_uniform()
initializer_truncated_normal()
initializer_variance_scaling()
initializer_zeros()

initializer_he_normal *He normal initializer.*

Description

It draws samples from a truncated normal distribution centered on 0 with `stddev = sqrt(2 / fan_in)` where `fan_in` is the number of input units in the weight tensor.

Usage

```
initializer_he_normal(seed = NULL)
```

Arguments

seed	An integer or instance of `random_seed_generator()`. Used to make the behavior of the initializer deterministic. Note that an initializer seeded with an integer or `NULL` (unseeded) will produce the same random values across multiple calls. To get different random values across multiple calls, use as seed an instance of `random_seed_generator()`.

Value

An `Initializer` instance that can be passed to layer or variable constructors, or called directly with a shape to return a Tensor.

Examples

```
# Standalone usage:
initializer <- initializer_he_normal()
values <- initializer(shape = c(2, 2))

# Usage in a Keras layer:
initializer <- initializer_he_normal()
layer <- layer_dense(units = 3, kernel_initializer = initializer)
```

Reference

- He et al., 2015

See Also

- https://keras.io/api/layers/initializers#henormal-class

Other random initializers:
initializer_glorot_normal()
initializer_glorot_uniform()
initializer_he_uniform()
initializer_lecun_normal()
initializer_lecun_uniform()
initializer_orthogonal()
initializer_random_normal()
initializer_random_uniform()
initializer_truncated_normal()
initializer_variance_scaling()

Other initializers:
initializer_constant()
initializer_glorot_normal()
initializer_glorot_uniform()
initializer_he_uniform()
initializer_identity()
initializer_lecun_normal()
initializer_lecun_uniform()
initializer_ones()
initializer_orthogonal()
initializer_random_normal()
initializer_random_uniform()
initializer_truncated_normal()
initializer_variance_scaling()
initializer_zeros()

---

initializer_he_uniform *He uniform variance scaling initializer.*

Description

Draws samples from a uniform distribution within `[-limit, limit]`, where `limit = sqrt(6 / fan_in)` (`fan_in` is the number of input units in the weight tensor).

Usage

```
initializer_he_uniform(seed = NULL)
```

Arguments

`seed`	A integer or instance of `random_seed_generator()`. Used to make the behavior of the initializer deterministic. Note that an initializer seeded with an integer or `NULL` (unseeded) will produce the same random values across multiple calls. To get different random values across multiple calls, use as seed an instance of `random_seed_generator()`.

Value

An `Initializer` instance that can be passed to layer or variable constructors, or called directly with a shape to return a Tensor.

Examples

```
# Standalone usage:
initializer <- initializer_he_uniform()
values <- initializer(shape = c(2, 2))

# Usage in a Keras layer:
initializer <- initializer_he_uniform()
layer <- layer_dense(units = 3, kernel_initializer = initializer)
```

Reference

- He et al., 2015

See Also

- `https://keras.io/api/layers/initializers#heuniform-class`

Other random initializers:
`initializer_glorot_normal()`
`initializer_glorot_uniform()`
`initializer_he_normal()`
`initializer_lecun_normal()`
`initializer_lecun_uniform()`
`initializer_orthogonal()`
`initializer_random_normal()`
`initializer_random_uniform()`
`initializer_truncated_normal()`
`initializer_variance_scaling()`

Other initializers:
`initializer_constant()`
`initializer_glorot_normal()`
`initializer_glorot_uniform()`
`initializer_he_normal()`
`initializer_identity()`
`initializer_lecun_normal()`

initializer_lecun_uniform()
initializer_ones()
initializer_orthogonal()
initializer_random_normal()
initializer_random_uniform()
initializer_truncated_normal()
initializer_variance_scaling()
initializer_zeros()

---

initializer_identity *Initializer that generates the identity matrix.*

Description

Only usable for generating 2D matrices.

Usage

```
initializer_identity(gain = 1)
```

Arguments

gain	Multiplicative factor to apply to the identity matrix.

Value

An `Initializer` instance that can be passed to layer or variable constructors, or called directly with a shape to return a Tensor.

Examples

```
# Standalone usage:
initializer <- initializer_identity()
values <- initializer(shape = c(2, 2))

# Usage in a Keras layer:
initializer <- initializer_identity()
layer <- layer_dense(units = 3, kernel_initializer = initializer)
```

See Also

Other constant initializers:
initializer_constant()
initializer_ones()
initializer_zeros()

Other initializers:
`initializer_constant()`
`initializer_glorot_normal()`
`initializer_glorot_uniform()`
`initializer_he_normal()`
`initializer_he_uniform()`
`initializer_lecun_normal()`
`initializer_lecun_uniform()`
`initializer_ones()`
`initializer_orthogonal()`
`initializer_random_normal()`
`initializer_random_uniform()`
`initializer_truncated_normal()`
`initializer_variance_scaling()`
`initializer_zeros()`

`initializer_lecun_normal` *Lecun normal initializer.*

Description

Initializers allow you to pre-specify an initialization strategy, encoded in the Initializer object, without knowing the shape and dtype of the variable being initialized.

Draws samples from a truncated normal distribution centered on 0 with `stddev = sqrt(1 / fan_in)` where `fan_in` is the number of input units in the weight tensor.

Usage

```
initializer_lecun_normal(seed = NULL)
```

Arguments

seed	An integer or instance of `random_seed_generator()`. Used to make the behavior of the initializer deterministic. Note that an initializer seeded with an integer or `NULL` (unseeded) will produce the same random values across multiple calls. To get different random values across multiple calls, use as seed an instance of `random_seed_generator()`.

Value

An `Initializer` instance that can be passed to layer or variable constructors, or called directly with a shape to return a Tensor.

Examples

```
# Standalone usage:
initializer <- initializer_lecun_normal()
values <- initializer(shape = c(2, 2))


# Usage in a Keras layer:
initializer <- initializer_lecun_normal()
layer <- layer_dense(units = 3, kernel_initializer = initializer)
```

Reference

- Klambauer et al., 2017

See Also

Other random initializers:
`initializer_glorot_normal()`
`initializer_glorot_uniform()`
`initializer_he_normal()`
`initializer_he_uniform()`
`initializer_lecun_uniform()`
`initializer_orthogonal()`
`initializer_random_normal()`
`initializer_random_uniform()`
`initializer_truncated_normal()`
`initializer_variance_scaling()`

Other initializers:
`initializer_constant()`
`initializer_glorot_normal()`
`initializer_glorot_uniform()`
`initializer_he_normal()`
`initializer_he_uniform()`
`initializer_identity()`
`initializer_lecun_uniform()`
`initializer_ones()`
`initializer_orthogonal()`
`initializer_random_normal()`
`initializer_random_uniform()`
`initializer_truncated_normal()`
`initializer_variance_scaling()`
`initializer_zeros()`

initializer_lecun_uniform

Lecun uniform initializer.

Description

Draws samples from a uniform distribution within `[-limit, limit]`, where `limit = sqrt(3 / fan_in)` (`fan_in` is the number of input units in the weight tensor).

Usage

```
initializer_lecun_uniform(seed = NULL)
```

Arguments

`seed`	An integer or instance of `random_seed_generator()`. Used to make the behavior of the initializer deterministic. Note that an initializer seeded with an integer or `NULL` (unseeded) will produce the same random values across multiple calls. To get different random values across multiple calls, use as seed an instance of `random_seed_generator()`.

Value

An `Initializer` instance that can be passed to layer or variable constructors, or called directly with a shape to return a Tensor.

Examples

```
# Standalone usage:
initializer <- initializer_lecun_uniform()
values <- initializer(shape = c(2, 2))

# Usage in a Keras layer:
initializer <- initializer_lecun_uniform()
layer <- layer_dense(units = 3, kernel_initializer = initializer)
```

Reference

- Klambauer et al., 2017

See Also

Other random initializers:
`initializer_glorot_normal()`
`initializer_glorot_uniform()`
`initializer_he_normal()`
`initializer_he_uniform()`
`initializer_lecun_normal()`

initializer_orthogonal()
initializer_random_normal()
initializer_random_uniform()
initializer_truncated_normal()
initializer_variance_scaling()

Other initializers:
initializer_constant()
initializer_glorot_normal()
initializer_glorot_uniform()
initializer_he_normal()
initializer_he_uniform()
initializer_identity()
initializer_lecun_normal()
initializer_ones()
initializer_orthogonal()
initializer_random_normal()
initializer_random_uniform()
initializer_truncated_normal()
initializer_variance_scaling()
initializer_zeros()

---

initializer_ones *Initializer that generates tensors initialized to 1.*

Description

Also available via the shortcut function ones.

Usage

```
initializer_ones()
```

Value

An Initializer instance that can be passed to layer or variable constructors, or called directly with a shape to return a Tensor.

Examples

```
# Standalone usage:
initializer <- initializer_ones()
values <- initializer(shape = c(2, 2))

# Usage in a Keras layer:
initializer <- initializer_ones()
layer <- layer_dense(units = 3, kernel_initializer = initializer)
```

See Also

- https://keras.io/api/layers/initializers#ones-class

Other constant initializers:
`initializer_constant()`
`initializer_identity()`
`initializer_zeros()`

Other initializers:
`initializer_constant()`
`initializer_glorot_normal()`
`initializer_glorot_uniform()`
`initializer_he_normal()`
`initializer_he_uniform()`
`initializer_identity()`
`initializer_lecun_normal()`
`initializer_lecun_uniform()`
`initializer_orthogonal()`
`initializer_random_normal()`
`initializer_random_uniform()`
`initializer_truncated_normal()`
`initializer_variance_scaling()`
`initializer_zeros()`

`initializer_orthogonal` *Initializer that generates an orthogonal matrix.*

Description

If the shape of the tensor to initialize is two-dimensional, it is initialized with an orthogonal matrix obtained from the QR decomposition of a matrix of random numbers drawn from a normal distribution. If the matrix has fewer rows than columns then the output will have orthogonal rows. Otherwise, the output will have orthogonal columns.

If the shape of the tensor to initialize is more than two-dimensional, a matrix of shape `(shape[1] * ... * shape[n - 1], sha` is initialized, where `n` is the length of the shape vector. The matrix is subsequently reshaped to give a tensor of the desired shape.

Usage

```
initializer_orthogonal(gain = 1, seed = NULL)
```

Arguments

`gain`	Multiplicative factor to apply to the orthogonal matrix.
`seed`	An integer. Used to make the behavior of the initializer deterministic.

Value

An Initializer instance that can be passed to layer or variable constructors, or called directly with a shape to return a Tensor.

Examples

```
# Standalone usage:
initializer <- initializer_orthogonal()
values <- initializer(shape = c(2, 2))

# Usage in a Keras layer:
initializer <- initializer_orthogonal()
layer <- layer_dense(units = 3, kernel_initializer = initializer)
```

Reference

- Saxe et al., 2014

See Also

Other random initializers:
initializer_glorot_normal()
initializer_glorot_uniform()
initializer_he_normal()
initializer_he_uniform()
initializer_lecun_normal()
initializer_lecun_uniform()
initializer_random_normal()
initializer_random_uniform()
initializer_truncated_normal()
initializer_variance_scaling()

Other initializers:
initializer_constant()
initializer_glorot_normal()
initializer_glorot_uniform()
initializer_he_normal()
initializer_he_uniform()
initializer_identity()
initializer_lecun_normal()
initializer_lecun_uniform()
initializer_ones()
initializer_random_normal()
initializer_random_uniform()
initializer_truncated_normal()
initializer_variance_scaling()
initializer_zeros()

`initializer_random_normal` *Random normal initializer.*

Description

Draws samples from a normal distribution for given parameters.

Usage

```
initializer_random_normal(mean = 0, stddev = 0.05, seed = NULL)
```

Arguments

`mean`	A numeric scalar. Mean of the random values to generate.
`stddev`	A numeric scalar. Standard deviation of the random values to generate.
`seed`	An integer or instance of `random_seed_generator()`. Used to make the behavior of the initializer deterministic. Note that an initializer seeded with an integer or `NULL` (unseeded) will produce the same random values across multiple calls. To get different random values across multiple calls, use as seed an instance of `random_seed_generator()`.

Value

An `Initializer` instance that can be passed to layer or variable constructors, or called directly with a shape to return a Tensor.

Examples

```
# Standalone usage:
initializer <- initializer_random_normal(mean = 0.0, stddev = 1.0)
values <- initializer(shape = c(2, 2))

# Usage in a Keras layer:
initializer <- initializer_random_normal(mean = 0.0, stddev = 1.0)
layer <- layer_dense(units = 3, kernel_initializer = initializer)
```

See Also

- `https://keras.io/api/layers/initializers#randomnormal-class`

Other random initializers:
`initializer_glorot_normal()`
`initializer_glorot_uniform()`
`initializer_he_normal()`
`initializer_he_uniform()`
`initializer_lecun_normal()`

initializer_lecun_uniform()
initializer_orthogonal()
initializer_random_uniform()
initializer_truncated_normal()
initializer_variance_scaling()

Other initializers:
initializer_constant()
initializer_glorot_normal()
initializer_glorot_uniform()
initializer_he_normal()
initializer_he_uniform()
initializer_identity()
initializer_lecun_normal()
initializer_lecun_uniform()
initializer_ones()
initializer_orthogonal()
initializer_random_uniform()
initializer_truncated_normal()
initializer_variance_scaling()
initializer_zeros()

initializer_random_uniform

Random uniform initializer.

Description

Draws samples from a uniform distribution for given parameters.

Usage

```
initializer_random_uniform(minval = -0.05, maxval = 0.05, seed = NULL)
```

Arguments

minval	A numeric scalar or a scalar keras tensor. Lower bound of the range of random values to generate (inclusive).
maxval	A numeric scalar or a scalar keras tensor. Upper bound of the range of random values to generate (exclusive).
seed	An integer or instance of random_seed_generator(). Used to make the behavior of the initializer deterministic. Note that an initializer seeded with an integer or NULL (unseeded) will produce the same random values across multiple calls. To get different random values across multiple calls, use as seed an instance of random_seed_generator().

Value

An `Initializer` instance that can be passed to layer or variable constructors, or called directly with a shape to return a Tensor.

Examples

```
# Standalone usage:
initializer <- initializer_random_uniform(minval = 0.0, maxval = 1.0)
values <- initializer(shape = c(2, 2))


# Usage in a Keras layer:
initializer <- initializer_random_uniform(minval = 0.0, maxval = 1.0)
layer <- layer_dense(units = 3, kernel_initializer = initializer)
```

See Also

- `https://keras.io/api/layers/initializers#randomuniform-class`

Other random initializers:
`initializer_glorot_normal()`
`initializer_glorot_uniform()`
`initializer_he_normal()`
`initializer_he_uniform()`
`initializer_lecun_normal()`
`initializer_lecun_uniform()`
`initializer_orthogonal()`
`initializer_random_normal()`
`initializer_truncated_normal()`
`initializer_variance_scaling()`

Other initializers:
`initializer_constant()`
`initializer_glorot_normal()`
`initializer_glorot_uniform()`
`initializer_he_normal()`
`initializer_he_uniform()`
`initializer_identity()`
`initializer_lecun_normal()`
`initializer_lecun_uniform()`
`initializer_ones()`
`initializer_orthogonal()`
`initializer_random_normal()`
`initializer_truncated_normal()`
`initializer_variance_scaling()`
`initializer_zeros()`

initializer_truncated_normal

Initializer that generates a truncated normal distribution.

Description

The values generated are similar to values from a `RandomNormal` initializer, except that values more than two standard deviations from the mean are discarded and re-drawn.

Usage

```
initializer_truncated_normal(mean = 0, stddev = 0.05, seed = NULL)
```

Arguments

`mean`	A numeric scalar. Mean of the random values to generate.
`stddev`	A numeric scalar. Standard deviation of the random values to generate.
`seed`	An integer or instance of `random_seed_generator()`. Used to make the behavior of the initializer deterministic. Note that an initializer seeded with an integer or `NULL` (unseeded) will produce the same random values across multiple calls. To get different random values across multiple calls, use as seed an instance of `random_seed_generator()`.

Value

An `Initializer` instance that can be passed to layer or variable constructors, or called directly with a shape to return a Tensor.

Examples

```
# Standalone usage:
initializer <- initializer_truncated_normal(mean = 0, stddev = 1)
values <- initializer(shape = c(2, 2))

# Usage in a Keras layer:
initializer <- initializer_truncated_normal(mean = 0, stddev = 1)
layer <- layer_dense(units = 3, kernel_initializer = initializer)
```

See Also

- https://keras.io/api/layers/initializers#truncatednormal-class

Other random initializers:
`initializer_glorot_normal()`
`initializer_glorot_uniform()`
`initializer_he_normal()`
`initializer_he_uniform()`

initializer_lecun_normal()
initializer_lecun_uniform()
initializer_orthogonal()
initializer_random_normal()
initializer_random_uniform()
initializer_variance_scaling()

Other initializers:
initializer_constant()
initializer_glorot_normal()
initializer_glorot_uniform()
initializer_he_normal()
initializer_he_uniform()
initializer_identity()
initializer_lecun_normal()
initializer_lecun_uniform()
initializer_ones()
initializer_orthogonal()
initializer_random_normal()
initializer_random_uniform()
initializer_variance_scaling()
initializer_zeros()

---

initializer_variance_scaling

Initializer that adapts its scale to the shape of its input tensors.

Description

With `distribution = "truncated_normal" or "untruncated_normal"`, samples are drawn from a truncated/untruncated normal distribution with a mean of zero and a standard deviation (after truncation, if used) `stddev = sqrt(scale / n)`, where `n` is:

- number of input units in the weight tensor, if `mode = "fan_in"`
- number of output units, if `mode = "fan_out"`
- average of the numbers of input and output units, if `mode = "fan_avg"`

With `distribution = "uniform"`, samples are drawn from a uniform distribution within `[-limit, limit]`, where `limit = sqrt(3 * scale / n)`.

Usage

```
initializer_variance_scaling(
  scale = 1,
  mode = "fan_in",
  distribution = "truncated_normal",
  seed = NULL
)
```

Arguments

scale	Scaling factor (positive float).
mode	One of "fan_in", "fan_out", "fan_avg".
distribution	Random distribution to use. One of "truncated_normal", "untruncated_normal", or "uniform".
seed	An integer or instance of random_seed_generator(). Used to make the behavior of the initializer deterministic. Note that an initializer seeded with an integer or NULL (unseeded) will produce the same random values across multiple calls. To get different random values across multiple calls, use as seed an instance of random_seed_generator().

Value

An Initializer instance that can be passed to layer or variable constructors, or called directly with a shape to return a Tensor.

Examples

```
# Standalone usage:
initializer <- initializer_variance_scaling(scale = 0.1, mode = 'fan_in',
                                            distribution = 'uniform')
values <- initializer(shape = c(2, 2))

# Usage in a Keras layer:
initializer <- initializer_variance_scaling(scale = 0.1, mode = 'fan_in',
                                            distribution = 'uniform')
layer <- layer_dense(units = 3, kernel_initializer = initializer)
```

See Also

- https://keras.io/api/layers/initializers#variancescaling-class

Other random initializers:
initializer_glorot_normal()
initializer_glorot_uniform()
initializer_he_normal()
initializer_he_uniform()
initializer_lecun_normal()
initializer_lecun_uniform()
initializer_orthogonal()
initializer_random_normal()
initializer_random_uniform()
initializer_truncated_normal()

Other initializers:
initializer_constant()
initializer_glorot_normal()
initializer_glorot_uniform()

initializer_he_normal()
initializer_he_uniform()
initializer_identity()
initializer_lecun_normal()
initializer_lecun_uniform()
initializer_ones()
initializer_orthogonal()
initializer_random_normal()
initializer_random_uniform()
initializer_truncated_normal()
initializer_zeros()

---

initializer_zeros *Initializer that generates tensors initialized to 0.*

Description

Initializer that generates tensors initialized to 0.

Usage

```
initializer_zeros()
```

Value

An `Initializer` instance that can be passed to layer or variable constructors, or called directly with a shape to return a Tensor.

Examples

```
# Standalone usage:
initializer <- initializer_zeros()
values <- initializer(shape = c(2, 2))

# Usage in a Keras layer:
initializer <- initializer_zeros()
layer <- layer_dense(units = 3, kernel_initializer = initializer)
```

See Also

- https://keras.io/api/layers/initializers#zeros-class

Other constant initializers:
initializer_constant()
initializer_identity()
initializer_ones()

Other initializers:
initializer_constant()
initializer_glorot_normal()
initializer_glorot_uniform()
initializer_he_normal()
initializer_he_uniform()
initializer_identity()
initializer_lecun_normal()
initializer_lecun_uniform()
initializer_ones()
initializer_orthogonal()
initializer_random_normal()
initializer_random_uniform()
initializer_truncated_normal()
initializer_variance_scaling()

install_keras *Install Keras*

Description

This function will install Keras along with a selected backend, including all Python dependencies.

Usage

```
install_keras(
  envname = "r-keras",
  ...,
  extra_packages = c("scipy", "pandas", "Pillow", "pydot", "ipython",
    "tensorflow_datasets"),
  python_version = ">=3.9,<=3.11",
  backend = c("tensorflow", "jax"),
  gpu = NA,
  restart_session = TRUE
)
```

Arguments

envname	Name of or path to a Python virtual environment
...	reserved for future compatability.
extra_packages	Additional Python packages to install alongside Keras
python_version	Passed on to reticulate::virtualenv_starter()
backend	Which backend(s) to install. Accepted values include "tensorflow", "jax" and "pytorch"
gpu	whether to install a GPU capable version of the backend.

`restart_session`
: Whether to restart the R session after installing (note this will only occur within RStudio).

Value

No return value, called for side effects.

See Also

`tensorflow::install_tensorflow()`

---

keras — *Main Keras module*

Description

The `keras` module object is the equivalent of `reticulate::import("keras")` and provided mainly as a convenience.

Format

An object of class `python.builtin.module`

Value

the keras Python module

---

keras_input — *Create a Keras tensor (Functional API input).*

Description

A Keras tensor is a symbolic tensor-like object, which we augment with certain attributes that allow us to build a Keras model just by knowing the inputs and outputs of the model.

For instance, if `a`, `b` and `c` are Keras tensors, it becomes possible to do: `model <- keras_model(input = c(a, b), output = c)`

Usage

```
keras_input(
  shape = NULL,
  batch_size = NULL,
  dtype = NULL,
  sparse = NULL,
  batch_shape = NULL,
  name = NULL,
  tensor = NULL
)
```

Arguments

shape	A shape list (list of integers or NULL objects), not including the batch size. For instance, shape = c(32) indicates that the expected input will be batches of 32-dimensional vectors. Elements of this list can be NULL or NA; NULL/NA elements represent dimensions where the shape is not known and may vary (e.g. sequence length).
batch_size	Optional static batch size (integer).
dtype	The data type expected by the input, as a string (e.g. "float32", "int32"...)
sparse	A boolean specifying whether the expected input will be sparse tensors. Note that, if sparse is FALSE, sparse tensors can still be passed into the input - they will be densified with a default value of 0. This feature is only supported with the TensorFlow backend. Defaults to FALSE.
batch_shape	Shape, including the batch dim.
name	Optional name string for the layer. Should be unique in a model (do not reuse the same name twice). It will be autogenerated if it isn't provided.
tensor	Optional existing tensor to wrap into the Input layer. If set, the layer will use this tensor rather than creating a new placeholder tensor.

Value

A Keras tensor, which can passed to the inputs argument of (keras_model()).

Examples

```
# This is a logistic regression in Keras
input <- layer_input(shape=c(32))
output <- input |> layer_dense(16, activation='softmax')
model <- keras_model(input, output)
```

See Also

Other model creation:
keras_model()
keras_model_sequential()

---

keras_model *Keras Model (Functional API)*

Description

A model is a directed acyclic graph of layers.

Usage

```
keras_model(inputs = NULL, outputs = NULL, ...)
```

Arguments

`inputs`	Input tensor(s) (from `keras_input()`)
`outputs`	Output tensors (from calling layers with `inputs`)
`...`	Any additional arguments

Value

A `Model` instance.

Examples

```
library(keras3)

# input tensor
inputs <- keras_input(shape = c(784))

# outputs compose input + dense layers
predictions <- inputs |>
  layer_dense(units = 64, activation = 'relu') |>
  layer_dense(units = 64, activation = 'relu') |>
  layer_dense(units = 10, activation = 'softmax')

# create and compile model
model <- keras_model(inputs = inputs, outputs = predictions)
model |> compile(
  optimizer = 'rmsprop',
  loss = 'categorical_crossentropy',
  metrics = c('accuracy')
)
```

See Also

Other model functions:
`get_config()`
`get_layer()`
`keras_model_sequential()`
`pop_layer()`
`summary.keras.src.models.model.Model()`

Other model creation:
`keras_input()`
`keras_model_sequential()`

keras_model_sequential *Keras Model composed of a linear stack of layers*

Description

Keras Model composed of a linear stack of layers

Usage

```
keras_model_sequential(
  input_shape = NULL,
  name = NULL,
  ...,
  input_dtype = NULL,
  input_batch_size = NULL,
  input_sparse = NULL,
  input_batch_shape = NULL,
  input_name = NULL,
  input_tensor = NULL,
  trainable = TRUE,
  layers = list()
)
```

Arguments

input_shape	A shape integer vector, not including the batch size. For instance, shape=c(32) indicates that the expected input will be batches of 32-dimensional vectors. Elements of this shape can be NA; NA elements represent dimensions where the shape is not known and may vary (e.g. sequence length).
name	Name of model
...	additional arguments passed on to keras.layers.InputLayer.
input_dtype	The data type expected by the input, as a string (e.g. "float32", "int32"...)
input_batch_size	Optional static batch size (integer).
input_sparse	A boolean specifying whether the expected input will be sparse tensors. Note that, if sparse is FALSE, sparse tensors can still be passed into the input - they will be densified with a default value of 0. This feature is only supported with the TensorFlow backend. Defaults to FALSE.
input_batch_shape	An optional way to specify batch_size and input_shape as one argument.
input_name	Optional name string for the input layer. Should be unique in a model (do not reuse the same name twice). It will be autogenerated if it isn't provided.
input_tensor	Optional existing tensor to wrap into the InputLayer. If set, the layer will use this tensor rather than creating a new placeholder tensor.

trainable	Boolean, whether the model's variables should be trainable. You can also change the trainable status of a model/layer with `freeze_weights()` and `unfreeze_weights()`.
layers	List of layers to add to the model.

Value

A `Sequential` model instance.

Examples

```
model <- keras_model_sequential(input_shape = c(784))
model |>
  layer_dense(units = 32) |>
  layer_activation('relu') |>
  layer_dense(units = 10) |>
  layer_activation('softmax')

model |> compile(
  optimizer = 'rmsprop',
  loss = 'categorical_crossentropy',
  metrics = c('accuracy')
)

model
```

```
## Model: "sequential"
## +---------------------------------+------------------------+---------------+
## | Layer (type)                    | Output Shape           |       Param # |
## +=================================+========================+===============+
## | dense_1 (Dense)                 | (None, 32)             |        25,120 |
## +---------------------------------+------------------------+---------------+
## | activation_1 (Activation)       | (None, 32)             |             0 |
## +---------------------------------+------------------------+---------------+
## | dense (Dense)                   | (None, 10)             |           330 |
## +---------------------------------+------------------------+---------------+
## | activation (Activation)         | (None, 10)             |             0 |
## +---------------------------------+------------------------+---------------+
##  Total params: 25,450 (99.41 KB)
##  Trainable params: 25,450 (99.41 KB)
##  Non-trainable params: 0 (0.00 B)
```

Note

If `input_shape` is omitted, then the model layer shapes, including the final model output shape, will not be known until the model is built, either by calling the model with an input tensor/array like `model(input)`, (possibly via `fit()`/`evaluate()`/`predict()`), or by explicitly calling `model$build(input_shape)`.

See Also

Other model functions:
`get_config()`
`get_layer()`
`keras_model()`
`pop_layer()`
`summary.keras.src.models.model.Model()`

Other model creation:
`keras_input()`
`keras_model()`

`Layer` *Define a custom* `Layer` *class.*

Description

A layer is a callable object that takes as input one or more tensors and that outputs one or more tensors. It involves *computation*, defined in the `call()` method, and a *state* (weight variables). State can be created:

- in `initialize()`, for instance via `self$add_weight()`;
- in the optional `build()` method, which is invoked by the first `call()` to the layer, and supplies the shape(s) of the input(s), which may not have been known at initialization time.

Layers are recursively composable: If you assign a Layer instance as an attribute of another Layer, the outer layer will start tracking the weights created by the inner layer. Nested layers should be instantiated in the `initialize()` method or `build()` method.

Users will just instantiate a layer and then treat it as a callable.

Usage

```
Layer(
  classname,
  initialize = NULL,
  call = NULL,
  build = NULL,
  get_config = NULL,
  ...,
  public = list(),
  private = list(),
  inherit = NULL,
  parent_env = parent.frame()
)
```

Arguments

classname	String, the name of the custom class. (Conventionally, CamelCase).
initialize, call, build, get_config	Recommended methods to implement. See description and details sections.
..., public	Additional methods or public members of the custom class.
private	Named list of R objects (typically, functions) to include in instance private environments. private methods will have all the same symbols in scope as public methods (See section "Symbols in Scope"). Each instance will have it's own private environment. Any objects in private will be invisible from the Keras framework and the Python runtime.
inherit	What the custom class will subclass. By default, the base keras class.
parent_env	The R environment that all class methods will have as a grandparent.

Value

A composing layer constructor, with similar behavior to other layer functions like `layer_dense()`. The first argument of the returned function will be `object`, enabling `initialize()`ing and `call()` the layer in one step while composing the layer with the pipe, like

```
layer_foo <- Layer("Foo", ....)
output <- inputs |> layer_foo()
```

To only `initialize()` a layer instance and not `call()` it, pass a missing or `NULL` value to `object`, or pass all arguments to `initialize()` by name.

```
layer <- layer_dense(units = 2, activation = "relu")
layer <- layer_dense(NULL, 2, activation = "relu")
layer <- layer_dense(, 2, activation = "relu")

# then you can call() the layer in a separate step
outputs <- inputs |> layer()
```

Symbols in scope

All R function custom methods (public and private) will have the following symbols in scope:

- `self`: The custom class instance.
- `super`: The custom class superclass.
- `private`: An R environment specific to the class instance. Any objects assigned here are invisible to the Keras framework.
- `__class__` and `as.symbol(classname)`: the custom class type object.

Attributes

- name: The name of the layer (string).
- dtype: Dtype of the layer's weights. Alias of layer$variable_dtype.
- variable_dtype: Dtype of the layer's weights.
- compute_dtype: The dtype of the layer's computations. Layers automatically cast inputs to this dtype, which causes the computations and output to also be in this dtype. When mixed precision is used with a keras$mixed_precision$DTypePolicy, this will be different than variable_dtype.
- trainable_weights: List of variables to be included in backprop.
- non_trainable_weights: List of variables that should not be included in backprop.
- weights: The concatenation of the lists trainable_weights and non_trainable_weights (in this order).
- trainable: Whether the layer should be trained (boolean), i.e. whether its potentially-trainable weights should be returned as part of layer$trainable_weights.
- input_spec: Optional (list of) InputSpec object(s) specifying the constraints on inputs that can be accepted by the layer.

We recommend that custom Layers implement the following methods:

- initialize(): Defines custom layer attributes, and creates layer weights that do not depend on input shapes, using add_weight(), or other state.
- build(input_shape): This method can be used to create weights that depend on the shape(s) of the input(s), using add_weight(), or other state. Calling call() will automatically build the layer (if it has not been built yet) by calling build().
- call(...): Method called after making sure build() has been called. call() performs the logic of applying the layer to the input arguments. Two reserved arguments you can optionally use in call() are:
  1. training (boolean, whether the call is in inference mode or training mode).
  2. mask (boolean tensor encoding masked timesteps in the input, used e.g. in RNN layers).

  A typical signature for this method is call(inputs), and user could optionally add training and mask if the layer need them.
- get_config(): Returns a named list containing the configuration used to initialize this layer. If the list names differ from the arguments in initialize(), then override from_config() as well. This method is used when saving the layer or a model that contains this layer.

Examples

Here's a basic example: a layer with two variables, w and b, that returns y <- (w %*% x) + b. It shows how to implement build() and call(). Variables set as attributes of a layer are tracked as weights of the layers (in layer$weights).

```
layer_simple_dense <- Layer(
  "SimpleDense",
  initialize = function(units = 32) {
    super$initialize()
    self$units <- units
  },

  # Create the state of the layer (weights)
  build = function(input_shape) {
    self$kernel <- self$add_weight(
      shape = shape(tail(input_shape, 1), self$units),
      initializer = "glorot_uniform",
      trainable = TRUE,
      name = "kernel"
    )
    self$bias = self$add_weight(
      shape = shape(self$units),
      initializer = "zeros",
      trainable = TRUE,
      name = "bias"
    )
  },

  # Defines the computation
  call = function(self, inputs) {
    op_matmul(inputs, self$kernel) + self$bias
  }
)

# Instantiates the layer.
# Supply missing `object` arg to skip invoking `call()` and instead return
# the Layer instance
linear_layer <- layer_simple_dense(, 4)

# This will call `build(input_shape)` and create the weights,
# and then invoke `call()`.
y <- linear_layer(op_ones(c(2, 2)))
stopifnot(length(linear_layer$weights) == 2)

# These weights are trainable, so they're listed in `trainable_weights`:
stopifnot(length(linear_layer$trainable_weights) == 2)
```

Besides trainable weights, updated via backpropagation during training, layers can also have non-trainable weights. These weights are meant to be updated manually during `call()`. Here's a example layer that computes the running sum of its inputs:

```
layer_compute_sum <- Layer(
  classname = "ComputeSum",

  initialize = function(input_dim) {
    super$initialize()

    # Create a non-trainable weight.
    self$total <- self$add_weight(
      shape = shape(),
      initializer = "zeros",
      trainable = FALSE,
      name = "total"
    )
  },

  call = function(inputs) {
    self$total$assign(self$total + op_sum(inputs))
    self$total
  }
)

my_sum <- layer_compute_sum(, 2)
x <- op_ones(c(2, 2))
y <- my_sum(x)

stopifnot(exprs = {
  all.equal(my_sum$weights,               list(my_sum$total))
  all.equal(my_sum$non_trainable_weights, list(my_sum$total))
  all.equal(my_sum$trainable_weights,     list())
})
```

Methods available

-
  ```
  initialize(...,
             activity_regularizer = NULL,
             trainable = TRUE,
             dtype = NULL,
             autocast = TRUE,
             name = NULL)
  ```

  Initialize self. This method is typically called from a custom `initialize()` method. Example:

  ```
  layer_my_layer <- Layer("MyLayer",
    initialize = function(units, ..., dtype = NULL, name = NULL) {
      super$initialize(..., dtype = dtype, name = name)
      # .... finish initializing `self` instance
    }
  )
  ```

  Args:

  - trainable: Boolean, whether the layer's variables should be trainable.
  - name: String name of the layer.
  - dtype: The dtype of the layer's computations and weights. Can also be a `keras$DTypePolicy`, which allows the computation and weight dtype to differ. Defaults to `NULL`. `NULL` means to use `config_dtype_policy()`, which is a `"float32"` policy unless set to different value (via `config_set_dtype_policy()`).

- `add_loss(loss)`

  Can be called inside of the `call()` method to add a scalar loss.

  Example:

  ```
  Layer("MyLayer",
    ...
    call = function(x) {
      self$add_loss(op_sum(x))
      x
    }
  ```

- `add_metric()`
- `add_variable(...)`

  Add a weight variable to the layer.

  Alias of `add_weight()`.

-
  ```
  add_weight(shape = NULL,
             initializer = NULL,
             dtype = NULL,
             trainable = TRUE,
             autocast = TRUE,
             regularizer = NULL,
             constraint = NULL,
             aggregation = 'mean',
             name = NULL)
  ```

  Add a weight variable to the layer.

  Args:

  – `shape`: shape for the variable (as defined by `shape()`) Must be fully-defined (no `NA`/`NULL`/`-1` entries). Defaults to `()` (scalar) if unspecified.
  – `initializer`: Initializer object to use to populate the initial variable value, or string name of a built-in initializer (e.g. `"random_normal"`). If unspecified, defaults to `"glorot_uniform"` for floating-point variables and to `"zeros"` for all other types (e.g. int, bool).
  – `dtype`: Dtype of the variable to create, e.g. `"float32"`. If unspecified, defaults to the layer's variable dtype (which itself defaults to `"float32"` if unspecified).
  – `trainable`: Boolean, whether the variable should be trainable via backprop or whether its updates are managed manually. Defaults to `TRUE`.
  – `autocast`: Boolean, whether to autocast layers variables when accessing them. Defaults to `TRUE`.
  – `regularizer`: Regularizer object to call to apply penalty on the weight. These penalties are summed into the loss function during optimization. Defaults to `NULL`.
  – `constraint`: Constraint object to call on the variable after any optimizer update, or string name of a built-in constraint. Defaults to `NULL`.
  – `aggregation`: String, one of `'mean'`, `'sum'`, `'only_first_replica'`. Annotates the variable with the type of multi-replica aggregation to be used for this variable when writing custom data parallel training loops.
  – `name`: String name of the variable. Useful for debugging purposes.

Returns:

A backend tensor, wrapped in a `KerasVariable` class. The `KerasVariable` class has

Methods:

  - `assign(value)`
  - `assign_add(value)`
  - `assign_sub(value)`
  - `numpy()` (calling `as.array(<variable>)` is preferred)

Properties/Attributes:

  - `value`
  - `dtype`
  - `ndim`
  - `shape` (calling `shape(<variable>)` is preferred)
  - `trainable`

- `build(input_shape)`
- `build_from_config(config)`

  Builds the layer's states with the supplied config (named list of args).

  By default, this method calls the `do.call(build, config$input_shape)` method, which creates weights based on the layer's input shape in the supplied config. If your config contains other information needed to load the layer's state, you should override this method.

  Args:

  - `config`: Named list containing the input shape associated with this layer.
- `call(...)`

  See description above
- `compute_mask(inputs, previous_mask)`
- `compute_output_shape(...)`
- `compute_output_spec(...)`
- `count_params()`

  Count the total number of scalars composing the weights.

  Returns: An integer count.
- `get_build_config()`

  Returns a named list with the layer's input shape.

  This method returns a config (named list) that can be used by `build_from_config(config)` to create all states (e.g. Variables and Lookup tables) needed by the layer.

  By default, the config only contains the input shape that the layer was built with. If you're writing a custom layer that creates state in an unusual way, you should override this method to make sure this state is already created when Keras attempts to load its value upon model loading.

  Returns: A named list containing the input shape associated with the layer.
- `get_config()`

  Returns the config of the object.

  An object config is a named list (serializable) containing the information needed to re-instantiate it. The config is expected to be serializable to JSON, and is expected to consist of a (potentially complex, nested) structure of names lists consisting of simple objects like strings, ints.

- `get_weights()`

  Return the values of `layer$weights` as a list of R or NumPy arrays.

- `quantize(mode)`

  Currently, only the `Dense` and `EinsumDense` layers support in-place quantization via this `quantize()` method.

  Example:

  ```
  model$quantize("int8") # quantize model in-place
  model |> predict(data) # faster inference
  ```

- `quantized_call(...)`
- `load_own_variables(store)`

  Loads the state of the layer.

  You can override this method to take full control of how the state of the layer is loaded upon calling `load_model()`.

  Args:

  – `store`: Named list from which the state of the model will be loaded.

- `save_own_variables(store)`

  Saves the state of the layer.

  You can override this method to take full control of how the state of the layer is saved upon calling `save_model()`.

  Args:

  – `store`: Named list where the state of the model will be saved.

- `set_weights(weights)`

  Sets the values of `weights` from a list of R or NumPy arrays.

-
  ```
  stateless_call(trainable_variables, non_trainable_variables,
                 ..., return_losses = FALSE)
  ```

  Call the layer without any side effects.

  Args:

  – `trainable_variables`: List of trainable variables of the model.

  – `non_trainable_variables`: List of non-trainable variables of the model.

  – `...`: Positional and named arguments to be passed to `call()`.

  – `return_losses`: If `TRUE`, `stateless_call()` will return the list of losses created during `call()` as part of its return values.

  Returns: An unnamed list. By default, returns `list(outputs, non_trainable_variables)`. If `return_losses = TRUE`, then returns `list(outputs, non_trainable_variables, losses)`.

  Note: `non_trainable_variables` include not only non-trainable weights such as `BatchNormalization` statistics, but also RNG seed state (if there are any random operations part of the layer, such as dropout), and `Metric` state (if there are any metrics attached to the layer). These are all elements of state of the layer.

  Example:

```
model <- ...
data <- ...
trainable_variables <- model$trainable_variables
non_trainable_variables <- model$non_trainable_variables
# Call the model with zero side effects
c(outputs, non_trainable_variables) %<-% model$stateless_call(
    trainable_variables,
    non_trainable_variables,
    data
)
# Attach the updated state to the model
# (until you do this, the model is still in its pre-call state).
purrr::walk2(
  model$non_trainable_variables, non_trainable_variables,
  \(variable, value) variable$assign(value))
```

- `symbolic_call(...)`
- `from_config(config)`

  Creates a layer from its config.

  This is a class method, meaning, the R function will not have a `self` symbol (a class instance) in scope. Use `__class__` or the classname symbol provided when the `Layer()` was constructed) to resolve the class definition. The default implementation is:

  ```
  from_config = function(config) {
    do.call(`__class__`, config)
  }
  ```

  This method is the reverse of `get_config()`, capable of instantiating the same layer from the config named list. It does not handle layer connectivity (handled by Network), nor weights (handled by `set_weights()`).

  Args:

  – `config`: A named list, typically the output of `get_config()`.

  Returns: A layer instance.

Readonly properties:

- `compute_dtype` The dtype of the computations performed by the layer.
- `dtype` Alias of `layer$variable_dtype`.
- `input_dtype` The dtype layer inputs should be converted to.
- `losses` List of scalar losses from `add_loss()`, regularizers and sublayers.
- `metrics_variables` List of all metric variables.
- `non_trainable_variables` List of all non-trainable layer state.

  This extends `layer$non_trainable_weights` to include all state used by the layer including state for metrics and `SeedGenerator`s.
- `non_trainable_weights` List of all non-trainable weight variables of the layer.

  These are the weights that should not be updated by the optimizer during training. Unlike, `layer$non_trainable_variables` this excludes metric state and random seeds.

- `trainable_variables` List of all trainable layer state.
  This is equivalent to `layer$trainable_weights`.
- `trainable_weights` List of all trainable weight variables of the layer.
  These are the weights that get updated by the optimizer during training.
- `variable_dtype` The dtype of the state (weights) of the layer.
- `variables` List of all layer state, including random seeds.
  This extends `layer$weights` to include all state used by the layer including `SeedGenerators`.
  Note that metrics variables are not included here, use `metrics_variables` to visit all the metric variables.
- `weights` List of all weight variables of the layer.
  Unlike, `layer$variables` this excludes metric state and random seeds.
- `input` Retrieves the input tensor(s) of a symbolic operation.
  Only returns the tensor(s) corresponding to the *first time* the operation was called.
  Returns: Input tensor or list of input tensors.
- `output` Retrieves the output tensor(s) of a layer.
  Only returns the tensor(s) corresponding to the *first time* the operation was called.
  Returns: Output tensor or list of output tensors.

Data descriptors (Attributes):

- `input_spec`
- `supports_masking` Whether this layer supports computing a mask using `compute_mask`.
- `trainable` Settable boolean, whether this layer should be trainable or not.

See Also

- `https://keras.io/api/layers/base_layer#layer-class`

Other layers:
`layer_activation()`
`layer_activation_elu()`
`layer_activation_leaky_relu()`
`layer_activation_parametric_relu()`
`layer_activation_relu()`
`layer_activation_softmax()`
`layer_activity_regularization()`
`layer_add()`
`layer_additive_attention()`
`layer_alpha_dropout()`
`layer_attention()`
`layer_average()`
`layer_average_pooling_1d()`
`layer_average_pooling_2d()`
`layer_average_pooling_3d()`
`layer_batch_normalization()`
`layer_bidirectional()`

layer_category_encoding()
layer_center_crop()
layer_concatenate()
layer_conv_1d()
layer_conv_1d_transpose()
layer_conv_2d()
layer_conv_2d_transpose()
layer_conv_3d()
layer_conv_3d_transpose()
layer_conv_lstm_1d()
layer_conv_lstm_2d()
layer_conv_lstm_3d()
layer_cropping_1d()
layer_cropping_2d()
layer_cropping_3d()
layer_dense()
layer_depthwise_conv_1d()
layer_depthwise_conv_2d()
layer_discretization()
layer_dot()
layer_dropout()
layer_einsum_dense()
layer_embedding()
layer_feature_space()
layer_flatten()
layer_flax_module_wrapper()
layer_gaussian_dropout()
layer_gaussian_noise()
layer_global_average_pooling_1d()
layer_global_average_pooling_2d()
layer_global_average_pooling_3d()
layer_global_max_pooling_1d()
layer_global_max_pooling_2d()
layer_global_max_pooling_3d()
layer_group_normalization()
layer_group_query_attention()
layer_gru()
layer_hashed_crossing()
layer_hashing()
layer_identity()
layer_integer_lookup()
layer_jax_model_wrapper()
layer_lambda()
layer_layer_normalization()
layer_lstm()
layer_masking()
layer_max_pooling_1d()
layer_max_pooling_2d()

layer_max_pooling_3d()
layer_maximum()
layer_mel_spectrogram()
layer_minimum()
layer_multi_head_attention()
layer_multiply()
layer_normalization()
layer_permute()
layer_random_brightness()
layer_random_contrast()
layer_random_crop()
layer_random_flip()
layer_random_rotation()
layer_random_translation()
layer_random_zoom()
layer_repeat_vector()
layer_rescaling()
layer_reshape()
layer_resizing()
layer_rnn()
layer_separable_conv_1d()
layer_separable_conv_2d()
layer_simple_rnn()
layer_spatial_dropout_1d()
layer_spatial_dropout_2d()
layer_spatial_dropout_3d()
layer_spectral_normalization()
layer_string_lookup()
layer_subtract()
layer_text_vectorization()
layer_tfsm()
layer_time_distributed()
layer_torch_module_wrapper()
layer_unit_normalization()
layer_upsampling_1d()
layer_upsampling_2d()
layer_upsampling_3d()
layer_zero_padding_1d()
layer_zero_padding_2d()
layer_zero_padding_3d()
rnn_cell_gru()
rnn_cell_lstm()
rnn_cell_simple()
rnn_cells_stack()

layer_activation *Applies an activation function to an output.*

Description

Applies an activation function to an output.

Usage

```
layer_activation(object, activation, ...)
```

Arguments

object	Object to compose the layer with. A tensor, array, or sequential model.
activation	Activation function. It could be a callable, or the name of an activation from the `keras3::activation_*` namespace.
...	Base layer keyword arguments, such as `name` and `dtype`.

Value

The return value depends on the value provided for the first argument. If `object` is:

- a `keras_model_sequential()`, then the layer is added to the sequential model (which is modified in place). To enable piping, the sequential model is also returned, invisibly.
- a `keras_input()`, then the output tensor from calling `layer(input)` is returned.
- `NULL` or missing, then a `Layer` instance is returned.

Examples

```
x <- array(c(-3, -1, 0, 2))
layer <- layer_activation(activation = 'relu')
layer(x)


## tf.Tensor([0. 0. 0. 2.], shape=(4), dtype=float32)



layer <- layer_activation(activation = activation_relu)
layer(x)


## tf.Tensor([0. 0. 0. 2.], shape=(4), dtype=float32)



layer <- layer_activation(activation = op_relu)
layer(x)


## tf.Tensor([0. 0. 0. 2.], shape=(4), dtype=float32)
```

See Also

- https://keras.io/api/layers/core_layers/activation#activation-class

Other activation layers:
layer_activation_elu()
layer_activation_leaky_relu()
layer_activation_parametric_relu()
layer_activation_relu()
layer_activation_softmax()

Other layers:
Layer()
layer_activation_elu()
layer_activation_leaky_relu()
layer_activation_parametric_relu()
layer_activation_relu()
layer_activation_softmax()
layer_activity_regularization()
layer_add()
layer_additive_attention()
layer_alpha_dropout()
layer_attention()
layer_average()
layer_average_pooling_1d()
layer_average_pooling_2d()
layer_average_pooling_3d()
layer_batch_normalization()
layer_bidirectional()
layer_category_encoding()
layer_center_crop()
layer_concatenate()
layer_conv_1d()
layer_conv_1d_transpose()
layer_conv_2d()
layer_conv_2d_transpose()
layer_conv_3d()
layer_conv_3d_transpose()
layer_conv_lstm_1d()
layer_conv_lstm_2d()
layer_conv_lstm_3d()
layer_cropping_1d()
layer_cropping_2d()
layer_cropping_3d()
layer_dense()
layer_depthwise_conv_1d()
layer_depthwise_conv_2d()
layer_discretization()
layer_dot()

layer_dropout()
layer_einsum_dense()
layer_embedding()
layer_feature_space()
layer_flatten()
layer_flax_module_wrapper()
layer_gaussian_dropout()
layer_gaussian_noise()
layer_global_average_pooling_1d()
layer_global_average_pooling_2d()
layer_global_average_pooling_3d()
layer_global_max_pooling_1d()
layer_global_max_pooling_2d()
layer_global_max_pooling_3d()
layer_group_normalization()
layer_group_query_attention()
layer_gru()
layer_hashed_crossing()
layer_hashing()
layer_identity()
layer_integer_lookup()
layer_jax_model_wrapper()
layer_lambda()
layer_layer_normalization()
layer_lstm()
layer_masking()
layer_max_pooling_1d()
layer_max_pooling_2d()
layer_max_pooling_3d()
layer_maximum()
layer_mel_spectrogram()
layer_minimum()
layer_multi_head_attention()
layer_multiply()
layer_normalization()
layer_permute()
layer_random_brightness()
layer_random_contrast()
layer_random_crop()
layer_random_flip()
layer_random_rotation()
layer_random_translation()
layer_random_zoom()
layer_repeat_vector()
layer_rescaling()
layer_reshape()
layer_resizing()
layer_rnn()

layer_separable_conv_1d()
layer_separable_conv_2d()
layer_simple_rnn()
layer_spatial_dropout_1d()
layer_spatial_dropout_2d()
layer_spatial_dropout_3d()
layer_spectral_normalization()
layer_string_lookup()
layer_subtract()
layer_text_vectorization()
layer_tfsm()
layer_time_distributed()
layer_torch_module_wrapper()
layer_unit_normalization()
layer_upsampling_1d()
layer_upsampling_2d()
layer_upsampling_3d()
layer_zero_padding_1d()
layer_zero_padding_2d()
layer_zero_padding_3d()
rnn_cell_gru()
rnn_cell_lstm()
rnn_cell_simple()
rnn_cells_stack()

---

layer_activation_elu *Applies an Exponential Linear Unit function to an output.*

Description

Formula:

```
f(x) = alpha * (exp(x) - 1.) for x < 0
f(x) = x for x >= 0
```

Usage

```
layer_activation_elu(object, alpha = 1, ...)
```

Arguments

object	Object to compose the layer with. A tensor, array, or sequential model.
alpha	float, slope of negative section. Defaults to 1.0.
...	Base layer keyword arguments, such as name and dtype.

Value

The return value depends on the value provided for the first argument. If `object` is:

- a `keras_model_sequential()`, then the layer is added to the sequential model (which is modified in place). To enable piping, the sequential model is also returned, invisibly.
- a `keras_input()`, then the output tensor from calling `layer(input)` is returned.
- `NULL` or missing, then a `Layer` instance is returned.

See Also

- `https://keras.io/api/layers/activation_layers/elu#elu-class`

Other activation layers:
`layer_activation()`
`layer_activation_leaky_relu()`
`layer_activation_parametric_relu()`
`layer_activation_relu()`
`layer_activation_softmax()`

Other layers:
`Layer()`
`layer_activation()`
`layer_activation_leaky_relu()`
`layer_activation_parametric_relu()`
`layer_activation_relu()`
`layer_activation_softmax()`
`layer_activity_regularization()`
`layer_add()`
`layer_additive_attention()`
`layer_alpha_dropout()`
`layer_attention()`
`layer_average()`
`layer_average_pooling_1d()`
`layer_average_pooling_2d()`
`layer_average_pooling_3d()`
`layer_batch_normalization()`
`layer_bidirectional()`
`layer_category_encoding()`
`layer_center_crop()`
`layer_concatenate()`
`layer_conv_1d()`
`layer_conv_1d_transpose()`
`layer_conv_2d()`
`layer_conv_2d_transpose()`
`layer_conv_3d()`
`layer_conv_3d_transpose()`
`layer_conv_lstm_1d()`
`layer_conv_lstm_2d()`

layer_conv_lstm_3d()
layer_cropping_1d()
layer_cropping_2d()
layer_cropping_3d()
layer_dense()
layer_depthwise_conv_1d()
layer_depthwise_conv_2d()
layer_discretization()
layer_dot()
layer_dropout()
layer_einsum_dense()
layer_embedding()
layer_feature_space()
layer_flatten()
layer_flax_module_wrapper()
layer_gaussian_dropout()
layer_gaussian_noise()
layer_global_average_pooling_1d()
layer_global_average_pooling_2d()
layer_global_average_pooling_3d()
layer_global_max_pooling_1d()
layer_global_max_pooling_2d()
layer_global_max_pooling_3d()
layer_group_normalization()
layer_group_query_attention()
layer_gru()
layer_hashed_crossing()
layer_hashing()
layer_identity()
layer_integer_lookup()
layer_jax_model_wrapper()
layer_lambda()
layer_layer_normalization()
layer_lstm()
layer_masking()
layer_max_pooling_1d()
layer_max_pooling_2d()
layer_max_pooling_3d()
layer_maximum()
layer_mel_spectrogram()
layer_minimum()
layer_multi_head_attention()
layer_multiply()
layer_normalization()
layer_permute()
layer_random_brightness()
layer_random_contrast()
layer_random_crop()

layer_random_flip()
layer_random_rotation()
layer_random_translation()
layer_random_zoom()
layer_repeat_vector()
layer_rescaling()
layer_reshape()
layer_resizing()
layer_rnn()
layer_separable_conv_1d()
layer_separable_conv_2d()
layer_simple_rnn()
layer_spatial_dropout_1d()
layer_spatial_dropout_2d()
layer_spatial_dropout_3d()
layer_spectral_normalization()
layer_string_lookup()
layer_subtract()
layer_text_vectorization()
layer_tfsm()
layer_time_distributed()
layer_torch_module_wrapper()
layer_unit_normalization()
layer_upsampling_1d()
layer_upsampling_2d()
layer_upsampling_3d()
layer_zero_padding_1d()
layer_zero_padding_2d()
layer_zero_padding_3d()
rnn_cell_gru()
rnn_cell_lstm()
rnn_cell_simple()
rnn_cells_stack()

---

layer_activation_leaky_relu
Leaky version of a Rectified Linear Unit activation layer.

Description

This layer allows a small gradient when the unit is not active.

Formula:

```
f <- function(x) ifelse(x >= 0, x, alpha * x)
```

Usage

```
layer_activation_leaky_relu(object, negative_slope = 0.3, ...)
```

Arguments

`object`	Object to compose the layer with. A tensor, array, or sequential model.
`negative_slope`	Float >= 0.0. Negative slope coefficient. Defaults to `0.3`.
`...`	Base layer keyword arguments, such as `name` and `dtype`.

Value

The return value depends on the value provided for the first argument. If `object` is:

- a `keras_model_sequential()`, then the layer is added to the sequential model (which is modified in place). To enable piping, the sequential model is also returned, invisibly.
- a `keras_input()`, then the output tensor from calling `layer(input)` is returned.
- `NULL` or missing, then a `Layer` instance is returned.

Examples

```
leaky_relu_layer <- layer_activation_leaky_relu(negative_slope=0.5)
input <- array(c(-10, -5, 0.0, 5, 10))
result <- leaky_relu_layer(input)
as.array(result)

## [1] -5.0 -2.5  0.0  5.0 10.0
```

See Also

- `https://keras.io/api/layers/activation_layers/leaky_relu#leakyrelu-class`

Other activation layers:
`layer_activation()`
`layer_activation_elu()`
`layer_activation_parametric_relu()`
`layer_activation_relu()`
`layer_activation_softmax()`

Other layers:
`Layer()`
`layer_activation()`
`layer_activation_elu()`
`layer_activation_parametric_relu()`
`layer_activation_relu()`
`layer_activation_softmax()`
`layer_activity_regularization()`
`layer_add()`

layer_additive_attention()
layer_alpha_dropout()
layer_attention()
layer_average()
layer_average_pooling_1d()
layer_average_pooling_2d()
layer_average_pooling_3d()
layer_batch_normalization()
layer_bidirectional()
layer_category_encoding()
layer_center_crop()
layer_concatenate()
layer_conv_1d()
layer_conv_1d_transpose()
layer_conv_2d()
layer_conv_2d_transpose()
layer_conv_3d()
layer_conv_3d_transpose()
layer_conv_lstm_1d()
layer_conv_lstm_2d()
layer_conv_lstm_3d()
layer_cropping_1d()
layer_cropping_2d()
layer_cropping_3d()
layer_dense()
layer_depthwise_conv_1d()
layer_depthwise_conv_2d()
layer_discretization()
layer_dot()
layer_dropout()
layer_einsum_dense()
layer_embedding()
layer_feature_space()
layer_flatten()
layer_flax_module_wrapper()
layer_gaussian_dropout()
layer_gaussian_noise()
layer_global_average_pooling_1d()
layer_global_average_pooling_2d()
layer_global_average_pooling_3d()
layer_global_max_pooling_1d()
layer_global_max_pooling_2d()
layer_global_max_pooling_3d()
layer_group_normalization()
layer_group_query_attention()
layer_gru()
layer_hashed_crossing()
layer_hashing()

layer_identity()
layer_integer_lookup()
layer_jax_model_wrapper()
layer_lambda()
layer_layer_normalization()
layer_lstm()
layer_masking()
layer_max_pooling_1d()
layer_max_pooling_2d()
layer_max_pooling_3d()
layer_maximum()
layer_mel_spectrogram()
layer_minimum()
layer_multi_head_attention()
layer_multiply()
layer_normalization()
layer_permute()
layer_random_brightness()
layer_random_contrast()
layer_random_crop()
layer_random_flip()
layer_random_rotation()
layer_random_translation()
layer_random_zoom()
layer_repeat_vector()
layer_rescaling()
layer_reshape()
layer_resizing()
layer_rnn()
layer_separable_conv_1d()
layer_separable_conv_2d()
layer_simple_rnn()
layer_spatial_dropout_1d()
layer_spatial_dropout_2d()
layer_spatial_dropout_3d()
layer_spectral_normalization()
layer_string_lookup()
layer_subtract()
layer_text_vectorization()
layer_tfsm()
layer_time_distributed()
layer_torch_module_wrapper()
layer_unit_normalization()
layer_upsampling_1d()
layer_upsampling_2d()
layer_upsampling_3d()
layer_zero_padding_1d()
layer_zero_padding_2d()

layer_zero_padding_3d()
rnn_cell_gru()
rnn_cell_lstm()
rnn_cell_simple()
rnn_cells_stack()

---

layer_activation_parametric_relu
Parametric Rectified Linear Unit activation layer.

Description

Formula:

```
f <- function(x) ifelse(x >= 0, x, alpha * x)
```

where alpha is a learned array with the same shape as x.

Usage

```
layer_activation_parametric_relu(
  object,
  alpha_initializer = "Zeros",
  alpha_regularizer = NULL,
  alpha_constraint = NULL,
  shared_axes = NULL,
  ...
)
```

Arguments

object	Object to compose the layer with. A tensor, array, or sequential model.
alpha_initializer	Initializer function for the weights.
alpha_regularizer	Regularizer for the weights.
alpha_constraint	Constraint for the weights.
shared_axes	The axes along which to share learnable parameters for the activation function. For example, if the incoming feature maps are from a 2D convolution with output shape (batch, height, width, channels), and you wish to share parameters across space so that each filter only has one set of parameters, set shared_axes=[1, 2].
...	Base layer keyword arguments, such as name and dtype.

Value

The return value depends on the value provided for the first argument. If `object` is:

- a `keras_model_sequential()`, then the layer is added to the sequential model (which is modified in place). To enable piping, the sequential model is also returned, invisibly.
- a `keras_input()`, then the output tensor from calling `layer(input)` is returned.
- `NULL` or missing, then a `Layer` instance is returned.

See Also

- https://keras.io/api/layers/activation_layers/prelu#prelu-class

Other activation layers:
`layer_activation()`
`layer_activation_elu()`
`layer_activation_leaky_relu()`
`layer_activation_relu()`
`layer_activation_softmax()`

Other layers:
`Layer()`
`layer_activation()`
`layer_activation_elu()`
`layer_activation_leaky_relu()`
`layer_activation_relu()`
`layer_activation_softmax()`
`layer_activity_regularization()`
`layer_add()`
`layer_additive_attention()`
`layer_alpha_dropout()`
`layer_attention()`
`layer_average()`
`layer_average_pooling_1d()`
`layer_average_pooling_2d()`
`layer_average_pooling_3d()`
`layer_batch_normalization()`
`layer_bidirectional()`
`layer_category_encoding()`
`layer_center_crop()`
`layer_concatenate()`
`layer_conv_1d()`
`layer_conv_1d_transpose()`
`layer_conv_2d()`
`layer_conv_2d_transpose()`
`layer_conv_3d()`
`layer_conv_3d_transpose()`
`layer_conv_lstm_1d()`
`layer_conv_lstm_2d()`

layer_conv_lstm_3d()
layer_cropping_1d()
layer_cropping_2d()
layer_cropping_3d()
layer_dense()
layer_depthwise_conv_1d()
layer_depthwise_conv_2d()
layer_discretization()
layer_dot()
layer_dropout()
layer_einsum_dense()
layer_embedding()
layer_feature_space()
layer_flatten()
layer_flax_module_wrapper()
layer_gaussian_dropout()
layer_gaussian_noise()
layer_global_average_pooling_1d()
layer_global_average_pooling_2d()
layer_global_average_pooling_3d()
layer_global_max_pooling_1d()
layer_global_max_pooling_2d()
layer_global_max_pooling_3d()
layer_group_normalization()
layer_group_query_attention()
layer_gru()
layer_hashed_crossing()
layer_hashing()
layer_identity()
layer_integer_lookup()
layer_jax_model_wrapper()
layer_lambda()
layer_layer_normalization()
layer_lstm()
layer_masking()
layer_max_pooling_1d()
layer_max_pooling_2d()
layer_max_pooling_3d()
layer_maximum()
layer_mel_spectrogram()
layer_minimum()
layer_multi_head_attention()
layer_multiply()
layer_normalization()
layer_permute()
layer_random_brightness()
layer_random_contrast()
layer_random_crop()

layer_random_flip()
layer_random_rotation()
layer_random_translation()
layer_random_zoom()
layer_repeat_vector()
layer_rescaling()
layer_reshape()
layer_resizing()
layer_rnn()
layer_separable_conv_1d()
layer_separable_conv_2d()
layer_simple_rnn()
layer_spatial_dropout_1d()
layer_spatial_dropout_2d()
layer_spatial_dropout_3d()
layer_spectral_normalization()
layer_string_lookup()
layer_subtract()
layer_text_vectorization()
layer_tfsm()
layer_time_distributed()
layer_torch_module_wrapper()
layer_unit_normalization()
layer_upsampling_1d()
layer_upsampling_2d()
layer_upsampling_3d()
layer_zero_padding_1d()
layer_zero_padding_2d()
layer_zero_padding_3d()
rnn_cell_gru()
rnn_cell_lstm()
rnn_cell_simple()
rnn_cells_stack()

layer_activation_relu *Rectified Linear Unit activation function layer.*

Description

Formula:

```
f <- function(x, max_value = Inf, negative_slope = 0, threshold = 0) {
 x <- max(x,0)
 if (x >= max_value)
   max_value
 else if (threshold <= x && x < max_value)
```

```
    x
  else
    negative_slope * (x - threshold)
}
```

Usage

```
layer_activation_relu(
  object,
  max_value = NULL,
  negative_slope = 0,
  threshold = 0,
  ...
)
```

Arguments

`object`	Object to compose the layer with. A tensor, array, or sequential model.
`max_value`	Float >= 0. Maximum activation value. `NULL` means unlimited. Defaults to `NULL`.
`negative_slope`	Float >= 0. Negative slope coefficient. Defaults to `0.0`.
`threshold`	Float >= 0. Threshold value for thresholded activation. Defaults to `0.0`.
`...`	Base layer keyword arguments, such as `name` and `dtype`.

Value

The return value depends on the value provided for the first argument. If `object` is:

- a `keras_model_sequential()`, then the layer is added to the sequential model (which is modified in place). To enable piping, the sequential model is also returned, invisibly.
- a `keras_input()`, then the output tensor from calling `layer(input)` is returned.
- `NULL` or missing, then a `Layer` instance is returned.

Examples

```
relu_layer <- layer_activation_relu(max_value = 10,
                                    negative_slope = 0.5,
                                    threshold = 0)
input <- array(c(-10, -5, 0.0, 5, 10))
result <- relu_layer(input)
as.array(result)


## [1] -5.0 -2.5  0.0  5.0 10.0
```

See Also

- https://keras.io/api/layers/activation_layers/relu#relu-class

Other activation layers:
layer_activation()
layer_activation_elu()
layer_activation_leaky_relu()
layer_activation_parametric_relu()
layer_activation_softmax()

Other layers:
Layer()
layer_activation()
layer_activation_elu()
layer_activation_leaky_relu()
layer_activation_parametric_relu()
layer_activation_softmax()
layer_activity_regularization()
layer_add()
layer_additive_attention()
layer_alpha_dropout()
layer_attention()
layer_average()
layer_average_pooling_1d()
layer_average_pooling_2d()
layer_average_pooling_3d()
layer_batch_normalization()
layer_bidirectional()
layer_category_encoding()
layer_center_crop()
layer_concatenate()
layer_conv_1d()
layer_conv_1d_transpose()
layer_conv_2d()
layer_conv_2d_transpose()
layer_conv_3d()
layer_conv_3d_transpose()
layer_conv_lstm_1d()
layer_conv_lstm_2d()
layer_conv_lstm_3d()
layer_cropping_1d()
layer_cropping_2d()
layer_cropping_3d()
layer_dense()
layer_depthwise_conv_1d()
layer_depthwise_conv_2d()
layer_discretization()
layer_dot()

layer_dropout()
layer_einsum_dense()
layer_embedding()
layer_feature_space()
layer_flatten()
layer_flax_module_wrapper()
layer_gaussian_dropout()
layer_gaussian_noise()
layer_global_average_pooling_1d()
layer_global_average_pooling_2d()
layer_global_average_pooling_3d()
layer_global_max_pooling_1d()
layer_global_max_pooling_2d()
layer_global_max_pooling_3d()
layer_group_normalization()
layer_group_query_attention()
layer_gru()
layer_hashed_crossing()
layer_hashing()
layer_identity()
layer_integer_lookup()
layer_jax_model_wrapper()
layer_lambda()
layer_layer_normalization()
layer_lstm()
layer_masking()
layer_max_pooling_1d()
layer_max_pooling_2d()
layer_max_pooling_3d()
layer_maximum()
layer_mel_spectrogram()
layer_minimum()
layer_multi_head_attention()
layer_multiply()
layer_normalization()
layer_permute()
layer_random_brightness()
layer_random_contrast()
layer_random_crop()
layer_random_flip()
layer_random_rotation()
layer_random_translation()
layer_random_zoom()
layer_repeat_vector()
layer_rescaling()
layer_reshape()
layer_resizing()
layer_rnn()

layer_separable_conv_1d()
layer_separable_conv_2d()
layer_simple_rnn()
layer_spatial_dropout_1d()
layer_spatial_dropout_2d()
layer_spatial_dropout_3d()
layer_spectral_normalization()
layer_string_lookup()
layer_subtract()
layer_text_vectorization()
layer_tfsm()
layer_time_distributed()
layer_torch_module_wrapper()
layer_unit_normalization()
layer_upsampling_1d()
layer_upsampling_2d()
layer_upsampling_3d()
layer_zero_padding_1d()
layer_zero_padding_2d()
layer_zero_padding_3d()
rnn_cell_gru()
rnn_cell_lstm()
rnn_cell_simple()
rnn_cells_stack()

---

layer_activation_softmax

Softmax activation layer.

Description

Formula:

```
exp_x = exp(x - max(x))
f(x) = exp_x / sum(exp_x)
```

Usage

```
layer_activation_softmax(object, axis = -1L, ...)
```

Arguments

object	Object to compose the layer with. A tensor, array, or sequential model.
axis	Integer, or list of Integers, axis along which the softmax normalization is applied.
...	Base layer keyword arguments, such as name and dtype.

Value

Softmaxed output with the same shape as `inputs`.

Examples

```
softmax_layer <- layer_activation_softmax()
input <- op_array(c(1, 2, 1))
softmax_layer(input)

## tf.Tensor([0.21194157 0.5761169  0.21194157], shape=(3), dtype=float32)
```

Call Arguments

- `inputs`: The inputs (logits) to the softmax layer.
- `mask`: A boolean mask of the same shape as `inputs`. The mask specifies 1 to keep and 0 to mask. Defaults to `NULL`.

See Also

- https://keras.io/api/layers/activation_layers/softmax#softmax-class

Other activation layers:
`layer_activation()`
`layer_activation_elu()`
`layer_activation_leaky_relu()`
`layer_activation_parametric_relu()`
`layer_activation_relu()`

Other layers:
`Layer()`
`layer_activation()`
`layer_activation_elu()`
`layer_activation_leaky_relu()`
`layer_activation_parametric_relu()`
`layer_activation_relu()`
`layer_activity_regularization()`
`layer_add()`
`layer_additive_attention()`
`layer_alpha_dropout()`
`layer_attention()`
`layer_average()`
`layer_average_pooling_1d()`
`layer_average_pooling_2d()`
`layer_average_pooling_3d()`
`layer_batch_normalization()`
`layer_bidirectional()`
`layer_category_encoding()`
`layer_center_crop()`

layer_concatenate()
layer_conv_1d()
layer_conv_1d_transpose()
layer_conv_2d()
layer_conv_2d_transpose()
layer_conv_3d()
layer_conv_3d_transpose()
layer_conv_lstm_1d()
layer_conv_lstm_2d()
layer_conv_lstm_3d()
layer_cropping_1d()
layer_cropping_2d()
layer_cropping_3d()
layer_dense()
layer_depthwise_conv_1d()
layer_depthwise_conv_2d()
layer_discretization()
layer_dot()
layer_dropout()
layer_einsum_dense()
layer_embedding()
layer_feature_space()
layer_flatten()
layer_flax_module_wrapper()
layer_gaussian_dropout()
layer_gaussian_noise()
layer_global_average_pooling_1d()
layer_global_average_pooling_2d()
layer_global_average_pooling_3d()
layer_global_max_pooling_1d()
layer_global_max_pooling_2d()
layer_global_max_pooling_3d()
layer_group_normalization()
layer_group_query_attention()
layer_gru()
layer_hashed_crossing()
layer_hashing()
layer_identity()
layer_integer_lookup()
layer_jax_model_wrapper()
layer_lambda()
layer_layer_normalization()
layer_lstm()
layer_masking()
layer_max_pooling_1d()
layer_max_pooling_2d()
layer_max_pooling_3d()
layer_maximum()

layer_mel_spectrogram()
layer_minimum()
layer_multi_head_attention()
layer_multiply()
layer_normalization()
layer_permute()
layer_random_brightness()
layer_random_contrast()
layer_random_crop()
layer_random_flip()
layer_random_rotation()
layer_random_translation()
layer_random_zoom()
layer_repeat_vector()
layer_rescaling()
layer_reshape()
layer_resizing()
layer_rnn()
layer_separable_conv_1d()
layer_separable_conv_2d()
layer_simple_rnn()
layer_spatial_dropout_1d()
layer_spatial_dropout_2d()
layer_spatial_dropout_3d()
layer_spectral_normalization()
layer_string_lookup()
layer_subtract()
layer_text_vectorization()
layer_tfsm()
layer_time_distributed()
layer_torch_module_wrapper()
layer_unit_normalization()
layer_upsampling_1d()
layer_upsampling_2d()
layer_upsampling_3d()
layer_zero_padding_1d()
layer_zero_padding_2d()
layer_zero_padding_3d()
rnn_cell_gru()
rnn_cell_lstm()
rnn_cell_simple()
rnn_cells_stack()

layer_activity_regularization

Layer that applies an update to the cost function based input activity.

Description

Layer that applies an update to the cost function based input activity.

Usage

```
layer_activity_regularization(object, l1 = 0, l2 = 0, ...)
```

Arguments

`object`	Object to compose the layer with. A tensor, array, or sequential model.
`l1`	L1 regularization factor (positive float).
`l2`	L2 regularization factor (positive float).
`...`	For forward/backward compatability.

Value

The return value depends on the value provided for the first argument. If `object` is:

- a `keras_model_sequential()`, then the layer is added to the sequential model (which is modified in place). To enable piping, the sequential model is also returned, invisibly.
- a `keras_input()`, then the output tensor from calling `layer(input)` is returned.
- `NULL` or missing, then a `Layer` instance is returned.

Input Shape

Arbitrary. Use the keyword argument `input_shape` (tuple of integers, does not include the samples axis) when using this layer as the first layer in a model.

Output Shape

Same shape as input.

See Also

- `https://keras.io/api/layers/regularization_layers/activity_regularization#activityregularizatio`

Other regularization layers:
`layer_alpha_dropout()`
`layer_dropout()`
`layer_gaussian_dropout()`
`layer_gaussian_noise()`
`layer_spatial_dropout_1d()`
`layer_spatial_dropout_2d()`
`layer_spatial_dropout_3d()`

Other layers:
`Layer()`
`layer_activation()`

layer_activation_elu()
layer_activation_leaky_relu()
layer_activation_parametric_relu()
layer_activation_relu()
layer_activation_softmax()
layer_add()
layer_additive_attention()
layer_alpha_dropout()
layer_attention()
layer_average()
layer_average_pooling_1d()
layer_average_pooling_2d()
layer_average_pooling_3d()
layer_batch_normalization()
layer_bidirectional()
layer_category_encoding()
layer_center_crop()
layer_concatenate()
layer_conv_1d()
layer_conv_1d_transpose()
layer_conv_2d()
layer_conv_2d_transpose()
layer_conv_3d()
layer_conv_3d_transpose()
layer_conv_lstm_1d()
layer_conv_lstm_2d()
layer_conv_lstm_3d()
layer_cropping_1d()
layer_cropping_2d()
layer_cropping_3d()
layer_dense()
layer_depthwise_conv_1d()
layer_depthwise_conv_2d()
layer_discretization()
layer_dot()
layer_dropout()
layer_einsum_dense()
layer_embedding()
layer_feature_space()
layer_flatten()
layer_flax_module_wrapper()
layer_gaussian_dropout()
layer_gaussian_noise()
layer_global_average_pooling_1d()
layer_global_average_pooling_2d()
layer_global_average_pooling_3d()
layer_global_max_pooling_1d()
layer_global_max_pooling_2d()

layer_global_max_pooling_3d()
layer_group_normalization()
layer_group_query_attention()
layer_gru()
layer_hashed_crossing()
layer_hashing()
layer_identity()
layer_integer_lookup()
layer_jax_model_wrapper()
layer_lambda()
layer_layer_normalization()
layer_lstm()
layer_masking()
layer_max_pooling_1d()
layer_max_pooling_2d()
layer_max_pooling_3d()
layer_maximum()
layer_mel_spectrogram()
layer_minimum()
layer_multi_head_attention()
layer_multiply()
layer_normalization()
layer_permute()
layer_random_brightness()
layer_random_contrast()
layer_random_crop()
layer_random_flip()
layer_random_rotation()
layer_random_translation()
layer_random_zoom()
layer_repeat_vector()
layer_rescaling()
layer_reshape()
layer_resizing()
layer_rnn()
layer_separable_conv_1d()
layer_separable_conv_2d()
layer_simple_rnn()
layer_spatial_dropout_1d()
layer_spatial_dropout_2d()
layer_spatial_dropout_3d()
layer_spectral_normalization()
layer_string_lookup()
layer_subtract()
layer_text_vectorization()
layer_tfsm()
layer_time_distributed()
layer_torch_module_wrapper()

layer_unit_normalization()
layer_upsampling_1d()
layer_upsampling_2d()
layer_upsampling_3d()
layer_zero_padding_1d()
layer_zero_padding_2d()
layer_zero_padding_3d()
rnn_cell_gru()
rnn_cell_lstm()
rnn_cell_simple()
rnn_cells_stack()

layer_add *Performs elementwise addition operation.*

Description

It takes as input a list of tensors, all of the same shape, and returns a single tensor (also of the same shape).

Usage

```
layer_add(inputs, ...)
```

Arguments

inputs	layers to combine
...	For forward/backward compatability.

Value

The return value depends on the value provided for the first argument. If object is:

- a keras_model_sequential(), then the layer is added to the sequential model (which is modified in place). To enable piping, the sequential model is also returned, invisibly.
- a keras_input(), then the output tensor from calling layer(input) is returned.
- NULL or missing, then a Layer instance is returned.

Examples

```
input_shape <- c(1, 2, 3)
x1 <- op_ones(input_shape)
x2 <- op_ones(input_shape)
layer_add(x1, x2)
```

```
## tf.Tensor(
## [[[2. 2. 2.]
##   [2. 2. 2.]]], shape=(1, 2, 3), dtype=float32)
```

Usage in a Keras model:

```
input1 <- layer_input(shape = c(16))
x1 <- input1 |> layer_dense(8, activation = 'relu')

input2 <- layer_input(shape = c(32))
x2 <- input2 |> layer_dense(8, activation = 'relu')

# equivalent to `added = layer_add([x1, x2))`
added <- layer_add(x1, x2)
output <- added |> layer_dense(4)

model <- keras_model(inputs = c(input1, input2), outputs = output)
```

See Also

- https://keras.io/api/layers/merging_layers/add#add-class

Other merging layers:
layer_average()
layer_concatenate()
layer_dot()
layer_maximum()
layer_minimum()
layer_multiply()
layer_subtract()

Other layers:
Layer()
layer_activation()
layer_activation_elu()
layer_activation_leaky_relu()
layer_activation_parametric_relu()
layer_activation_relu()
layer_activation_softmax()
layer_activity_regularization()
layer_additive_attention()
layer_alpha_dropout()
layer_attention()
layer_average()
layer_average_pooling_1d()
layer_average_pooling_2d()
layer_average_pooling_3d()
layer_batch_normalization()

layer_bidirectional()
layer_category_encoding()
layer_center_crop()
layer_concatenate()
layer_conv_1d()
layer_conv_1d_transpose()
layer_conv_2d()
layer_conv_2d_transpose()
layer_conv_3d()
layer_conv_3d_transpose()
layer_conv_lstm_1d()
layer_conv_lstm_2d()
layer_conv_lstm_3d()
layer_cropping_1d()
layer_cropping_2d()
layer_cropping_3d()
layer_dense()
layer_depthwise_conv_1d()
layer_depthwise_conv_2d()
layer_discretization()
layer_dot()
layer_dropout()
layer_einsum_dense()
layer_embedding()
layer_feature_space()
layer_flatten()
layer_flax_module_wrapper()
layer_gaussian_dropout()
layer_gaussian_noise()
layer_global_average_pooling_1d()
layer_global_average_pooling_2d()
layer_global_average_pooling_3d()
layer_global_max_pooling_1d()
layer_global_max_pooling_2d()
layer_global_max_pooling_3d()
layer_group_normalization()
layer_group_query_attention()
layer_gru()
layer_hashed_crossing()
layer_hashing()
layer_identity()
layer_integer_lookup()
layer_jax_model_wrapper()
layer_lambda()
layer_layer_normalization()
layer_lstm()
layer_masking()
layer_max_pooling_1d()

layer_max_pooling_2d()
layer_max_pooling_3d()
layer_maximum()
layer_mel_spectrogram()
layer_minimum()
layer_multi_head_attention()
layer_multiply()
layer_normalization()
layer_permute()
layer_random_brightness()
layer_random_contrast()
layer_random_crop()
layer_random_flip()
layer_random_rotation()
layer_random_translation()
layer_random_zoom()
layer_repeat_vector()
layer_rescaling()
layer_reshape()
layer_resizing()
layer_rnn()
layer_separable_conv_1d()
layer_separable_conv_2d()
layer_simple_rnn()
layer_spatial_dropout_1d()
layer_spatial_dropout_2d()
layer_spatial_dropout_3d()
layer_spectral_normalization()
layer_string_lookup()
layer_subtract()
layer_text_vectorization()
layer_tfsm()
layer_time_distributed()
layer_torch_module_wrapper()
layer_unit_normalization()
layer_upsampling_1d()
layer_upsampling_2d()
layer_upsampling_3d()
layer_zero_padding_1d()
layer_zero_padding_2d()
layer_zero_padding_3d()
rnn_cell_gru()
rnn_cell_lstm()
rnn_cell_simple()
rnn_cells_stack()

layer_additive_attention

Additive attention layer, a.k.a. Bahdanau-style attention.

Description

Inputs are a list with 2 or 3 elements:

1. A `query` tensor of shape `(batch_size, Tq, dim)`.
2. A `value` tensor of shape `(batch_size, Tv, dim)`.
3. A optional `key` tensor of shape `(batch_size, Tv, dim)`. If none supplied, `value` will be used as `key`.

The calculation follows the steps:

1. Calculate attention scores using `query` and `key` with shape `(batch_size, Tq, Tv)` as a non-linear sum `scores = reduce_sum(tanh(query + key), axis=-1)`.
2. Use scores to calculate a softmax distribution with shape `(batch_size, Tq, Tv)`.
3. Use the softmax distribution to create a linear combination of `value` with shape `(batch_size, Tq, dim)`.

Usage

```
layer_additive_attention(object, use_scale = TRUE, dropout = 0, ...)
```

Arguments

`object`	Object to compose the layer with. A tensor, array, or sequential model.
`use_scale`	If `TRUE`, will create a scalar variable to scale the attention scores.
`dropout`	Float between 0 and 1. Fraction of the units to drop for the attention scores. Defaults to `0.0`.
`...`	For forward/backward compatability.

Value

The return value depends on the value provided for the first argument. If `object` is:

- a `keras_model_sequential()`, then the layer is added to the sequential model (which is modified in place). To enable piping, the sequential model is also returned, invisibly.
- a `keras_input()`, then the output tensor from calling `layer(input)` is returned.
- `NULL` or missing, then a `Layer` instance is returned.

Call Arguments

- `inputs`: List of the following tensors:
  - `query`: Query tensor of shape `(batch_size, Tq, dim)`.
  - `value`: Value tensor of shape `(batch_size, Tv, dim)`.
  - `key`: Optional key tensor of shape `(batch_size, Tv, dim)`. If not given, will use `value` for both `key` and `value`, which is the most common case.
- `mask`: List of the following tensors:
  - `query_mask`: A boolean mask tensor of shape `(batch_size, Tq)`. If given, the output will be zero at the positions where `mask==FALSE`.
  - `value_mask`: A boolean mask tensor of shape `(batch_size, Tv)`. If given, will apply the mask such that values at positions where `mask==FALSE` do not contribute to the result.
- `return_attention_scores`: bool, it `TRUE`, returns the attention scores (after masking and softmax) as an additional output argument.
- `training`: Python boolean indicating whether the layer should behave in training mode (adding dropout) or in inference mode (no dropout).
- `use_causal_mask`: Boolean. Set to `TRUE` for decoder self-attention. Adds a mask such that position `i` cannot attend to positions `j > i`. This prevents the flow of information from the future towards the past. Defaults to `FALSE`.

Output

Attention outputs of shape `(batch_size, Tq, dim)`. (Optional) Attention scores after masking and softmax with shape `(batch_size, Tq, Tv)`.

See Also

- `https://keras.io/api/layers/attention_layers/additive_attention#additiveattention-class`

Other attention layers:
`layer_attention()`
`layer_group_query_attention()`
`layer_multi_head_attention()`

Other layers:
`Layer()`
`layer_activation()`
`layer_activation_elu()`
`layer_activation_leaky_relu()`
`layer_activation_parametric_relu()`
`layer_activation_relu()`
`layer_activation_softmax()`
`layer_activity_regularization()`
`layer_add()`
`layer_alpha_dropout()`
`layer_attention()`
`layer_average()`

layer_average_pooling_1d()
layer_average_pooling_2d()
layer_average_pooling_3d()
layer_batch_normalization()
layer_bidirectional()
layer_category_encoding()
layer_center_crop()
layer_concatenate()
layer_conv_1d()
layer_conv_1d_transpose()
layer_conv_2d()
layer_conv_2d_transpose()
layer_conv_3d()
layer_conv_3d_transpose()
layer_conv_lstm_1d()
layer_conv_lstm_2d()
layer_conv_lstm_3d()
layer_cropping_1d()
layer_cropping_2d()
layer_cropping_3d()
layer_dense()
layer_depthwise_conv_1d()
layer_depthwise_conv_2d()
layer_discretization()
layer_dot()
layer_dropout()
layer_einsum_dense()
layer_embedding()
layer_feature_space()
layer_flatten()
layer_flax_module_wrapper()
layer_gaussian_dropout()
layer_gaussian_noise()
layer_global_average_pooling_1d()
layer_global_average_pooling_2d()
layer_global_average_pooling_3d()
layer_global_max_pooling_1d()
layer_global_max_pooling_2d()
layer_global_max_pooling_3d()
layer_group_normalization()
layer_group_query_attention()
layer_gru()
layer_hashed_crossing()
layer_hashing()
layer_identity()
layer_integer_lookup()
layer_jax_model_wrapper()
layer_lambda()

layer_layer_normalization()
layer_lstm()
layer_masking()
layer_max_pooling_1d()
layer_max_pooling_2d()
layer_max_pooling_3d()
layer_maximum()
layer_mel_spectrogram()
layer_minimum()
layer_multi_head_attention()
layer_multiply()
layer_normalization()
layer_permute()
layer_random_brightness()
layer_random_contrast()
layer_random_crop()
layer_random_flip()
layer_random_rotation()
layer_random_translation()
layer_random_zoom()
layer_repeat_vector()
layer_rescaling()
layer_reshape()
layer_resizing()
layer_rnn()
layer_separable_conv_1d()
layer_separable_conv_2d()
layer_simple_rnn()
layer_spatial_dropout_1d()
layer_spatial_dropout_2d()
layer_spatial_dropout_3d()
layer_spectral_normalization()
layer_string_lookup()
layer_subtract()
layer_text_vectorization()
layer_tfsm()
layer_time_distributed()
layer_torch_module_wrapper()
layer_unit_normalization()
layer_upsampling_1d()
layer_upsampling_2d()
layer_upsampling_3d()
layer_zero_padding_1d()
layer_zero_padding_2d()
layer_zero_padding_3d()
rnn_cell_gru()
rnn_cell_lstm()
rnn_cell_simple()

rnn_cells_stack()

---

layer_alpha_dropout *Applies Alpha Dropout to the input.*

Description

Alpha Dropout is a `Dropout` that keeps mean and variance of inputs to their original values, in order to ensure the self-normalizing property even after this dropout. Alpha Dropout fits well to Scaled Exponential Linear Units (SELU) by randomly setting activations to the negative saturation value.

Usage

```
layer_alpha_dropout(object, rate, noise_shape = NULL, seed = NULL, ...)
```

Arguments

`object`	Object to compose the layer with. A tensor, array, or sequential model.
`rate`	Float between 0 and 1. The multiplicative noise will have standard deviation `sqrt(rate / (1 - rate))`.
`noise_shape`	1D integer tensor representing the shape of the binary alpha dropout mask that will be multiplied with the input. For instance, if your inputs have shape `(batch_size, timesteps, feat` and you want the alpha dropout mask to be the same for all timesteps, you can use `noise_shape = (batch_size, 1, features)`.
`seed`	An integer to use as random seed.
`...`	For forward/backward compatability.

Value

The return value depends on the value provided for the first argument. If `object` is:

- a `keras_model_sequential()`, then the layer is added to the sequential model (which is modified in place). To enable piping, the sequential model is also returned, invisibly.
- a `keras_input()`, then the output tensor from calling `layer(input)` is returned.
- `NULL` or missing, then a `Layer` instance is returned.

Call Arguments

- `inputs`: Input tensor (of any rank).
- `training`: R boolean indicating whether the layer should behave in training mode (adding alpha dropout) or in inference mode (doing nothing).

See Also

- https://www.tensorflow.org/api_docs/python/tf/keras/layers/AlphaDropout

Other regularization layers:
layer_activity_regularization()
layer_dropout()
layer_gaussian_dropout()
layer_gaussian_noise()
layer_spatial_dropout_1d()
layer_spatial_dropout_2d()
layer_spatial_dropout_3d()

Other layers:
Layer()
layer_activation()
layer_activation_elu()
layer_activation_leaky_relu()
layer_activation_parametric_relu()
layer_activation_relu()
layer_activation_softmax()
layer_activity_regularization()
layer_add()
layer_additive_attention()
layer_attention()
layer_average()
layer_average_pooling_1d()
layer_average_pooling_2d()
layer_average_pooling_3d()
layer_batch_normalization()
layer_bidirectional()
layer_category_encoding()
layer_center_crop()
layer_concatenate()
layer_conv_1d()
layer_conv_1d_transpose()
layer_conv_2d()
layer_conv_2d_transpose()
layer_conv_3d()
layer_conv_3d_transpose()
layer_conv_lstm_1d()
layer_conv_lstm_2d()
layer_conv_lstm_3d()
layer_cropping_1d()
layer_cropping_2d()
layer_cropping_3d()
layer_dense()
layer_depthwise_conv_1d()
layer_depthwise_conv_2d()

layer_discretization()
layer_dot()
layer_dropout()
layer_einsum_dense()
layer_embedding()
layer_feature_space()
layer_flatten()
layer_flax_module_wrapper()
layer_gaussian_dropout()
layer_gaussian_noise()
layer_global_average_pooling_1d()
layer_global_average_pooling_2d()
layer_global_average_pooling_3d()
layer_global_max_pooling_1d()
layer_global_max_pooling_2d()
layer_global_max_pooling_3d()
layer_group_normalization()
layer_group_query_attention()
layer_gru()
layer_hashed_crossing()
layer_hashing()
layer_identity()
layer_integer_lookup()
layer_jax_model_wrapper()
layer_lambda()
layer_layer_normalization()
layer_lstm()
layer_masking()
layer_max_pooling_1d()
layer_max_pooling_2d()
layer_max_pooling_3d()
layer_maximum()
layer_mel_spectrogram()
layer_minimum()
layer_multi_head_attention()
layer_multiply()
layer_normalization()
layer_permute()
layer_random_brightness()
layer_random_contrast()
layer_random_crop()
layer_random_flip()
layer_random_rotation()
layer_random_translation()
layer_random_zoom()
layer_repeat_vector()
layer_rescaling()
layer_reshape()

layer_resizing()
layer_rnn()
layer_separable_conv_1d()
layer_separable_conv_2d()
layer_simple_rnn()
layer_spatial_dropout_1d()
layer_spatial_dropout_2d()
layer_spatial_dropout_3d()
layer_spectral_normalization()
layer_string_lookup()
layer_subtract()
layer_text_vectorization()
layer_tfsm()
layer_time_distributed()
layer_torch_module_wrapper()
layer_unit_normalization()
layer_upsampling_1d()
layer_upsampling_2d()
layer_upsampling_3d()
layer_zero_padding_1d()
layer_zero_padding_2d()
layer_zero_padding_3d()
rnn_cell_gru()
rnn_cell_lstm()
rnn_cell_simple()
rnn_cells_stack()

layer_attention *Dot-product attention layer, a.k.a. Luong-style attention.*

Description

Inputs are a list with 2 or 3 elements:

1. A query tensor of shape (batch_size, Tq, dim).
2. A value tensor of shape (batch_size, Tv, dim).
3. A optional key tensor of shape (batch_size, Tv, dim). If none supplied, value will be used as a key.

The calculation follows the steps:

1. Calculate attention scores using query and key with shape (batch_size, Tq, Tv).
2. Use scores to calculate a softmax distribution with shape (batch_size, Tq, Tv).
3. Use the softmax distribution to create a linear combination of value with shape (batch_size, Tq, dim).

Usage

```
layer_attention(
  object,
  use_scale = FALSE,
  score_mode = "dot",
  dropout = 0,
  seed = NULL,
  ...
)
```

Arguments

`object`	Object to compose the layer with. A tensor, array, or sequential model.
`use_scale`	If `TRUE`, will create a scalar variable to scale the attention scores.
`score_mode`	Function to use to compute attention scores, one of `{"dot", "concat"}`. `"dot"` refers to the dot product between the query and key vectors. `"concat"` refers to the hyperbolic tangent of the concatenation of the `query` and `key` vectors.
`dropout`	Float between 0 and 1. Fraction of the units to drop for the attention scores. Defaults to `0.0`.
`seed`	An integer to use as random seed incase of `dropout`.
`...`	For forward/backward compatability.

Value

The return value depends on the value provided for the first argument. If `object` is:

- a `keras_model_sequential()`, then the layer is added to the sequential model (which is modified in place). To enable piping, the sequential model is also returned, invisibly.
- a `keras_input()`, then the output tensor from calling `layer(input)` is returned.
- `NULL` or missing, then a `Layer` instance is returned.

Call Arguments

- `inputs`: List of the following tensors:
  - `query`: Query tensor of shape `(batch_size, Tq, dim)`.
  - `value`: Value tensor of shape `(batch_size, Tv, dim)`.
  - `key`: Optional key tensor of shape `(batch_size, Tv, dim)`. If not given, will use `value` for both `key` and `value`, which is the most common case.
- `mask`: List of the following tensors:
  - `query_mask`: A boolean mask tensor of shape `(batch_size, Tq)`. If given, the output will be zero at the positions where `mask==FALSE`.
  - `value_mask`: A boolean mask tensor of shape `(batch_size, Tv)`. If given, will apply the mask such that values at positions where `mask==FALSE` do not contribute to the result.
- `return_attention_scores`: bool, it `TRUE`, returns the attention scores (after masking and softmax) as an additional output argument.

- `training`: Python boolean indicating whether the layer should behave in training mode (adding dropout) or in inference mode (no dropout).
- `use_causal_mask`: Boolean. Set to `TRUE` for decoder self-attention. Adds a mask such that position `i` cannot attend to positions `j` > `i`. This prevents the flow of information from the future towards the past. Defaults to `FALSE`.

Output

Attention outputs of shape `(batch_size, Tq, dim)`. (Optional) Attention scores after masking and softmax with shape `(batch_size, Tq, Tv)`.

See Also

- https://keras.io/api/layers/attention_layers/attention#attention-class

Other attention layers:
`layer_additive_attention()`
`layer_group_query_attention()`
`layer_multi_head_attention()`

Other layers:
`Layer()`
`layer_activation()`
`layer_activation_elu()`
`layer_activation_leaky_relu()`
`layer_activation_parametric_relu()`
`layer_activation_relu()`
`layer_activation_softmax()`
`layer_activity_regularization()`
`layer_add()`
`layer_additive_attention()`
`layer_alpha_dropout()`
`layer_average()`
`layer_average_pooling_1d()`
`layer_average_pooling_2d()`
`layer_average_pooling_3d()`
`layer_batch_normalization()`
`layer_bidirectional()`
`layer_category_encoding()`
`layer_center_crop()`
`layer_concatenate()`
`layer_conv_1d()`
`layer_conv_1d_transpose()`
`layer_conv_2d()`
`layer_conv_2d_transpose()`
`layer_conv_3d()`
`layer_conv_3d_transpose()`
`layer_conv_lstm_1d()`
`layer_conv_lstm_2d()`

layer_conv_lstm_3d()
layer_cropping_1d()
layer_cropping_2d()
layer_cropping_3d()
layer_dense()
layer_depthwise_conv_1d()
layer_depthwise_conv_2d()
layer_discretization()
layer_dot()
layer_dropout()
layer_einsum_dense()
layer_embedding()
layer_feature_space()
layer_flatten()
layer_flax_module_wrapper()
layer_gaussian_dropout()
layer_gaussian_noise()
layer_global_average_pooling_1d()
layer_global_average_pooling_2d()
layer_global_average_pooling_3d()
layer_global_max_pooling_1d()
layer_global_max_pooling_2d()
layer_global_max_pooling_3d()
layer_group_normalization()
layer_group_query_attention()
layer_gru()
layer_hashed_crossing()
layer_hashing()
layer_identity()
layer_integer_lookup()
layer_jax_model_wrapper()
layer_lambda()
layer_layer_normalization()
layer_lstm()
layer_masking()
layer_max_pooling_1d()
layer_max_pooling_2d()
layer_max_pooling_3d()
layer_maximum()
layer_mel_spectrogram()
layer_minimum()
layer_multi_head_attention()
layer_multiply()
layer_normalization()
layer_permute()
layer_random_brightness()
layer_random_contrast()
layer_random_crop()

layer_random_flip()
layer_random_rotation()
layer_random_translation()
layer_random_zoom()
layer_repeat_vector()
layer_rescaling()
layer_reshape()
layer_resizing()
layer_rnn()
layer_separable_conv_1d()
layer_separable_conv_2d()
layer_simple_rnn()
layer_spatial_dropout_1d()
layer_spatial_dropout_2d()
layer_spatial_dropout_3d()
layer_spectral_normalization()
layer_string_lookup()
layer_subtract()
layer_text_vectorization()
layer_tfsm()
layer_time_distributed()
layer_torch_module_wrapper()
layer_unit_normalization()
layer_upsampling_1d()
layer_upsampling_2d()
layer_upsampling_3d()
layer_zero_padding_1d()
layer_zero_padding_2d()
layer_zero_padding_3d()
rnn_cell_gru()
rnn_cell_lstm()
rnn_cell_simple()
rnn_cells_stack()

layer_average *Averages a list of inputs element-wise..*

Description

It takes as input a list of tensors, all of the same shape, and returns a single tensor (also of the same shape).

Usage

```
layer_average(inputs, ...)
```

Arguments

`inputs`	layers to combine
`...`	For forward/backward compatability.

Value

The return value depends on the value provided for the first argument. If `object` is:

- a `keras_model_sequential()`, then the layer is added to the sequential model (which is modified in place). To enable piping, the sequential model is also returned, invisibly.
- a `keras_input()`, then the output tensor from calling `layer(input)` is returned.
- `NULL` or missing, then a `Layer` instance is returned.

Examples

```
input_shape <- c(1, 2, 3)
x1 <- op_ones(input_shape)
x2 <- op_zeros(input_shape)
layer_average(x1, x2)
```

```
## tf.Tensor(
## [[[0.5 0.5 0.5]
##   [0.5 0.5 0.5]]], shape=(1, 2, 3), dtype=float32)
```

Usage in a Keras model:

```
input1 <- layer_input(shape = c(16))
x1 <- input1 |> layer_dense(8, activation = 'relu')

input2 <- layer_input(shape = c(32))
x2 <- input2 |> layer_dense(8, activation = 'relu')

added <- layer_average(x1, x2)
output <- added |> layer_dense(4)

model <- keras_model(inputs = c(input1, input2), outputs = output)
```

See Also

- `https://keras.io/api/layers/merging_layers/average#average-class`

Other merging layers:
`layer_add()`
`layer_concatenate()`
`layer_dot()`
`layer_maximum()`
`layer_minimum()`

layer_multiply()
layer_subtract()

Other layers:
Layer()
layer_activation()
layer_activation_elu()
layer_activation_leaky_relu()
layer_activation_parametric_relu()
layer_activation_relu()
layer_activation_softmax()
layer_activity_regularization()
layer_add()
layer_additive_attention()
layer_alpha_dropout()
layer_attention()
layer_average_pooling_1d()
layer_average_pooling_2d()
layer_average_pooling_3d()
layer_batch_normalization()
layer_bidirectional()
layer_category_encoding()
layer_center_crop()
layer_concatenate()
layer_conv_1d()
layer_conv_1d_transpose()
layer_conv_2d()
layer_conv_2d_transpose()
layer_conv_3d()
layer_conv_3d_transpose()
layer_conv_lstm_1d()
layer_conv_lstm_2d()
layer_conv_lstm_3d()
layer_cropping_1d()
layer_cropping_2d()
layer_cropping_3d()
layer_dense()
layer_depthwise_conv_1d()
layer_depthwise_conv_2d()
layer_discretization()
layer_dot()
layer_dropout()
layer_einsum_dense()
layer_embedding()
layer_feature_space()
layer_flatten()
layer_flax_module_wrapper()
layer_gaussian_dropout()

layer_gaussian_noise()
layer_global_average_pooling_1d()
layer_global_average_pooling_2d()
layer_global_average_pooling_3d()
layer_global_max_pooling_1d()
layer_global_max_pooling_2d()
layer_global_max_pooling_3d()
layer_group_normalization()
layer_group_query_attention()
layer_gru()
layer_hashed_crossing()
layer_hashing()
layer_identity()
layer_integer_lookup()
layer_jax_model_wrapper()
layer_lambda()
layer_layer_normalization()
layer_lstm()
layer_masking()
layer_max_pooling_1d()
layer_max_pooling_2d()
layer_max_pooling_3d()
layer_maximum()
layer_mel_spectrogram()
layer_minimum()
layer_multi_head_attention()
layer_multiply()
layer_normalization()
layer_permute()
layer_random_brightness()
layer_random_contrast()
layer_random_crop()
layer_random_flip()
layer_random_rotation()
layer_random_translation()
layer_random_zoom()
layer_repeat_vector()
layer_rescaling()
layer_reshape()
layer_resizing()
layer_rnn()
layer_separable_conv_1d()
layer_separable_conv_2d()
layer_simple_rnn()
layer_spatial_dropout_1d()
layer_spatial_dropout_2d()
layer_spatial_dropout_3d()
layer_spectral_normalization()

layer_string_lookup()
layer_subtract()
layer_text_vectorization()
layer_tfsm()
layer_time_distributed()
layer_torch_module_wrapper()
layer_unit_normalization()
layer_upsampling_1d()
layer_upsampling_2d()
layer_upsampling_3d()
layer_zero_padding_1d()
layer_zero_padding_2d()
layer_zero_padding_3d()
rnn_cell_gru()
rnn_cell_lstm()
rnn_cell_simple()
rnn_cells_stack()

---

layer_average_pooling_1d

Average pooling for temporal data.

Description

Downsamples the input representation by taking the average value over the window defined by pool_size. The window is shifted by strides. The resulting output when using "valid" padding option has a shape of: output_shape = (input_shape - pool_size + 1) / strides)

The resulting output shape when using the "same" padding option is: output_shape = input_shape / strides

Usage

```
layer_average_pooling_1d(
  object,
  pool_size,
  strides = NULL,
  padding = "valid",
  data_format = NULL,
  name = NULL,
  ...
)
```

Arguments

object	Object to compose the layer with. A tensor, array, or sequential model.
pool_size	int, size of the max pooling window.

strides	int or NULL. Specifies how much the pooling window moves for each pooling step. If NULL, it will default to pool_size.
padding	string, either "valid" or "same" (case-insensitive). "valid" means no padding. "same" results in padding evenly to the left/right or up/down of the input such that output has the same height/width dimension as the input.
data_format	string, either "channels_last" or "channels_first". The ordering of the dimensions in the inputs. "channels_last" corresponds to inputs with shape (batch, steps, features) while "channels_first" corresponds to inputs with shape (batch, features, steps). It defaults to the image_data_format value found in your Keras config file at ~/.keras/keras.json. If you never set it, then it will be "channels_last".
name	String, name for the object
...	For forward/backward compatability.

Value

The return value depends on the value provided for the first argument. If object is:

- a keras_model_sequential(), then the layer is added to the sequential model (which is modified in place). To enable piping, the sequential model is also returned, invisibly.
- a keras_input(), then the output tensor from calling layer(input) is returned.
- NULL or missing, then a Layer instance is returned.

Input Shape

- If data_format="channels_last": 3D tensor with shape (batch_size, steps, features).
- If data_format="channels_first": 3D tensor with shape (batch_size, features, steps).

Output Shape

- If data_format="channels_last": 3D tensor with shape (batch_size, downsampled_steps, features).
- If data_format="channels_first": 3D tensor with shape (batch_size, features, downsampled_steps).

Examples

strides=1 and padding="valid":

```
x <- op_array(c(1., 2., 3., 4., 5.)) |> op_reshape(c(1, 5, 1))
output <- x |>
  layer_average_pooling_1d(pool_size = 2,
                           strides = 1,
                           padding = "valid")
output

## tf.Tensor(
## [[[1.5]
##   [2.5]
##   [3.5]
##   [4.5]]], shape=(1, 4, 1), dtype=float32)
```

strides=2 and padding="valid":

```
x <- op_array(c(1., 2., 3., 4., 5.)) |> op_reshape(c(1, 5, 1))
output <- x |>
  layer_average_pooling_1d(pool_size = 2,
                           strides = 2,
                           padding = "valid")
output
```

```
## tf.Tensor(
## [[[1.5]
##   [3.5]]], shape=(1, 2, 1), dtype=float32)
```

strides=1 and padding="same":

```
x <- op_array(c(1., 2., 3., 4., 5.)) |> op_reshape(c(1, 5, 1))
output <- x |>
  layer_average_pooling_1d(pool_size = 2,
                           strides = 1,
                           padding = "same")
output
```

```
## tf.Tensor(
## [[[1.5]
##   [2.5]
##   [3.5]
##   [4.5]
##   [5. ]]], shape=(1, 5, 1), dtype=float32)
```

See Also

- https://keras.io/api/layers/pooling_layers/average_pooling1d#averagepooling1d-class

Other pooling layers:
layer_average_pooling_2d()
layer_average_pooling_3d()
layer_global_average_pooling_1d()
layer_global_average_pooling_2d()
layer_global_average_pooling_3d()
layer_global_max_pooling_1d()
layer_global_max_pooling_2d()
layer_global_max_pooling_3d()
layer_max_pooling_1d()

layer_max_pooling_2d()
layer_max_pooling_3d()

Other layers:
Layer()
layer_activation()
layer_activation_elu()
layer_activation_leaky_relu()
layer_activation_parametric_relu()
layer_activation_relu()
layer_activation_softmax()
layer_activity_regularization()
layer_add()
layer_additive_attention()
layer_alpha_dropout()
layer_attention()
layer_average()
layer_average_pooling_2d()
layer_average_pooling_3d()
layer_batch_normalization()
layer_bidirectional()
layer_category_encoding()
layer_center_crop()
layer_concatenate()
layer_conv_1d()
layer_conv_1d_transpose()
layer_conv_2d()
layer_conv_2d_transpose()
layer_conv_3d()
layer_conv_3d_transpose()
layer_conv_lstm_1d()
layer_conv_lstm_2d()
layer_conv_lstm_3d()
layer_cropping_1d()
layer_cropping_2d()
layer_cropping_3d()
layer_dense()
layer_depthwise_conv_1d()
layer_depthwise_conv_2d()
layer_discretization()
layer_dot()
layer_dropout()
layer_einsum_dense()
layer_embedding()
layer_feature_space()
layer_flatten()
layer_flax_module_wrapper()
layer_gaussian_dropout()

layer_gaussian_noise()
layer_global_average_pooling_1d()
layer_global_average_pooling_2d()
layer_global_average_pooling_3d()
layer_global_max_pooling_1d()
layer_global_max_pooling_2d()
layer_global_max_pooling_3d()
layer_group_normalization()
layer_group_query_attention()
layer_gru()
layer_hashed_crossing()
layer_hashing()
layer_identity()
layer_integer_lookup()
layer_jax_model_wrapper()
layer_lambda()
layer_layer_normalization()
layer_lstm()
layer_masking()
layer_max_pooling_1d()
layer_max_pooling_2d()
layer_max_pooling_3d()
layer_maximum()
layer_mel_spectrogram()
layer_minimum()
layer_multi_head_attention()
layer_multiply()
layer_normalization()
layer_permute()
layer_random_brightness()
layer_random_contrast()
layer_random_crop()
layer_random_flip()
layer_random_rotation()
layer_random_translation()
layer_random_zoom()
layer_repeat_vector()
layer_rescaling()
layer_reshape()
layer_resizing()
layer_rnn()
layer_separable_conv_1d()
layer_separable_conv_2d()
layer_simple_rnn()
layer_spatial_dropout_1d()
layer_spatial_dropout_2d()
layer_spatial_dropout_3d()
layer_spectral_normalization()

layer_string_lookup()
layer_subtract()
layer_text_vectorization()
layer_tfsm()
layer_time_distributed()
layer_torch_module_wrapper()
layer_unit_normalization()
layer_upsampling_1d()
layer_upsampling_2d()
layer_upsampling_3d()
layer_zero_padding_1d()
layer_zero_padding_2d()
layer_zero_padding_3d()
rnn_cell_gru()
rnn_cell_lstm()
rnn_cell_simple()
rnn_cells_stack()

---

layer_average_pooling_2d

Average pooling operation for 2D spatial data.

Description

Downsamples the input along its spatial dimensions (height and width) by taking the average value over an input window (of size defined by `pool_size`) for each channel of the input. The window is shifted by `strides` along each dimension.

The resulting output when using the `"valid"` padding option has a spatial shape (number of rows or columns) of: `output_shape = math.floor((input_shape - pool_size) / strides) + 1` (when `input_shape >= pool_size`)

The resulting output shape when using the `"same"` padding option is: `output_shape = math.floor((input_shape - 1) / strides) + 1`

Usage

```
layer_average_pooling_2d(
  object,
  pool_size,
  strides = NULL,
  padding = "valid",
  data_format = NULL,
  name = NULL,
  ...
)
```

Arguments

`object`	Object to compose the layer with. A tensor, array, or sequential model.
`pool_size`	int or list of 2 integers, factors by which to downscale (dim1, dim2). If only one integer is specified, the same window length will be used for all dimensions.
`strides`	int or list of 2 integers, or `NULL`. Strides values. If `NULL`, it will default to `pool_size`. If only one int is specified, the same stride size will be used for all dimensions.
`padding`	string, either `"valid"` or `"same"` (case-insensitive). `"valid"` means no padding. `"same"` results in padding evenly to the left/right or up/down of the input such that output has the same height/width dimension as the input.
`data_format`	string, either `"channels_last"` or `"channels_first"`. The ordering of the dimensions in the inputs. `"channels_last"` corresponds to inputs with shape `(batch, height, width, channels)` while `"channels_first"` corresponds to inputs with shape `(batch, channels, height, width)`. It defaults to the `image_data_format` value found in your Keras config file at `~/.keras/keras.json`. If you never set it, then it will be `"channels_last"`.
`name`	String, name for the object
`...`	For forward/backward compatability.

Value

The return value depends on the value provided for the first argument. If `object` is:

- a `keras_model_sequential()`, then the layer is added to the sequential model (which is modified in place). To enable piping, the sequential model is also returned, invisibly.
- a `keras_input()`, then the output tensor from calling `layer(input)` is returned.
- `NULL` or missing, then a `Layer` instance is returned.

Input Shape

- If `data_format="channels_last"`: 4D tensor with shape `(batch_size, height, width, channels)`.
- If `data_format="channels_first"`: 4D tensor with shape `(batch_size, channels, height, width)`.

Output Shape

- If `data_format="channels_last"`: 4D tensor with shape `(batch_size, pooled_height, pooled_width, channel`
- If `data_format="channels_first"`: 4D tensor with shape `(batch_size, channels, pooled_height, pooled_wid`

Examples

`strides=(1, 1)` and `padding="valid"`:

```
x <- op_array(1:9, "float32") |> op_reshape(c(1, 3, 3, 1))
output <- x |>
  layer_average_pooling_2d(pool_size = c(2, 2),
                           strides = c(1, 1),
                           padding = "valid")
output
```

```
## tf.Tensor(
## [[[[3.]
##    [4.]]
##
##   [[6.]
##    [7.]]]], shape=(1, 2, 2, 1), dtype=float32)
```

strides=(2, 2) and padding="valid":

```
x <- op_array(1:12, "float32") |> op_reshape(c(1, 3, 4, 1))
output <- x |>
  layer_average_pooling_2d(pool_size = c(2, 2),
                           strides = c(2, 2),
                           padding = "valid")
output
```

```
## tf.Tensor(
## [[[[3.5]
##    [5.5]]]], shape=(1, 1, 2, 1), dtype=float32)
```

stride=(1, 1) and padding="same":

```
x <- op_array(1:9, "float32") |> op_reshape(c(1, 3, 3, 1))
output <- x |>
  layer_average_pooling_2d(pool_size = c(2, 2),
                           strides = c(1, 1),
                           padding = "same")
output
```

```
## tf.Tensor(
## [[[[3. ]
##    [4. ]
##    [4.5]]
##
##   [[6. ]
##    [7. ]
##    [7.5]]
##
##   [[7.5]
##    [8.5]
##    [9. ]]]], shape=(1, 3, 3, 1), dtype=float32)
```

See Also

- https://keras.io/api/layers/pooling_layers/average_pooling2d#averagepooling2d-class

Other pooling layers:
layer_average_pooling_1d()
layer_average_pooling_3d()
layer_global_average_pooling_1d()
layer_global_average_pooling_2d()
layer_global_average_pooling_3d()
layer_global_max_pooling_1d()
layer_global_max_pooling_2d()
layer_global_max_pooling_3d()
layer_max_pooling_1d()
layer_max_pooling_2d()
layer_max_pooling_3d()

Other layers:
Layer()
layer_activation()
layer_activation_elu()
layer_activation_leaky_relu()
layer_activation_parametric_relu()
layer_activation_relu()
layer_activation_softmax()
layer_activity_regularization()
layer_add()
layer_additive_attention()
layer_alpha_dropout()
layer_attention()
layer_average()
layer_average_pooling_1d()
layer_average_pooling_3d()
layer_batch_normalization()
layer_bidirectional()
layer_category_encoding()
layer_center_crop()
layer_concatenate()
layer_conv_1d()
layer_conv_1d_transpose()
layer_conv_2d()
layer_conv_2d_transpose()
layer_conv_3d()
layer_conv_3d_transpose()
layer_conv_lstm_1d()
layer_conv_lstm_2d()
layer_conv_lstm_3d()
layer_cropping_1d()
layer_cropping_2d()
layer_cropping_3d()
layer_dense()
layer_depthwise_conv_1d()

layer_depthwise_conv_2d()
layer_discretization()
layer_dot()
layer_dropout()
layer_einsum_dense()
layer_embedding()
layer_feature_space()
layer_flatten()
layer_flax_module_wrapper()
layer_gaussian_dropout()
layer_gaussian_noise()
layer_global_average_pooling_1d()
layer_global_average_pooling_2d()
layer_global_average_pooling_3d()
layer_global_max_pooling_1d()
layer_global_max_pooling_2d()
layer_global_max_pooling_3d()
layer_group_normalization()
layer_group_query_attention()
layer_gru()
layer_hashed_crossing()
layer_hashing()
layer_identity()
layer_integer_lookup()
layer_jax_model_wrapper()
layer_lambda()
layer_layer_normalization()
layer_lstm()
layer_masking()
layer_max_pooling_1d()
layer_max_pooling_2d()
layer_max_pooling_3d()
layer_maximum()
layer_mel_spectrogram()
layer_minimum()
layer_multi_head_attention()
layer_multiply()
layer_normalization()
layer_permute()
layer_random_brightness()
layer_random_contrast()
layer_random_crop()
layer_random_flip()
layer_random_rotation()
layer_random_translation()
layer_random_zoom()
layer_repeat_vector()
layer_rescaling()

layer_reshape()
layer_resizing()
layer_rnn()
layer_separable_conv_1d()
layer_separable_conv_2d()
layer_simple_rnn()
layer_spatial_dropout_1d()
layer_spatial_dropout_2d()
layer_spatial_dropout_3d()
layer_spectral_normalization()
layer_string_lookup()
layer_subtract()
layer_text_vectorization()
layer_tfsm()
layer_time_distributed()
layer_torch_module_wrapper()
layer_unit_normalization()
layer_upsampling_1d()
layer_upsampling_2d()
layer_upsampling_3d()
layer_zero_padding_1d()
layer_zero_padding_2d()
layer_zero_padding_3d()
rnn_cell_gru()
rnn_cell_lstm()
rnn_cell_simple()
rnn_cells_stack()

layer_average_pooling_3d

Average pooling operation for 3D data (spatial or spatio-temporal).

Description

Downsamples the input along its spatial dimensions (depth, height, and width) by taking the average value over an input window (of size defined by `pool_size`) for each channel of the input. The window is shifted by `strides` along each dimension.

Usage

```
layer_average_pooling_3d(
  object,
  pool_size,
  strides = NULL,
  padding = "valid",
  data_format = NULL,
  name = NULL,
  ...
)
```

Arguments

`object`	Object to compose the layer with. A tensor, array, or sequential model.
`pool_size`	int or list of 3 integers, factors by which to downscale (dim1, dim2, dim3). If only one integer is specified, the same window length will be used for all dimensions.
`strides`	int or list of 3 integers, or `NULL`. Strides values. If `NULL`, it will default to `pool_size`. If only one int is specified, the same stride size will be used for all dimensions.
`padding`	string, either `"valid"` or `"same"` (case-insensitive). `"valid"` means no padding. `"same"` results in padding evenly to the left/right or up/down of the input such that output has the same height/width dimension as the input.
`data_format`	string, either `"channels_last"` or `"channels_first"`. The ordering of the dimensions in the inputs. `"channels_last"` corresponds to inputs with shape `(batch, spatial_dim1, spatial_dim2, spatial_dim3, channels)` while `"channels_first"` corresponds to inputs with shape `(batch, channels, spatial_dim1, spatial_di` It defaults to the `image_data_format` value found in your Keras config file at `~/.keras/keras.json`. If you never set it, then it will be `"channels_last"`.
`name`	String, name for the object
`...`	For forward/backward compatability.

Value

The return value depends on the value provided for the first argument. If `object` is:

- a `keras_model_sequential()`, then the layer is added to the sequential model (which is modified in place). To enable piping, the sequential model is also returned, invisibly.
- a `keras_input()`, then the output tensor from calling `layer(input)` is returned.
- `NULL` or missing, then a `Layer` instance is returned.

Input Shape

- If `data_format="channels_last"`: 5D tensor with shape: `(batch_size, spatial_dim1, spatial_dim2, spatial_`
- If `data_format="channels_first"`: 5D tensor with shape: `(batch_size, channels, spatial_dim1, spatial_di`

Output Shape

- If `data_format="channels_last"`: 5D tensor with shape: `(batch_size, pooled_dim1, pooled_dim2, pooled_di`
- If `data_format="channels_first"`: 5D tensor with shape: `(batch_size, channels, pooled_dim1, pooled_dim2`

Examples

```
depth <- height <- width <- 30
channels <- 3

inputs <- layer_input(shape = c(depth, height, width, channels))
outputs <- inputs |> layer_average_pooling_3d(pool_size = 3)
outputs # Shape: (batch_size, 10, 10, 10, 3)


## <KerasTensor shape=(None, 10, 10, 10, 3), dtype=float32, sparse=False, name=keras_tensor_1>
```

See Also

- https://keras.io/api/layers/pooling_layers/average_pooling3d#averagepooling3d-class

Other pooling layers:
layer_average_pooling_1d()
layer_average_pooling_2d()
layer_global_average_pooling_1d()
layer_global_average_pooling_2d()
layer_global_average_pooling_3d()
layer_global_max_pooling_1d()
layer_global_max_pooling_2d()
layer_global_max_pooling_3d()
layer_max_pooling_1d()
layer_max_pooling_2d()
layer_max_pooling_3d()

Other layers:
Layer()
layer_activation()
layer_activation_elu()
layer_activation_leaky_relu()
layer_activation_parametric_relu()
layer_activation_relu()
layer_activation_softmax()
layer_activity_regularization()
layer_add()
layer_additive_attention()
layer_alpha_dropout()
layer_attention()
layer_average()
layer_average_pooling_1d()
layer_average_pooling_2d()
layer_batch_normalization()
layer_bidirectional()
layer_category_encoding()
layer_center_crop()

layer_concatenate()
layer_conv_1d()
layer_conv_1d_transpose()
layer_conv_2d()
layer_conv_2d_transpose()
layer_conv_3d()
layer_conv_3d_transpose()
layer_conv_lstm_1d()
layer_conv_lstm_2d()
layer_conv_lstm_3d()
layer_cropping_1d()
layer_cropping_2d()
layer_cropping_3d()
layer_dense()
layer_depthwise_conv_1d()
layer_depthwise_conv_2d()
layer_discretization()
layer_dot()
layer_dropout()
layer_einsum_dense()
layer_embedding()
layer_feature_space()
layer_flatten()
layer_flax_module_wrapper()
layer_gaussian_dropout()
layer_gaussian_noise()
layer_global_average_pooling_1d()
layer_global_average_pooling_2d()
layer_global_average_pooling_3d()
layer_global_max_pooling_1d()
layer_global_max_pooling_2d()
layer_global_max_pooling_3d()
layer_group_normalization()
layer_group_query_attention()
layer_gru()
layer_hashed_crossing()
layer_hashing()
layer_identity()
layer_integer_lookup()
layer_jax_model_wrapper()
layer_lambda()
layer_layer_normalization()
layer_lstm()
layer_masking()
layer_max_pooling_1d()
layer_max_pooling_2d()
layer_max_pooling_3d()
layer_maximum()

layer_mel_spectrogram()
layer_minimum()
layer_multi_head_attention()
layer_multiply()
layer_normalization()
layer_permute()
layer_random_brightness()
layer_random_contrast()
layer_random_crop()
layer_random_flip()
layer_random_rotation()
layer_random_translation()
layer_random_zoom()
layer_repeat_vector()
layer_rescaling()
layer_reshape()
layer_resizing()
layer_rnn()
layer_separable_conv_1d()
layer_separable_conv_2d()
layer_simple_rnn()
layer_spatial_dropout_1d()
layer_spatial_dropout_2d()
layer_spatial_dropout_3d()
layer_spectral_normalization()
layer_string_lookup()
layer_subtract()
layer_text_vectorization()
layer_tfsm()
layer_time_distributed()
layer_torch_module_wrapper()
layer_unit_normalization()
layer_upsampling_1d()
layer_upsampling_2d()
layer_upsampling_3d()
layer_zero_padding_1d()
layer_zero_padding_2d()
layer_zero_padding_3d()
rnn_cell_gru()
rnn_cell_lstm()
rnn_cell_simple()
rnn_cells_stack()

---

layer_batch_normalization

Layer that normalizes its inputs.

Description

Batch normalization applies a transformation that maintains the mean output close to 0 and the output standard deviation close to 1.

Importantly, batch normalization works differently during training and during inference.

During training (i.e. when using `fit()` or when calling the layer/model with the argument `training = TRUE`), the layer normalizes its output using the mean and standard deviation of the current batch of inputs. That is to say, for each channel being normalized, the layer returns `gamma * (batch - mean(batch)) / sqrt(var(batch) + epsilon) + beta`, where:

- `epsilon` is small constant (configurable as part of the constructor arguments)
- `gamma` is a learned scaling factor (initialized as 1), which can be disabled by passing `scale = FALSE` to the constructor.
- `beta` is a learned offset factor (initialized as 0), which can be disabled by passing `center = FALSE` to the constructor.

During inference (i.e. when using `evaluate()` or `predict()` or when calling the layer/model with the argument `training = FALSE` (which is the default), the layer normalizes its output using a moving average of the mean and standard deviation of the batches it has seen during training. That is to say, it returns `gamma * (batch - self$moving_mean) / sqrt(self$moving_var+epsilon) + beta`.

`self$moving_mean` and `self$moving_var` are non-trainable variables that are updated each time the layer in called in training mode, as such:

- `moving_mean = moving_mean * momentum + mean(batch) * (1 - momentum)`
- `moving_var = moving_var * momentum + var(batch) * (1 - momentum)`

As such, the layer will only normalize its inputs during inference *after having been trained on data that has similar statistics as the inference data*.

About setting `layer$trainable <- FALSE` on a `BatchNormalization` layer:

The meaning of setting `layer$trainable <- FALSE` is to freeze the layer, i.e. its internal state will not change during training: its trainable weights will not be updated during `fit()` or `train_on_batch()`, and its state updates will not be run.

Usually, this does not necessarily mean that the layer is run in inference mode (which is normally controlled by the `training` argument that can be passed when calling a layer). "Frozen state" and "inference mode" are two separate concepts.

However, in the case of the `BatchNormalization` layer, **setting `trainable <- FALSE` on the layer means that the layer will be subsequently run in inference mode** (meaning that it will use the moving mean and the moving variance to normalize the current batch, rather than using the mean and variance of the current batch).

Note that:

- Setting `trainable` on an model containing other layers will recursively set the `trainable` value of all inner layers.
- If the value of the `trainable` attribute is changed after calling `compile()` on a model, the new value doesn't take effect for this model until `compile()` is called again.

Usage

```
layer_batch_normalization(
  object,
  axis = -1L,
  momentum = 0.99,
  epsilon = 0.001,
  center = TRUE,
  scale = TRUE,
  beta_initializer = "zeros",
  gamma_initializer = "ones",
  moving_mean_initializer = "zeros",
  moving_variance_initializer = "ones",
  beta_regularizer = NULL,
  gamma_regularizer = NULL,
  beta_constraint = NULL,
  gamma_constraint = NULL,
  synchronized = FALSE,
  ...
)
```

Arguments

`object`	Object to compose the layer with. A tensor, array, or sequential model.
`axis`	Integer, the axis that should be normalized (typically the features axis). For instance, after a `Conv2D` layer with `data_format = "channels_first"`, use `axis = 2`.
`momentum`	Momentum for the moving average.
`epsilon`	Small float added to variance to avoid dividing by zero.
`center`	If `TRUE`, add offset of `beta` to normalized tensor. If `FALSE`, `beta` is ignored.
`scale`	If `TRUE`, multiply by `gamma`. If `FALSE`, `gamma` is not used. When the next layer is linear this can be disabled since the scaling will be done by the next layer.
`beta_initializer`	Initializer for the beta weight.
`gamma_initializer`	Initializer for the gamma weight.
`moving_mean_initializer`	Initializer for the moving mean.
`moving_variance_initializer`	Initializer for the moving variance.
`beta_regularizer`	Optional regularizer for the beta weight.
`gamma_regularizer`	Optional regularizer for the gamma weight.
`beta_constraint`	Optional constraint for the beta weight.

`gamma_constraint`	Optional constraint for the gamma weight.
`synchronized`	Only applicable with the TensorFlow backend. If `TRUE`, synchronizes the global batch statistics (mean and variance) for the layer across all devices at each training step in a distributed training strategy. If `FALSE`, each replica uses its own local batch statistics.
`...`	Base layer keyword arguments (e.g. `name` and `dtype`).

Value

The return value depends on the value provided for the first argument. If `object` is:

- a `keras_model_sequential()`, then the layer is added to the sequential model (which is modified in place). To enable piping, the sequential model is also returned, invisibly.
- a `keras_input()`, then the output tensor from calling `layer(input)` is returned.
- `NULL` or missing, then a `Layer` instance is returned.

Call Arguments

- `inputs`: Input tensor (of any rank).
- `training`: R boolean indicating whether the layer should behave in training mode or in inference mode.
  - `training = TRUE`: The layer will normalize its inputs using the mean and variance of the current batch of inputs.
  - `training = FALSE`: The layer will normalize its inputs using the mean and variance of its moving statistics, learned during training.
- `mask`: Binary tensor of shape broadcastable to `inputs` tensor, with `TRUE` values indicating the positions for which mean and variance should be computed. Masked elements of the current inputs are not taken into account for mean and variance computation during training. Any prior unmasked element values will be taken into account until their momentum expires.

Reference

- Ioffe and Szegedy, 2015.

See Also

- `https://keras.io/api/layers/normalization_layers/batch_normalization#batchnormalization-class`

Other normalization layers:
`layer_group_normalization()`
`layer_layer_normalization()`
`layer_spectral_normalization()`
`layer_unit_normalization()`

Other layers:
`Layer()`
`layer_activation()`

layer_activation_elu()
layer_activation_leaky_relu()
layer_activation_parametric_relu()
layer_activation_relu()
layer_activation_softmax()
layer_activity_regularization()
layer_add()
layer_additive_attention()
layer_alpha_dropout()
layer_attention()
layer_average()
layer_average_pooling_1d()
layer_average_pooling_2d()
layer_average_pooling_3d()
layer_bidirectional()
layer_category_encoding()
layer_center_crop()
layer_concatenate()
layer_conv_1d()
layer_conv_1d_transpose()
layer_conv_2d()
layer_conv_2d_transpose()
layer_conv_3d()
layer_conv_3d_transpose()
layer_conv_lstm_1d()
layer_conv_lstm_2d()
layer_conv_lstm_3d()
layer_cropping_1d()
layer_cropping_2d()
layer_cropping_3d()
layer_dense()
layer_depthwise_conv_1d()
layer_depthwise_conv_2d()
layer_discretization()
layer_dot()
layer_dropout()
layer_einsum_dense()
layer_embedding()
layer_feature_space()
layer_flatten()
layer_flax_module_wrapper()
layer_gaussian_dropout()
layer_gaussian_noise()
layer_global_average_pooling_1d()
layer_global_average_pooling_2d()
layer_global_average_pooling_3d()
layer_global_max_pooling_1d()
layer_global_max_pooling_2d()

layer_global_max_pooling_3d()
layer_group_normalization()
layer_group_query_attention()
layer_gru()
layer_hashed_crossing()
layer_hashing()
layer_identity()
layer_integer_lookup()
layer_jax_model_wrapper()
layer_lambda()
layer_layer_normalization()
layer_lstm()
layer_masking()
layer_max_pooling_1d()
layer_max_pooling_2d()
layer_max_pooling_3d()
layer_maximum()
layer_mel_spectrogram()
layer_minimum()
layer_multi_head_attention()
layer_multiply()
layer_normalization()
layer_permute()
layer_random_brightness()
layer_random_contrast()
layer_random_crop()
layer_random_flip()
layer_random_rotation()
layer_random_translation()
layer_random_zoom()
layer_repeat_vector()
layer_rescaling()
layer_reshape()
layer_resizing()
layer_rnn()
layer_separable_conv_1d()
layer_separable_conv_2d()
layer_simple_rnn()
layer_spatial_dropout_1d()
layer_spatial_dropout_2d()
layer_spatial_dropout_3d()
layer_spectral_normalization()
layer_string_lookup()
layer_subtract()
layer_text_vectorization()
layer_tfsm()
layer_time_distributed()
layer_torch_module_wrapper()

layer_unit_normalization()
layer_upsampling_1d()
layer_upsampling_2d()
layer_upsampling_3d()
layer_zero_padding_1d()
layer_zero_padding_2d()
layer_zero_padding_3d()
rnn_cell_gru()
rnn_cell_lstm()
rnn_cell_simple()
rnn_cells_stack()

---

layer_bidirectional *Bidirectional wrapper for RNNs.*

Description

Bidirectional wrapper for RNNs.

Usage

```
layer_bidirectional(
  object,
  layer,
  merge_mode = "concat",
  weights = NULL,
  backward_layer = NULL,
  ...
)
```

Arguments

object	Object to compose the layer with. A tensor, array, or sequential model.
layer	RNN instance, such as layer_lstm() or layer_gru(). It could also be a Layer() instance that meets the following criteria: 1. Be a sequence-processing layer (accepts 3D+ inputs). 2. Have a go_backwards, return_sequences and return_state attribute (with the same semantics as for the RNN class). 3. Have an input_spec attribute. 4. Implement serialization via get_config() and from_config(). Note that the recommended way to create new RNN layers is to write a custom RNN cell and use it with layer_rnn(), instead of subclassing with Layer() directly. When return_sequences is TRUE, the output of the masked timestep will be zero regardless of the layer's original zero_output_for_mask value.

`merge_mode`	Mode by which outputs of the forward and backward RNNs will be combined. One of `{"sum", "mul", "concat", "ave", NULL}`. If `NULL`, the outputs will not be combined, they will be returned as a list. Defaults to `"concat"`.
`weights`	see description
`backward_layer`	Optional `RNN`, or `Layer()` instance to be used to handle backwards input processing. If `backward_layer` is not provided, the layer instance passed as the `layer` argument will be used to generate the backward layer automatically. Note that the provided `backward_layer` layer should have properties matching those of the `layer` argument, in particular it should have the same values for `stateful`, `return_states`, `return_sequences`, etc. In addition, `backward_layer` and `layer` should have different `go_backwards` argument values. A `ValueError` will be raised if these requirements are not met.
`...`	For forward/backward compatability.

Value

The return value depends on the value provided for the first argument. If `object` is:

- a `keras_model_sequential()`, then the layer is added to the sequential model (which is modified in place). To enable piping, the sequential model is also returned, invisibly.
- a `keras_input()`, then the output tensor from calling `layer(input)` is returned.
- `NULL` or missing, then a `Layer` instance is returned.

Call Arguments

The call arguments for this layer are the same as those of the wrapped RNN layer. Beware that when passing the `initial_state` argument during the call of this layer, the first half in the list of elements in the `initial_state` list will be passed to the forward RNN call and the last half in the list of elements will be passed to the backward RNN call.

Note

instantiating a `Bidirectional` layer from an existing RNN layer instance will not reuse the weights state of the RNN layer instance – the `Bidirectional` layer will have freshly initialized weights.

Examples

```
model <- keras_model_sequential(input_shape = c(5, 10)) %>%
  layer_bidirectional(layer_lstm(units = 10, return_sequences = TRUE)) %>%
  layer_bidirectional(layer_lstm(units = 10)) %>%
  layer_dense(5, activation = "softmax")

model %>% compile(loss = "categorical_crossentropy",
                  optimizer = "rmsprop")

# With custom backward layer
forward_layer <- layer_lstm(units = 10, return_sequences = TRUE)
backward_layer <- layer_lstm(units = 10, activation = "relu",
                              return_sequences = TRUE, go_backwards = TRUE)

model <- keras_model_sequential(input_shape = c(5, 10)) %>%
  bidirectional(forward_layer, backward_layer = backward_layer) %>%
  layer_dense(5, activation = "softmax")

model %>% compile(loss = "categorical_crossentropy",
                  optimizer = "rmsprop")
```

States

A `Bidirectional` layer instance has property `states`, which you can access with `layer$states`. You can also reset states using `reset_state()`

See Also

- https://keras.io/api/layers/recurrent_layers/bidirectional#bidirectional-class

Other rnn layers:
`layer_conv_lstm_1d()`
`layer_conv_lstm_2d()`
`layer_conv_lstm_3d()`
`layer_gru()`
`layer_lstm()`
`layer_rnn()`
`layer_simple_rnn()`
`layer_time_distributed()`
`rnn_cell_gru()`
`rnn_cell_lstm()`
`rnn_cell_simple()`
`rnn_cells_stack()`

Other layers:
`Layer()`
`layer_activation()`
`layer_activation_elu()`
`layer_activation_leaky_relu()`
`layer_activation_parametric_relu()`
`layer_activation_relu()`
`layer_activation_softmax()`
`layer_activity_regularization()`
`layer_add()`
`layer_additive_attention()`
`layer_alpha_dropout()`
`layer_attention()`
`layer_average()`
`layer_average_pooling_1d()`
`layer_average_pooling_2d()`
`layer_average_pooling_3d()`

layer_batch_normalization()
layer_category_encoding()
layer_center_crop()
layer_concatenate()
layer_conv_1d()
layer_conv_1d_transpose()
layer_conv_2d()
layer_conv_2d_transpose()
layer_conv_3d()
layer_conv_3d_transpose()
layer_conv_lstm_1d()
layer_conv_lstm_2d()
layer_conv_lstm_3d()
layer_cropping_1d()
layer_cropping_2d()
layer_cropping_3d()
layer_dense()
layer_depthwise_conv_1d()
layer_depthwise_conv_2d()
layer_discretization()
layer_dot()
layer_dropout()
layer_einsum_dense()
layer_embedding()
layer_feature_space()
layer_flatten()
layer_flax_module_wrapper()
layer_gaussian_dropout()
layer_gaussian_noise()
layer_global_average_pooling_1d()
layer_global_average_pooling_2d()
layer_global_average_pooling_3d()
layer_global_max_pooling_1d()
layer_global_max_pooling_2d()
layer_global_max_pooling_3d()
layer_group_normalization()
layer_group_query_attention()
layer_gru()
layer_hashed_crossing()
layer_hashing()
layer_identity()
layer_integer_lookup()
layer_jax_model_wrapper()
layer_lambda()
layer_layer_normalization()
layer_lstm()
layer_masking()
layer_max_pooling_1d()

layer_max_pooling_2d()
layer_max_pooling_3d()
layer_maximum()
layer_mel_spectrogram()
layer_minimum()
layer_multi_head_attention()
layer_multiply()
layer_normalization()
layer_permute()
layer_random_brightness()
layer_random_contrast()
layer_random_crop()
layer_random_flip()
layer_random_rotation()
layer_random_translation()
layer_random_zoom()
layer_repeat_vector()
layer_rescaling()
layer_reshape()
layer_resizing()
layer_rnn()
layer_separable_conv_1d()
layer_separable_conv_2d()
layer_simple_rnn()
layer_spatial_dropout_1d()
layer_spatial_dropout_2d()
layer_spatial_dropout_3d()
layer_spectral_normalization()
layer_string_lookup()
layer_subtract()
layer_text_vectorization()
layer_tfsm()
layer_time_distributed()
layer_torch_module_wrapper()
layer_unit_normalization()
layer_upsampling_1d()
layer_upsampling_2d()
layer_upsampling_3d()
layer_zero_padding_1d()
layer_zero_padding_2d()
layer_zero_padding_3d()
rnn_cell_gru()
rnn_cell_lstm()
rnn_cell_simple()
rnn_cells_stack()

`layer_category_encoding`

A preprocessing layer which encodes integer features.

Description

This layer provides options for condensing data into a categorical encoding when the total number of tokens are known in advance. It accepts integer values as inputs, and it outputs a dense or sparse representation of those inputs. For integer inputs where the total number of tokens is not known, use `layer_integer_lookup()` instead.

Note: This layer is safe to use inside a `tf.data` pipeline (independently of which backend you're using).

Usage

```
layer_category_encoding(
  object,
  num_tokens = NULL,
  output_mode = "multi_hot",
  sparse = FALSE,
  ...
)
```

Arguments

`object`	Object to compose the layer with. A tensor, array, or sequential model.
`num_tokens`	The total number of tokens the layer should support. All inputs to the layer must integers in the range `0 <= value < num_tokens`, or an error will be thrown.
`output_mode`	Specification for the output of the layer. Values can be `"one_hot"`, `"multi_hot"` or `"count"`, configuring the layer as follows: - `"one_hot"`: Encodes each individual element in the input into an array of `num_tokens` size, containing a 1 at the element index. If the last dimension is size 1, will encode on that dimension. If the last dimension is not size 1, will append a new dimension for the encoded output. - `"multi_hot"`: Encodes each sample in the input into a single array of `num_tokens` size, containing a 1 for each vocabulary term present in the sample. Treats the last dimension as the sample dimension, if input shape is `(..., sample_length)`, output shape will be `(..., num_tokens)`. - `"count"`: Like `"multi_hot"`, but the int array contains a count of the number of times the token at that index appeared in the sample. For all output modes, currently only output up to rank 2 is supported. Defaults to `"multi_hot"`.
`sparse`	Whether to return a sparse tensor; for backends that support sparse tensors.
`...`	For forward/backward compatability.

Value

The return value depends on the value provided for the first argument. If `object` is:

- a `keras_model_sequential()`, then the layer is added to the sequential model (which is modified in place). To enable piping, the sequential model is also returned, invisibly.
- a `keras_input()`, then the output tensor from calling `layer(input)` is returned.
- `NULL` or missing, then a `Layer` instance is returned.

Examples

One-hot encoding data

```
layer <- layer_category_encoding(num_tokens = 4, output_mode = "one_hot")
x <- op_array(c(3, 2, 0, 1), "int32")
layer(x)

## tf.Tensor(
## [[0. 0. 0. 1.]
##  [0. 0. 1. 0.]
##  [1. 0. 0. 0.]
##  [0. 1. 0. 0.]], shape=(4, 4), dtype=float32)
```

Multi-hot encoding data

```
layer <- layer_category_encoding(num_tokens = 4, output_mode = "multi_hot")
x <- op_array(rbind(c(0, 1),
                   c(0, 0),
                   c(1, 2),
                   c(3, 1)), "int32")
layer(x)

## tf.Tensor(
## [[1. 1. 0. 0.]
##  [1. 0. 0. 0.]
##  [0. 1. 1. 0.]
##  [0. 1. 0. 1.]], shape=(4, 4), dtype=float32)
```

Using weighted inputs in `"count"` mode

```
layer <- layer_category_encoding(num_tokens = 4, output_mode = "count")
count_weights <- op_array(rbind(c(.1, .2),
                               c(.1, .1),
                               c(.2, .3),
                               c(.4, .2)))
x <- op_array(rbind(c(0, 1),
                   c(0, 0),
                   c(1, 2),
                   c(3, 1)), "int32")
layer(x, count_weights = count_weights)
#   array([[01, 02, 0. , 0. ],
#          [02, 0. , 0. , 0. ],
#          [0. , 02, 03, 0. ],
#          [0. , 02, 0. , 04]]>
```

Call Arguments

- `inputs`: A 1D or 2D tensor of integer inputs.
- `count_weights`: A tensor in the same shape as `inputs` indicating the weight for each sample value when summing up in `count` mode. Not used in `"multi_hot"` or `"one_hot"` modes.

See Also

- `https://keras.io/api/layers/preprocessing_layers/categorical/category_encoding#categoryencoding-class`

Other categorical features preprocessing layers:
`layer_hashed_crossing()`
`layer_hashing()`
`layer_integer_lookup()`
`layer_string_lookup()`

Other preprocessing layers:
`layer_center_crop()`
`layer_discretization()`
`layer_feature_space()`
`layer_hashed_crossing()`
`layer_hashing()`
`layer_integer_lookup()`
`layer_mel_spectrogram()`
`layer_normalization()`
`layer_random_brightness()`
`layer_random_contrast()`
`layer_random_crop()`
`layer_random_flip()`
`layer_random_rotation()`
`layer_random_translation()`
`layer_random_zoom()`
`layer_rescaling()`
`layer_resizing()`
`layer_string_lookup()`
`layer_text_vectorization()`

Other layers:
`Layer()`

layer_activation()
layer_activation_elu()
layer_activation_leaky_relu()
layer_activation_parametric_relu()
layer_activation_relu()
layer_activation_softmax()
layer_activity_regularization()
layer_add()
layer_additive_attention()
layer_alpha_dropout()
layer_attention()
layer_average()
layer_average_pooling_1d()
layer_average_pooling_2d()
layer_average_pooling_3d()
layer_batch_normalization()
layer_bidirectional()
layer_center_crop()
layer_concatenate()
layer_conv_1d()
layer_conv_1d_transpose()
layer_conv_2d()
layer_conv_2d_transpose()
layer_conv_3d()
layer_conv_3d_transpose()
layer_conv_lstm_1d()
layer_conv_lstm_2d()
layer_conv_lstm_3d()
layer_cropping_1d()
layer_cropping_2d()
layer_cropping_3d()
layer_dense()
layer_depthwise_conv_1d()
layer_depthwise_conv_2d()
layer_discretization()
layer_dot()
layer_dropout()
layer_einsum_dense()
layer_embedding()
layer_feature_space()
layer_flatten()
layer_flax_module_wrapper()
layer_gaussian_dropout()
layer_gaussian_noise()
layer_global_average_pooling_1d()
layer_global_average_pooling_2d()
layer_global_average_pooling_3d()
layer_global_max_pooling_1d()

layer_global_max_pooling_2d()
layer_global_max_pooling_3d()
layer_group_normalization()
layer_group_query_attention()
layer_gru()
layer_hashed_crossing()
layer_hashing()
layer_identity()
layer_integer_lookup()
layer_jax_model_wrapper()
layer_lambda()
layer_layer_normalization()
layer_lstm()
layer_masking()
layer_max_pooling_1d()
layer_max_pooling_2d()
layer_max_pooling_3d()
layer_maximum()
layer_mel_spectrogram()
layer_minimum()
layer_multi_head_attention()
layer_multiply()
layer_normalization()
layer_permute()
layer_random_brightness()
layer_random_contrast()
layer_random_crop()
layer_random_flip()
layer_random_rotation()
layer_random_translation()
layer_random_zoom()
layer_repeat_vector()
layer_rescaling()
layer_reshape()
layer_resizing()
layer_rnn()
layer_separable_conv_1d()
layer_separable_conv_2d()
layer_simple_rnn()
layer_spatial_dropout_1d()
layer_spatial_dropout_2d()
layer_spatial_dropout_3d()
layer_spectral_normalization()
layer_string_lookup()
layer_subtract()
layer_text_vectorization()
layer_tfsm()
layer_time_distributed()

layer_torch_module_wrapper()
layer_unit_normalization()
layer_upsampling_1d()
layer_upsampling_2d()
layer_upsampling_3d()
layer_zero_padding_1d()
layer_zero_padding_2d()
layer_zero_padding_3d()
rnn_cell_gru()
rnn_cell_lstm()
rnn_cell_simple()
rnn_cells_stack()

layer_center_crop *A preprocessing layer which crops images.*

Description

This layers crops the central portion of the images to a target size. If an image is smaller than the target size, it will be resized and cropped so as to return the largest possible window in the image that matches the target aspect ratio.

Input pixel values can be of any range (e.g. `[0., 1.)` or `[0, 255]`).

Usage

```
layer_center_crop(object, height, width, data_format = NULL, ...)
```

Arguments

object	Object to compose the layer with. A tensor, array, or sequential model.
height	Integer, the height of the output shape.
width	Integer, the width of the output shape.
data_format	string, either "channels_last" or "channels_first". The ordering of the dimensions in the inputs. "channels_last" corresponds to inputs with shape (batch, height, width, channels) while "channels_first" corresponds to inputs with shape (batch, channels, height, width). It defaults to the image_data_format value found in your Keras config file at ~/.keras/keras.json. If you never set it, then it will be "channels_last".
...	For forward/backward compatability.

Value

The return value depends on the value provided for the first argument. If `object` is:

- a `keras_model_sequential()`, then the layer is added to the sequential model (which is modified in place). To enable piping, the sequential model is also returned, invisibly.
- a `keras_input()`, then the output tensor from calling `layer(input)` is returned.
- `NULL` or missing, then a `Layer` instance is returned.

Input Shape

3D (unbatched) or 4D (batched) tensor with shape: `(..., height, width, channels)`, in `"channels_last"` format, or `(..., channels, height, width)`, in `"channels_first"` format.

Output Shape

3D (unbatched) or 4D (batched) tensor with shape: `(..., target_height, target_width, channels)`, or `(..., channels, target_height, target_width)`, in `"channels_first"` format.

If the input height/width is even and the target height/width is odd (or inversely), the input image is left-padded by 1 pixel.

Note: This layer is safe to use inside a `tf.data` pipeline (independently of which backend you're using).

See Also

- `https://keras.io/api/layers/preprocessing_layers/image_preprocessing/center_crop#centercrop-class`

Other image preprocessing layers:
`layer_rescaling()`
`layer_resizing()`

Other preprocessing layers:
`layer_category_encoding()`
`layer_discretization()`
`layer_feature_space()`
`layer_hashed_crossing()`
`layer_hashing()`
`layer_integer_lookup()`
`layer_mel_spectrogram()`
`layer_normalization()`
`layer_random_brightness()`
`layer_random_contrast()`
`layer_random_crop()`
`layer_random_flip()`
`layer_random_rotation()`
`layer_random_translation()`
`layer_random_zoom()`

layer_rescaling()
layer_resizing()
layer_string_lookup()
layer_text_vectorization()

Other layers:
Layer()
layer_activation()
layer_activation_elu()
layer_activation_leaky_relu()
layer_activation_parametric_relu()
layer_activation_relu()
layer_activation_softmax()
layer_activity_regularization()
layer_add()
layer_additive_attention()
layer_alpha_dropout()
layer_attention()
layer_average()
layer_average_pooling_1d()
layer_average_pooling_2d()
layer_average_pooling_3d()
layer_batch_normalization()
layer_bidirectional()
layer_category_encoding()
layer_concatenate()
layer_conv_1d()
layer_conv_1d_transpose()
layer_conv_2d()
layer_conv_2d_transpose()
layer_conv_3d()
layer_conv_3d_transpose()
layer_conv_lstm_1d()
layer_conv_lstm_2d()
layer_conv_lstm_3d()
layer_cropping_1d()
layer_cropping_2d()
layer_cropping_3d()
layer_dense()
layer_depthwise_conv_1d()
layer_depthwise_conv_2d()
layer_discretization()
layer_dot()
layer_dropout()
layer_einsum_dense()
layer_embedding()
layer_feature_space()
layer_flatten()

layer_flax_module_wrapper()
layer_gaussian_dropout()
layer_gaussian_noise()
layer_global_average_pooling_1d()
layer_global_average_pooling_2d()
layer_global_average_pooling_3d()
layer_global_max_pooling_1d()
layer_global_max_pooling_2d()
layer_global_max_pooling_3d()
layer_group_normalization()
layer_group_query_attention()
layer_gru()
layer_hashed_crossing()
layer_hashing()
layer_identity()
layer_integer_lookup()
layer_jax_model_wrapper()
layer_lambda()
layer_layer_normalization()
layer_lstm()
layer_masking()
layer_max_pooling_1d()
layer_max_pooling_2d()
layer_max_pooling_3d()
layer_maximum()
layer_mel_spectrogram()
layer_minimum()
layer_multi_head_attention()
layer_multiply()
layer_normalization()
layer_permute()
layer_random_brightness()
layer_random_contrast()
layer_random_crop()
layer_random_flip()
layer_random_rotation()
layer_random_translation()
layer_random_zoom()
layer_repeat_vector()
layer_rescaling()
layer_reshape()
layer_resizing()
layer_rnn()
layer_separable_conv_1d()
layer_separable_conv_2d()
layer_simple_rnn()
layer_spatial_dropout_1d()
layer_spatial_dropout_2d()

layer_spatial_dropout_3d()
layer_spectral_normalization()
layer_string_lookup()
layer_subtract()
layer_text_vectorization()
layer_tfsm()
layer_time_distributed()
layer_torch_module_wrapper()
layer_unit_normalization()
layer_upsampling_1d()
layer_upsampling_2d()
layer_upsampling_3d()
layer_zero_padding_1d()
layer_zero_padding_2d()
layer_zero_padding_3d()
rnn_cell_gru()
rnn_cell_lstm()
rnn_cell_simple()
rnn_cells_stack()

---

layer_concatenate *Concatenates a list of inputs.*

Description

It takes as input a list of tensors, all of the same shape except for the concatenation axis, and returns a single tensor that is the concatenation of all inputs.

Usage

```
layer_concatenate(inputs, ..., axis = -1L)
```

Arguments

inputs	layers to combine
...	Standard layer keyword arguments.
axis	Axis along which to concatenate.

Value

A tensor, the concatenation of the inputs alongside axis axis.

Examples

```
x <- op_arange(20) |> op_reshape(c(2, 2, 5))
y <- op_arange(20, 40) |> op_reshape(c(2, 2, 5))
layer_concatenate(x, y, axis = 2)
```

```
## tf.Tensor(
## [[[ 0.  1.  2.  3.  4.]
##   [ 5.  6.  7.  8.  9.]
##   [20. 21. 22. 23. 24.]
##   [25. 26. 27. 28. 29.]]
##
##  [[10. 11. 12. 13. 14.]
##   [15. 16. 17. 18. 19.]
##   [30. 31. 32. 33. 34.]
##   [35. 36. 37. 38. 39.]]], shape=(2, 4, 5), dtype=float32)
```

Usage in a Keras model:

```
x1 <- op_arange(10)     |> op_reshape(c(5, 2)) |> layer_dense(8)
x2 <- op_arange(10, 20) |> op_reshape(c(5, 2)) |> layer_dense(8)
y <- layer_concatenate(x1, x2)
```

See Also

- https://keras.io/api/layers/merging_layers/concatenate#concatenate-class

Other merging layers:
layer_add()
layer_average()
layer_dot()
layer_maximum()
layer_minimum()
layer_multiply()
layer_subtract()

Other layers:
Layer()
layer_activation()
layer_activation_elu()
layer_activation_leaky_relu()
layer_activation_parametric_relu()
layer_activation_relu()
layer_activation_softmax()
layer_activity_regularization()
layer_add()
layer_additive_attention()
layer_alpha_dropout()

layer_attention()
layer_average()
layer_average_pooling_1d()
layer_average_pooling_2d()
layer_average_pooling_3d()
layer_batch_normalization()
layer_bidirectional()
layer_category_encoding()
layer_center_crop()
layer_conv_1d()
layer_conv_1d_transpose()
layer_conv_2d()
layer_conv_2d_transpose()
layer_conv_3d()
layer_conv_3d_transpose()
layer_conv_lstm_1d()
layer_conv_lstm_2d()
layer_conv_lstm_3d()
layer_cropping_1d()
layer_cropping_2d()
layer_cropping_3d()
layer_dense()
layer_depthwise_conv_1d()
layer_depthwise_conv_2d()
layer_discretization()
layer_dot()
layer_dropout()
layer_einsum_dense()
layer_embedding()
layer_feature_space()
layer_flatten()
layer_flax_module_wrapper()
layer_gaussian_dropout()
layer_gaussian_noise()
layer_global_average_pooling_1d()
layer_global_average_pooling_2d()
layer_global_average_pooling_3d()
layer_global_max_pooling_1d()
layer_global_max_pooling_2d()
layer_global_max_pooling_3d()
layer_group_normalization()
layer_group_query_attention()
layer_gru()
layer_hashed_crossing()
layer_hashing()
layer_identity()
layer_integer_lookup()
layer_jax_model_wrapper()

layer_lambda()
layer_layer_normalization()
layer_lstm()
layer_masking()
layer_max_pooling_1d()
layer_max_pooling_2d()
layer_max_pooling_3d()
layer_maximum()
layer_mel_spectrogram()
layer_minimum()
layer_multi_head_attention()
layer_multiply()
layer_normalization()
layer_permute()
layer_random_brightness()
layer_random_contrast()
layer_random_crop()
layer_random_flip()
layer_random_rotation()
layer_random_translation()
layer_random_zoom()
layer_repeat_vector()
layer_rescaling()
layer_reshape()
layer_resizing()
layer_rnn()
layer_separable_conv_1d()
layer_separable_conv_2d()
layer_simple_rnn()
layer_spatial_dropout_1d()
layer_spatial_dropout_2d()
layer_spatial_dropout_3d()
layer_spectral_normalization()
layer_string_lookup()
layer_subtract()
layer_text_vectorization()
layer_tfsm()
layer_time_distributed()
layer_torch_module_wrapper()
layer_unit_normalization()
layer_upsampling_1d()
layer_upsampling_2d()
layer_upsampling_3d()
layer_zero_padding_1d()
layer_zero_padding_2d()
layer_zero_padding_3d()
rnn_cell_gru()
rnn_cell_lstm()

rnn_cell_simple()
rnn_cells_stack()

`layer_conv_1d` *1D convolution layer (e.g. temporal convolution).*

Description

This layer creates a convolution kernel that is convolved with the layer input over a single spatial (or temporal) dimension to produce a tensor of outputs. If `use_bias` is `TRUE`, a bias vector is created and added to the outputs. Finally, if `activation` is not `NULL`, it is applied to the outputs as well.

Usage

```
layer_conv_1d(
  object,
  filters,
  kernel_size,
  strides = 1L,
  padding = "valid",
  data_format = NULL,
  dilation_rate = 1L,
  groups = 1L,
  activation = NULL,
  use_bias = TRUE,
  kernel_initializer = "glorot_uniform",
  bias_initializer = "zeros",
  kernel_regularizer = NULL,
  bias_regularizer = NULL,
  activity_regularizer = NULL,
  kernel_constraint = NULL,
  bias_constraint = NULL,
  ...
)
```

Arguments

`object`	Object to compose the layer with. A tensor, array, or sequential model.
`filters`	int, the dimension of the output space (the number of filters in the convolution).
`kernel_size`	int or list of 1 integer, specifying the size of the convolution window.
`strides`	int or list of 1 integer, specifying the stride length of the convolution. `strides` > 1 is incompatible with `dilation_rate` > 1.
`padding`	string, `"valid"`, `"same"` or `"causal"`(case-insensitive). `"valid"` means no padding. `"same"` results in padding evenly to the left/right or up/down of the input. When `padding="same"` and `strides=1`, the output has the same size as

	the input. `"causal"` results in causal (dilated) convolutions, e.g. `output[t]` does not depend on`tail(input, t+1)`. Useful when modeling temporal data where the model should not violate the temporal order. See WaveNet: A Generative Model for Raw Audio, section2.1.
`data_format`	string, either `"channels_last"` or `"channels_first"`. The ordering of the dimensions in the inputs. `"channels_last"` corresponds to inputs with shape `(batch, steps, features)` while `"channels_first"` corresponds to inputs with shape `(batch, features, steps)`. It defaults to the `image_data_format` value found in your Keras config file at `~/.keras/keras.json`. If you never set it, then it will be `"channels_last"`.
`dilation_rate`	int or list of 1 integers, specifying the dilation rate to use for dilated convolution.
`groups`	A positive int specifying the number of groups in which the input is split along the channel axis. Each group is convolved separately with `filters // groups` filters. The output is the concatenation of all the `groups` results along the channel axis. Input channels and `filters` must both be divisible by `groups`.
`activation`	Activation function. If `NULL`, no activation is applied.
`use_bias`	bool, if `TRUE`, bias will be added to the output.
`kernel_initializer`	Initializer for the convolution kernel. If `NULL`, the default initializer (`"glorot_uniform"`) will be used.
`bias_initializer`	Initializer for the bias vector. If `NULL`, the default initializer (`"zeros"`) will be used.
`kernel_regularizer`	Optional regularizer for the convolution kernel.
`bias_regularizer`	Optional regularizer for the bias vector.
`activity_regularizer`	Optional regularizer function for the output.
`kernel_constraint`	Optional projection function to be applied to the kernel after being updated by an `Optimizer` (e.g. used to implement norm constraints or value constraints for layer weights). The function must take as input the unprojected variable and must return the projected variable (which must have the same shape). Constraints are not safe to use when doing asynchronous distributed training.
`bias_constraint`	Optional projection function to be applied to the bias after being updated by an `Optimizer`.
`...`	For forward/backward compatability.

Value

A 3D tensor representing `activation(conv1d(inputs, kernel) + bias)`.

Input Shape

- If `data_format="channels_last"`: A 3D tensor with shape: `(batch_shape, steps, channels)`
- If `data_format="channels_first"`: A 3D tensor with shape: `(batch_shape, channels, steps)`

Output Shape

- If data_format="channels_last": A 3D tensor with shape: (batch_shape, new_steps, filters)
- If data_format="channels_first": A 3D tensor with shape: (batch_shape, filters, new_steps)

Raises

ValueError: when both strides > 1 and dilation_rate > 1.

Example

```
# The inputs are 128-length vectors with 10 timesteps, and the
# batch size is 4.
x <- random_uniform(c(4, 10, 128))
y <- x |> layer_conv_1d(32, 3, activation='relu')
shape(y)

## shape(4, 8, 32)
```

See Also

- https://keras.io/api/layers/convolution_layers/convolution1d#conv1d-class

Other convolutional layers:
layer_conv_1d_transpose()
layer_conv_2d()
layer_conv_2d_transpose()
layer_conv_3d()
layer_conv_3d_transpose()
layer_depthwise_conv_1d()
layer_depthwise_conv_2d()
layer_separable_conv_1d()
layer_separable_conv_2d()

Other layers:
Layer()
layer_activation()
layer_activation_elu()
layer_activation_leaky_relu()
layer_activation_parametric_relu()
layer_activation_relu()
layer_activation_softmax()
layer_activity_regularization()
layer_add()
layer_additive_attention()
layer_alpha_dropout()
layer_attention()
layer_average()
layer_average_pooling_1d()

layer_average_pooling_2d()
layer_average_pooling_3d()
layer_batch_normalization()
layer_bidirectional()
layer_category_encoding()
layer_center_crop()
layer_concatenate()
layer_conv_1d_transpose()
layer_conv_2d()
layer_conv_2d_transpose()
layer_conv_3d()
layer_conv_3d_transpose()
layer_conv_lstm_1d()
layer_conv_lstm_2d()
layer_conv_lstm_3d()
layer_cropping_1d()
layer_cropping_2d()
layer_cropping_3d()
layer_dense()
layer_depthwise_conv_1d()
layer_depthwise_conv_2d()
layer_discretization()
layer_dot()
layer_dropout()
layer_einsum_dense()
layer_embedding()
layer_feature_space()
layer_flatten()
layer_flax_module_wrapper()
layer_gaussian_dropout()
layer_gaussian_noise()
layer_global_average_pooling_1d()
layer_global_average_pooling_2d()
layer_global_average_pooling_3d()
layer_global_max_pooling_1d()
layer_global_max_pooling_2d()
layer_global_max_pooling_3d()
layer_group_normalization()
layer_group_query_attention()
layer_gru()
layer_hashed_crossing()
layer_hashing()
layer_identity()
layer_integer_lookup()
layer_jax_model_wrapper()
layer_lambda()
layer_layer_normalization()
layer_lstm()

layer_masking()
layer_max_pooling_1d()
layer_max_pooling_2d()
layer_max_pooling_3d()
layer_maximum()
layer_mel_spectrogram()
layer_minimum()
layer_multi_head_attention()
layer_multiply()
layer_normalization()
layer_permute()
layer_random_brightness()
layer_random_contrast()
layer_random_crop()
layer_random_flip()
layer_random_rotation()
layer_random_translation()
layer_random_zoom()
layer_repeat_vector()
layer_rescaling()
layer_reshape()
layer_resizing()
layer_rnn()
layer_separable_conv_1d()
layer_separable_conv_2d()
layer_simple_rnn()
layer_spatial_dropout_1d()
layer_spatial_dropout_2d()
layer_spatial_dropout_3d()
layer_spectral_normalization()
layer_string_lookup()
layer_subtract()
layer_text_vectorization()
layer_tfsm()
layer_time_distributed()
layer_torch_module_wrapper()
layer_unit_normalization()
layer_upsampling_1d()
layer_upsampling_2d()
layer_upsampling_3d()
layer_zero_padding_1d()
layer_zero_padding_2d()
layer_zero_padding_3d()
rnn_cell_gru()
rnn_cell_lstm()
rnn_cell_simple()
rnn_cells_stack()

layer_conv_1d_transpose

1D transposed convolution layer.

Description

The need for transposed convolutions generally arise from the desire to use a transformation going in the opposite direction of a normal convolution, i.e., from something that has the shape of the output of some convolution to something that has the shape of its input while maintaining a connectivity pattern that is compatible with said convolution.

Usage

```
layer_conv_1d_transpose(
  object,
  filters,
  kernel_size,
  strides = 1L,
  padding = "valid",
  data_format = NULL,
  dilation_rate = 1L,
  activation = NULL,
  use_bias = TRUE,
  kernel_initializer = "glorot_uniform",
  bias_initializer = "zeros",
  kernel_regularizer = NULL,
  bias_regularizer = NULL,
  activity_regularizer = NULL,
  kernel_constraint = NULL,
  bias_constraint = NULL,
  ...
)
```

Arguments

`object`	Object to compose the layer with. A tensor, array, or sequential model.
`filters`	int, the dimension of the output space (the number of filters in the transpose convolution).
`kernel_size`	int or list of 1 integer, specifying the size of the transposed convolution window.
`strides`	int or list of 1 integer, specifying the stride length of the transposed convolution. `strides` > 1 is incompatible with `dilation_rate` > 1.
`padding`	string, either `"valid"` or `"same"` (case-insensitive). `"valid"` means no padding. `"same"` results in padding evenly to the left/right or up/down of the input such that output has the same height/width dimension as the input.

`data_format`	string, either `"channels_last"` or `"channels_first"`. The ordering of the dimensions in the inputs. `"channels_last"` corresponds to inputs with shape `(batch, steps, features)` while `"channels_first"` corresponds to inputs with shape `(batch, features, steps)`. It defaults to the `image_data_format` value found in your Keras config file at `~/.keras/keras.json`. If you never set it, then it will be `"channels_last"`.
`dilation_rate`	int or list of 1 integers, specifying the dilation rate to use for dilated transposed convolution.
`activation`	Activation function. If `NULL`, no activation is applied.
`use_bias`	bool, if `TRUE`, bias will be added to the output.
`kernel_initializer`	Initializer for the convolution kernel. If `NULL`, the default initializer (`"glorot_uniform"`) will be used.
`bias_initializer`	Initializer for the bias vector. If `NULL`, the default initializer (`"zeros"`) will be used.
`kernel_regularizer`	Optional regularizer for the convolution kernel.
`bias_regularizer`	Optional regularizer for the bias vector.
`activity_regularizer`	Optional regularizer function for the output.
`kernel_constraint`	Optional projection function to be applied to the kernel after being updated by an `Optimizer` (e.g. used to implement norm constraints or value constraints for layer weights). The function must take as input the unprojected variable and must return the projected variable (which must have the same shape). Constraints are not safe to use when doing asynchronous distributed training.
`bias_constraint`	Optional projection function to be applied to the bias after being updated by an `Optimizer`.
`...`	For forward/backward compatability.

Value

A 3D tensor representing `activation(conv1d_transpose(inputs, kernel) + bias)`.

Input Shape

- If `data_format="channels_last"`: A 3D tensor with shape: `(batch_shape, steps, channels)`
- If `data_format="channels_first"`: A 3D tensor with shape: `(batch_shape, channels, steps)`

Output Shape

- If `data_format="channels_last"`: A 3D tensor with shape: `(batch_shape, new_steps, filters)`
- If `data_format="channels_first"`: A 3D tensor with shape: `(batch_shape, filters, new_steps)`

Raises

ValueError: when both strides > 1 and dilation_rate > 1.

References

- A guide to convolution arithmetic for deep learning
- Deconvolutional Networks

Example

```
x <- random_uniform(c(4, 10, 128))
y <- x |> layer_conv_1d_transpose(32, 3, 2, activation='relu')
shape(y)

## shape(4, 21, 32)
```

See Also

- https://keras.io/api/layers/convolution_layers/convolution1d_transpose#conv1dtranspose-class

Other convolutional layers:
layer_conv_1d()
layer_conv_2d()
layer_conv_2d_transpose()
layer_conv_3d()
layer_conv_3d_transpose()
layer_depthwise_conv_1d()
layer_depthwise_conv_2d()
layer_separable_conv_1d()
layer_separable_conv_2d()

Other layers:
Layer()
layer_activation()
layer_activation_elu()
layer_activation_leaky_relu()
layer_activation_parametric_relu()
layer_activation_relu()
layer_activation_softmax()
layer_activity_regularization()
layer_add()
layer_additive_attention()
layer_alpha_dropout()
layer_attention()
layer_average()
layer_average_pooling_1d()
layer_average_pooling_2d()
layer_average_pooling_3d()

layer_batch_normalization()
layer_bidirectional()
layer_category_encoding()
layer_center_crop()
layer_concatenate()
layer_conv_1d()
layer_conv_2d()
layer_conv_2d_transpose()
layer_conv_3d()
layer_conv_3d_transpose()
layer_conv_lstm_1d()
layer_conv_lstm_2d()
layer_conv_lstm_3d()
layer_cropping_1d()
layer_cropping_2d()
layer_cropping_3d()
layer_dense()
layer_depthwise_conv_1d()
layer_depthwise_conv_2d()
layer_discretization()
layer_dot()
layer_dropout()
layer_einsum_dense()
layer_embedding()
layer_feature_space()
layer_flatten()
layer_flax_module_wrapper()
layer_gaussian_dropout()
layer_gaussian_noise()
layer_global_average_pooling_1d()
layer_global_average_pooling_2d()
layer_global_average_pooling_3d()
layer_global_max_pooling_1d()
layer_global_max_pooling_2d()
layer_global_max_pooling_3d()
layer_group_normalization()
layer_group_query_attention()
layer_gru()
layer_hashed_crossing()
layer_hashing()
layer_identity()
layer_integer_lookup()
layer_jax_model_wrapper()
layer_lambda()
layer_layer_normalization()
layer_lstm()
layer_masking()
layer_max_pooling_1d()

layer_max_pooling_2d()
layer_max_pooling_3d()
layer_maximum()
layer_mel_spectrogram()
layer_minimum()
layer_multi_head_attention()
layer_multiply()
layer_normalization()
layer_permute()
layer_random_brightness()
layer_random_contrast()
layer_random_crop()
layer_random_flip()
layer_random_rotation()
layer_random_translation()
layer_random_zoom()
layer_repeat_vector()
layer_rescaling()
layer_reshape()
layer_resizing()
layer_rnn()
layer_separable_conv_1d()
layer_separable_conv_2d()
layer_simple_rnn()
layer_spatial_dropout_1d()
layer_spatial_dropout_2d()
layer_spatial_dropout_3d()
layer_spectral_normalization()
layer_string_lookup()
layer_subtract()
layer_text_vectorization()
layer_tfsm()
layer_time_distributed()
layer_torch_module_wrapper()
layer_unit_normalization()
layer_upsampling_1d()
layer_upsampling_2d()
layer_upsampling_3d()
layer_zero_padding_1d()
layer_zero_padding_2d()
layer_zero_padding_3d()
rnn_cell_gru()
rnn_cell_lstm()
rnn_cell_simple()
rnn_cells_stack()

layer_conv_2d *2D convolution layer.*

Description

This layer creates a convolution kernel that is convolved with the layer input over a single spatial (or temporal) dimension to produce a tensor of outputs. If `use_bias` is `TRUE`, a bias vector is created and added to the outputs. Finally, if `activation` is not `NULL`, it is applied to the outputs as well.

Usage

```
layer_conv_2d(
  object,
  filters,
  kernel_size,
  strides = list(1L, 1L),
  padding = "valid",
  data_format = NULL,
  dilation_rate = list(1L, 1L),
  groups = 1L,
  activation = NULL,
  use_bias = TRUE,
  kernel_initializer = "glorot_uniform",
  bias_initializer = "zeros",
  kernel_regularizer = NULL,
  bias_regularizer = NULL,
  activity_regularizer = NULL,
  kernel_constraint = NULL,
  bias_constraint = NULL,
  ...
)
```

Arguments

`object`	Object to compose the layer with. A tensor, array, or sequential model.
`filters`	int, the dimension of the output space (the number of filters in the convolution).
`kernel_size`	int or list of 2 integer, specifying the size of the convolution window.
`strides`	int or list of 2 integer, specifying the stride length of the convolution. `strides > 1` is incompatible with `dilation_rate > 1`.
`padding`	string, either `"valid"` or `"same"` (case-insensitive). `"valid"` means no padding. `"same"` results in padding evenly to the left/right or up/down of the input. When `padding="same"` and `strides=1`, the output has the same size as the input.
`data_format`	string, either `"channels_last"` or `"channels_first"`. The ordering of the dimensions in the inputs. `"channels_last"` corresponds to inputs with shape `(batch_size, height, width, channels)` while `"channels_first"` corresponds to inputs with shape `(batch_size, channels, height, width)`.

It defaults to the `image_data_format` value found in your Keras config file at `~/.keras/keras.json`. If you never set it, then it will be `"channels_last"`.

`dilation_rate`	int or list of 2 integers, specifying the dilation rate to use for dilated convolution.
`groups`	A positive int specifying the number of groups in which the input is split along the channel axis. Each group is convolved separately with `filters // groups` filters. The output is the concatenation of all the `groups` results along the channel axis. Input channels and `filters` must both be divisible by `groups`.
`activation`	Activation function. If `NULL`, no activation is applied.
`use_bias`	bool, if `TRUE`, bias will be added to the output.
`kernel_initializer`	Initializer for the convolution kernel. If `NULL`, the default initializer (`"glorot_uniform"`) will be used.
`bias_initializer`	Initializer for the bias vector. If `NULL`, the default initializer (`"zeros"`) will be used.
`kernel_regularizer`	Optional regularizer for the convolution kernel.
`bias_regularizer`	Optional regularizer for the bias vector.
`activity_regularizer`	Optional regularizer function for the output.
`kernel_constraint`	Optional projection function to be applied to the kernel after being updated by an `Optimizer` (e.g. used to implement norm constraints or value constraints for layer weights). The function must take as input the unprojected variable and must return the projected variable (which must have the same shape). Constraints are not safe to use when doing asynchronous distributed training.
`bias_constraint`	Optional projection function to be applied to the bias after being updated by an `Optimizer`.
`...`	For forward/backward compatability.

Value

A 4D tensor representing `activation(conv2d(inputs, kernel) + bias)`.

Input Shape

- If `data_format="channels_last"`: A 4D tensor with shape: `(batch_size, height, width, channels)`
- If `data_format="channels_first"`: A 4D tensor with shape: `(batch_size, channels, height, width)`

Output Shape

- If `data_format="channels_last"`: A 4D tensor with shape: `(batch_size, new_height, new_width, filters)`
- If `data_format="channels_first"`: A 4D tensor with shape: `(batch_size, filters, new_height, new_width)`

Raises

ValueError: when both `strides` > 1 and `dilation_rate` > 1.

Example

```
x <- random_uniform(c(4, 10, 10, 128))
y <- x |> layer_conv_2d(32, 3, activation='relu')
shape(y)

## shape(4, 8, 8, 32)
```

See Also

- https://keras.io/api/layers/convolution_layers/convolution2d#conv2d-class

Other convolutional layers:
`layer_conv_1d()`
`layer_conv_1d_transpose()`
`layer_conv_2d_transpose()`
`layer_conv_3d()`
`layer_conv_3d_transpose()`
`layer_depthwise_conv_1d()`
`layer_depthwise_conv_2d()`
`layer_separable_conv_1d()`
`layer_separable_conv_2d()`

Other layers:
`Layer()`
`layer_activation()`
`layer_activation_elu()`
`layer_activation_leaky_relu()`
`layer_activation_parametric_relu()`
`layer_activation_relu()`
`layer_activation_softmax()`
`layer_activity_regularization()`
`layer_add()`
`layer_additive_attention()`
`layer_alpha_dropout()`
`layer_attention()`
`layer_average()`
`layer_average_pooling_1d()`
`layer_average_pooling_2d()`
`layer_average_pooling_3d()`
`layer_batch_normalization()`
`layer_bidirectional()`
`layer_category_encoding()`
`layer_center_crop()`
`layer_concatenate()`

layer_conv_1d()
layer_conv_1d_transpose()
layer_conv_2d_transpose()
layer_conv_3d()
layer_conv_3d_transpose()
layer_conv_lstm_1d()
layer_conv_lstm_2d()
layer_conv_lstm_3d()
layer_cropping_1d()
layer_cropping_2d()
layer_cropping_3d()
layer_dense()
layer_depthwise_conv_1d()
layer_depthwise_conv_2d()
layer_discretization()
layer_dot()
layer_dropout()
layer_einsum_dense()
layer_embedding()
layer_feature_space()
layer_flatten()
layer_flax_module_wrapper()
layer_gaussian_dropout()
layer_gaussian_noise()
layer_global_average_pooling_1d()
layer_global_average_pooling_2d()
layer_global_average_pooling_3d()
layer_global_max_pooling_1d()
layer_global_max_pooling_2d()
layer_global_max_pooling_3d()
layer_group_normalization()
layer_group_query_attention()
layer_gru()
layer_hashed_crossing()
layer_hashing()
layer_identity()
layer_integer_lookup()
layer_jax_model_wrapper()
layer_lambda()
layer_layer_normalization()
layer_lstm()
layer_masking()
layer_max_pooling_1d()
layer_max_pooling_2d()
layer_max_pooling_3d()
layer_maximum()
layer_mel_spectrogram()
layer_minimum()

layer_multi_head_attention()
layer_multiply()
layer_normalization()
layer_permute()
layer_random_brightness()
layer_random_contrast()
layer_random_crop()
layer_random_flip()
layer_random_rotation()
layer_random_translation()
layer_random_zoom()
layer_repeat_vector()
layer_rescaling()
layer_reshape()
layer_resizing()
layer_rnn()
layer_separable_conv_1d()
layer_separable_conv_2d()
layer_simple_rnn()
layer_spatial_dropout_1d()
layer_spatial_dropout_2d()
layer_spatial_dropout_3d()
layer_spectral_normalization()
layer_string_lookup()
layer_subtract()
layer_text_vectorization()
layer_tfsm()
layer_time_distributed()
layer_torch_module_wrapper()
layer_unit_normalization()
layer_upsampling_1d()
layer_upsampling_2d()
layer_upsampling_3d()
layer_zero_padding_1d()
layer_zero_padding_2d()
layer_zero_padding_3d()
rnn_cell_gru()
rnn_cell_lstm()
rnn_cell_simple()
rnn_cells_stack()

layer_conv_2d_transpose *2D transposed convolution layer.*

Description

The need for transposed convolutions generally arise from the desire to use a transformation going in the opposite direction of a normal convolution, i.e., from something that has the shape of the output of some convolution to something that has the shape of its input while maintaining a connectivity pattern that is compatible with said convolution.

Usage

```
layer_conv_2d_transpose(
  object,
  filters,
  kernel_size,
  strides = list(1L, 1L),
  padding = "valid",
  data_format = NULL,
  dilation_rate = list(1L, 1L),
  activation = NULL,
  use_bias = TRUE,
  kernel_initializer = "glorot_uniform",
  bias_initializer = "zeros",
  kernel_regularizer = NULL,
  bias_regularizer = NULL,
  activity_regularizer = NULL,
  kernel_constraint = NULL,
  bias_constraint = NULL,
  ...
)
```

Arguments

`object`	Object to compose the layer with. A tensor, array, or sequential model.
`filters`	int, the dimension of the output space (the number of filters in the transposed convolution).
`kernel_size`	int or list of 1 integer, specifying the size of the transposed convolution window.
`strides`	int or list of 1 integer, specifying the stride length of the transposed convolution. `strides` > 1 is incompatible with `dilation_rate` > 1.
`padding`	string, either `"valid"` or `"same"` (case-insensitive). `"valid"` means no padding. `"same"` results in padding evenly to the left/right or up/down of the input. When `padding="same"` and `strides=1`, the output has the same size as the input.
`data_format`	string, either `"channels_last"` or `"channels_first"`. The ordering of the dimensions in the inputs. `"channels_last"` corresponds to inputs with shape `(batch_size, height, width, channels)` while `"channels_first"` corresponds to inputs with shape `(batch_size, channels, height, width)`. It defaults to the `image_data_format` value found in your Keras config file at `~/.keras/keras.json`. If you never set it, then it will be `"channels_last"`.
`dilation_rate`	int or list of 1 integers, specifying the dilation rate to use for dilated transposed convolution.

`activation`	Activation function. If `NULL`, no activation is applied.
`use_bias`	bool, if `TRUE`, bias will be added to the output.
`kernel_initializer`	Initializer for the convolution kernel. If `NULL`, the default initializer (`"glorot_uniform"`) will be used.
`bias_initializer`	Initializer for the bias vector. If `NULL`, the default initializer (`"zeros"`) will be used.
`kernel_regularizer`	Optional regularizer for the convolution kernel.
`bias_regularizer`	Optional regularizer for the bias vector.
`activity_regularizer`	Optional regularizer function for the output.
`kernel_constraint`	Optional projection function to be applied to the kernel after being updated by an `Optimizer` (e.g. used to implement norm constraints or value constraints for layer weights). The function must take as input the unprojected variable and must return the projected variable (which must have the same shape). Constraints are not safe to use when doing asynchronous distributed training.
`bias_constraint`	Optional projection function to be applied to the bias after being updated by an `Optimizer`.
`...`	For forward/backward compatability.

Value

A 4D tensor representing `activation(conv2d_transpose(inputs, kernel) + bias)`.

Input Shape

- If `data_format="channels_last"`: A 4D tensor with shape: `(batch_size, height, width, channels)`
- If `data_format="channels_first"`: A 4D tensor with shape: `(batch_size, channels, height, width)`

Output Shape

- If `data_format="channels_last"`: A 4D tensor with shape: `(batch_size, new_height, new_width, filters)`
- If `data_format="channels_first"`: A 4D tensor with shape: `(batch_size, filters, new_height, new_width)`

Raises

ValueError: when both `strides` > `1` and `dilation_rate` > `1`.

References

- A guide to convolution arithmetic for deep learning
- Deconvolutional Networks

Example

```
x <- random_uniform(c(4, 10, 8, 128))
y <- x |> layer_conv_2d_transpose(32, 2, 2, activation='relu')
shape(y)

## shape(4, 20, 16, 32)



# (4, 20, 16, 32)
```

See Also

- https://keras.io/api/layers/convolution_layers/convolution2d_transpose#conv2dtranspose-class

Other convolutional layers:
`layer_conv_1d()`
`layer_conv_1d_transpose()`
`layer_conv_2d()`
`layer_conv_3d()`
`layer_conv_3d_transpose()`
`layer_depthwise_conv_1d()`
`layer_depthwise_conv_2d()`
`layer_separable_conv_1d()`
`layer_separable_conv_2d()`

Other layers:
`Layer()`
`layer_activation()`
`layer_activation_elu()`
`layer_activation_leaky_relu()`
`layer_activation_parametric_relu()`
`layer_activation_relu()`
`layer_activation_softmax()`
`layer_activity_regularization()`
`layer_add()`
`layer_additive_attention()`
`layer_alpha_dropout()`
`layer_attention()`
`layer_average()`
`layer_average_pooling_1d()`
`layer_average_pooling_2d()`
`layer_average_pooling_3d()`
`layer_batch_normalization()`
`layer_bidirectional()`
`layer_category_encoding()`
`layer_center_crop()`
`layer_concatenate()`
`layer_conv_1d()`

layer_conv_1d_transpose()
layer_conv_2d()
layer_conv_3d()
layer_conv_3d_transpose()
layer_conv_lstm_1d()
layer_conv_lstm_2d()
layer_conv_lstm_3d()
layer_cropping_1d()
layer_cropping_2d()
layer_cropping_3d()
layer_dense()
layer_depthwise_conv_1d()
layer_depthwise_conv_2d()
layer_discretization()
layer_dot()
layer_dropout()
layer_einsum_dense()
layer_embedding()
layer_feature_space()
layer_flatten()
layer_flax_module_wrapper()
layer_gaussian_dropout()
layer_gaussian_noise()
layer_global_average_pooling_1d()
layer_global_average_pooling_2d()
layer_global_average_pooling_3d()
layer_global_max_pooling_1d()
layer_global_max_pooling_2d()
layer_global_max_pooling_3d()
layer_group_normalization()
layer_group_query_attention()
layer_gru()
layer_hashed_crossing()
layer_hashing()
layer_identity()
layer_integer_lookup()
layer_jax_model_wrapper()
layer_lambda()
layer_layer_normalization()
layer_lstm()
layer_masking()
layer_max_pooling_1d()
layer_max_pooling_2d()
layer_max_pooling_3d()
layer_maximum()
layer_mel_spectrogram()
layer_minimum()
layer_multi_head_attention()

layer_multiply()
layer_normalization()
layer_permute()
layer_random_brightness()
layer_random_contrast()
layer_random_crop()
layer_random_flip()
layer_random_rotation()
layer_random_translation()
layer_random_zoom()
layer_repeat_vector()
layer_rescaling()
layer_reshape()
layer_resizing()
layer_rnn()
layer_separable_conv_1d()
layer_separable_conv_2d()
layer_simple_rnn()
layer_spatial_dropout_1d()
layer_spatial_dropout_2d()
layer_spatial_dropout_3d()
layer_spectral_normalization()
layer_string_lookup()
layer_subtract()
layer_text_vectorization()
layer_tfsm()
layer_time_distributed()
layer_torch_module_wrapper()
layer_unit_normalization()
layer_upsampling_1d()
layer_upsampling_2d()
layer_upsampling_3d()
layer_zero_padding_1d()
layer_zero_padding_2d()
layer_zero_padding_3d()
rnn_cell_gru()
rnn_cell_lstm()
rnn_cell_simple()
rnn_cells_stack()

`layer_conv_3d` *3D convolution layer.*

Description

This layer creates a convolution kernel that is convolved with the layer input over a single spatial (or temporal) dimension to produce a tensor of outputs. If `use_bias` is `TRUE`, a bias vector is created and added to the outputs. Finally, if `activation` is not `NULL`, it is applied to the outputs as well.

Usage

```
layer_conv_3d(
  object,
  filters,
  kernel_size,
  strides = list(1L, 1L, 1L),
  padding = "valid",
  data_format = NULL,
  dilation_rate = list(1L, 1L, 1L),
  groups = 1L,
  activation = NULL,
  use_bias = TRUE,
  kernel_initializer = "glorot_uniform",
  bias_initializer = "zeros",
  kernel_regularizer = NULL,
  bias_regularizer = NULL,
  activity_regularizer = NULL,
  kernel_constraint = NULL,
  bias_constraint = NULL,
  ...
)
```

Arguments

`object`	Object to compose the layer with. A tensor, array, or sequential model.
`filters`	int, the dimension of the output space (the number of filters in the convolution).
`kernel_size`	int or list of 3 integer, specifying the size of the convolution window.
`strides`	int or list of 3 integer, specifying the stride length of the convolution. `strides > 1` is incompatible with `dilation_rate > 1`.
`padding`	string, either `"valid"` or `"same"` (case-insensitive). `"valid"` means no padding. `"same"` results in padding evenly to the left/right or up/down of the input. When `padding="same"` and `strides=1`, the output has the same size as the input.
`data_format`	string, either `"channels_last"` or `"channels_first"`. The ordering of the dimensions in the inputs. `"channels_last"` corresponds to inputs with shape `(batch_size, spatial_dim1, spatial_dim2, spatial_dim3, channels)` while `"channels_first"` corresponds to inputs with shape `(batch_size, channels, spatial_dim1, s` It defaults to the `image_data_format` value found in your Keras config file at `~/.keras/keras.json`. If you never set it, then it will be `"channels_last"`.
`dilation_rate`	int or list of 3 integers, specifying the dilation rate to use for dilated convolution.
`groups`	A positive int specifying the number of groups in which the input is split along the channel axis. Each group is convolved separately with `filters %/% groups` filters. The output is the concatenation of all the `groups` results along the channel axis. Input channels and `filters` must both be divisible by `groups`.
`activation`	Activation function. If `NULL`, no activation is applied.

`use_bias`	bool, if `TRUE`, bias will be added to the output.
`kernel_initializer`	Initializer for the convolution kernel. If `NULL`, the default initializer (`"glorot_uniform"`) will be used.
`bias_initializer`	Initializer for the bias vector. If `NULL`, the default initializer (`"zeros"`) will be used.
`kernel_regularizer`	Optional regularizer for the convolution kernel.
`bias_regularizer`	Optional regularizer for the bias vector.
`activity_regularizer`	Optional regularizer function for the output.
`kernel_constraint`	Optional projection function to be applied to the kernel after being updated by an `Optimizer` (e.g. used to implement norm constraints or value constraints for layer weights). The function must take as input the unprojected variable and must return the projected variable (which must have the same shape). Constraints are not safe to use when doing asynchronous distributed training.
`bias_constraint`	Optional projection function to be applied to the bias after being updated by an `Optimizer`.
`...`	For forward/backward compatability.

Value

A 5D tensor representing `activation(conv3d(inputs, kernel) + bias)`.

Input Shape

- If `data_format="channels_last"`: 5D tensor with shape: `(batch_size, spatial_dim1, spatial_dim2, spatial_`
- If `data_format="channels_first"`: 5D tensor with shape: `(batch_size, channels, spatial_dim1, spatial_di`

Output Shape

- If `data_format="channels_last"`: 5D tensor with shape: `(batch_size, new_spatial_dim1, new_spatial_dim2,`
- If `data_format="channels_first"`: 5D tensor with shape: `(batch_size, filters, new_spatial_dim1, new_spa`

Raises

ValueError: when both `strides > 1` and `dilation_rate > 1`.

Example

```
x <- random_uniform(c(4, 10, 10, 10, 128))
y <- x |> layer_conv_3d(32, 3, activation = 'relu')
shape(y)
```

```
## shape(4, 8, 8, 8, 32)
```

See Also

- https://keras.io/api/layers/convolution_layers/convolution3d#conv3d-class

Other convolutional layers:
`layer_conv_1d()`
`layer_conv_1d_transpose()`
`layer_conv_2d()`
`layer_conv_2d_transpose()`
`layer_conv_3d_transpose()`
`layer_depthwise_conv_1d()`
`layer_depthwise_conv_2d()`
`layer_separable_conv_1d()`
`layer_separable_conv_2d()`

Other layers:
`Layer()`
`layer_activation()`
`layer_activation_elu()`
`layer_activation_leaky_relu()`
`layer_activation_parametric_relu()`
`layer_activation_relu()`
`layer_activation_softmax()`
`layer_activity_regularization()`
`layer_add()`
`layer_additive_attention()`
`layer_alpha_dropout()`
`layer_attention()`
`layer_average()`
`layer_average_pooling_1d()`
`layer_average_pooling_2d()`
`layer_average_pooling_3d()`
`layer_batch_normalization()`
`layer_bidirectional()`
`layer_category_encoding()`
`layer_center_crop()`
`layer_concatenate()`
`layer_conv_1d()`
`layer_conv_1d_transpose()`
`layer_conv_2d()`
`layer_conv_2d_transpose()`
`layer_conv_3d_transpose()`
`layer_conv_lstm_1d()`
`layer_conv_lstm_2d()`
`layer_conv_lstm_3d()`
`layer_cropping_1d()`
`layer_cropping_2d()`
`layer_cropping_3d()`
`layer_dense()`

layer_depthwise_conv_1d()
layer_depthwise_conv_2d()
layer_discretization()
layer_dot()
layer_dropout()
layer_einsum_dense()
layer_embedding()
layer_feature_space()
layer_flatten()
layer_flax_module_wrapper()
layer_gaussian_dropout()
layer_gaussian_noise()
layer_global_average_pooling_1d()
layer_global_average_pooling_2d()
layer_global_average_pooling_3d()
layer_global_max_pooling_1d()
layer_global_max_pooling_2d()
layer_global_max_pooling_3d()
layer_group_normalization()
layer_group_query_attention()
layer_gru()
layer_hashed_crossing()
layer_hashing()
layer_identity()
layer_integer_lookup()
layer_jax_model_wrapper()
layer_lambda()
layer_layer_normalization()
layer_lstm()
layer_masking()
layer_max_pooling_1d()
layer_max_pooling_2d()
layer_max_pooling_3d()
layer_maximum()
layer_mel_spectrogram()
layer_minimum()
layer_multi_head_attention()
layer_multiply()
layer_normalization()
layer_permute()
layer_random_brightness()
layer_random_contrast()
layer_random_crop()
layer_random_flip()
layer_random_rotation()
layer_random_translation()
layer_random_zoom()
layer_repeat_vector()

layer_rescaling()
layer_reshape()
layer_resizing()
layer_rnn()
layer_separable_conv_1d()
layer_separable_conv_2d()
layer_simple_rnn()
layer_spatial_dropout_1d()
layer_spatial_dropout_2d()
layer_spatial_dropout_3d()
layer_spectral_normalization()
layer_string_lookup()
layer_subtract()
layer_text_vectorization()
layer_tfsm()
layer_time_distributed()
layer_torch_module_wrapper()
layer_unit_normalization()
layer_upsampling_1d()
layer_upsampling_2d()
layer_upsampling_3d()
layer_zero_padding_1d()
layer_zero_padding_2d()
layer_zero_padding_3d()
rnn_cell_gru()
rnn_cell_lstm()
rnn_cell_simple()
rnn_cells_stack()

---

layer_conv_3d_transpose

3D transposed convolution layer.

Description

The need for transposed convolutions generally arise from the desire to use a transformation going in the opposite direction of a normal convolution, i.e., from something that has the shape of the output of some convolution to something that has the shape of its input while maintaining a connectivity pattern that is compatible with said convolution.

Usage

```
layer_conv_3d_transpose(
  object,
  filters,
  kernel_size,
  strides = list(1L, 1L, 1L),
  padding = "valid",
  data_format = NULL,
  dilation_rate = list(1L, 1L, 1L),
  activation = NULL,
  use_bias = TRUE,
  kernel_initializer = "glorot_uniform",
  bias_initializer = "zeros",
  kernel_regularizer = NULL,
  bias_regularizer = NULL,
  activity_regularizer = NULL,
  kernel_constraint = NULL,
  bias_constraint = NULL,
  ...
)
```

Arguments

`object`	Object to compose the layer with. A tensor, array, or sequential model.
`filters`	int, the dimension of the output space (the number of filters in the transposed convolution).
`kernel_size`	int or list of 1 integer, specifying the size of the transposed convolution window.
`strides`	int or list of 1 integer, specifying the stride length of the transposed convolution. `strides` > 1 is incompatible with `dilation_rate` > 1.
`padding`	string, either `"valid"` or `"same"` (case-insensitive). `"valid"` means no padding. `"same"` results in padding evenly to the left/right or up/down of the input. When `padding="same"` and `strides=1`, the output has the same size as the input.
`data_format`	string, either `"channels_last"` or `"channels_first"`. The ordering of the dimensions in the inputs. `"channels_last"` corresponds to inputs with shape `(batch_size, spatial_dim1, spatial_dim2, spatial_dim3, channels)` while `"channels_first"` corresponds to inputs with shape `(batch_size, channels, spatial_dim1, s` It defaults to the `image_data_format` value found in your Keras config file at `~/.keras/keras.json`. If you never set it, then it will be `"channels_last"`.
`dilation_rate`	int or list of 1 integers, specifying the dilation rate to use for dilated transposed convolution.
`activation`	Activation function. If `NULL`, no activation is applied.
`use_bias`	bool, if `TRUE`, bias will be added to the output.
`kernel_initializer`	Initializer for the convolution kernel. If `NULL`, the default initializer (`"glorot_uniform"`) will be used.
`bias_initializer`	Initializer for the bias vector. If `NULL`, the default initializer (`"zeros"`) will be used.
`kernel_regularizer`	Optional regularizer for the convolution kernel.

`bias_regularizer`
Optional regularizer for the bias vector.

`activity_regularizer`
Optional regularizer function for the output.

`kernel_constraint`
Optional projection function to be applied to the kernel after being updated by an `Optimizer` (e.g. used to implement norm constraints or value constraints for layer weights). The function must take as input the unprojected variable and must return the projected variable (which must have the same shape). Constraints are not safe to use when doing asynchronous distributed training.

`bias_constraint`
Optional projection function to be applied to the bias after being updated by an `Optimizer`.

`...` For forward/backward compatability.

Value

A 5D tensor representing `activation(conv3d(inputs, kernel) + bias)`.

Input Shape

- If `data_format="channels_last"`: 5D tensor with shape: `(batch_size, spatial_dim1, spatial_dim2, spatial`
- If `data_format="channels_first"`: 5D tensor with shape: `(batch_size, channels, spatial_dim1, spatial_di`

Output Shape

- If `data_format="channels_last"`: 5D tensor with shape: `(batch_size, new_spatial_dim1, new_spatial_dim2,`
- If `data_format="channels_first"`: 5D tensor with shape: `(batch_size, filters, new_spatial_dim1, new_spa`

Raises

ValueError: when both `strides` > 1 and `dilation_rate` > 1.

References

- A guide to convolution arithmetic for deep learning
- Deconvolutional Networks

Example

```
x <- random_uniform(c(4, 10, 8, 12, 128))
y <- x |> layer_conv_3d_transpose(32, 2, 2, activation = 'relu')
shape(y)
```

```
## shape(4, 20, 16, 24, 32)
```

See Also

- https://keras.io/api/layers/convolution_layers/convolution3d_transpose#conv3dtranspose-class

Other convolutional layers:
layer_conv_1d()
layer_conv_1d_transpose()
layer_conv_2d()
layer_conv_2d_transpose()
layer_conv_3d()
layer_depthwise_conv_1d()
layer_depthwise_conv_2d()
layer_separable_conv_1d()
layer_separable_conv_2d()

Other layers:
Layer()
layer_activation()
layer_activation_elu()
layer_activation_leaky_relu()
layer_activation_parametric_relu()
layer_activation_relu()
layer_activation_softmax()
layer_activity_regularization()
layer_add()
layer_additive_attention()
layer_alpha_dropout()
layer_attention()
layer_average()
layer_average_pooling_1d()
layer_average_pooling_2d()
layer_average_pooling_3d()
layer_batch_normalization()
layer_bidirectional()
layer_category_encoding()
layer_center_crop()
layer_concatenate()
layer_conv_1d()
layer_conv_1d_transpose()
layer_conv_2d()
layer_conv_2d_transpose()
layer_conv_3d()
layer_conv_lstm_1d()
layer_conv_lstm_2d()
layer_conv_lstm_3d()
layer_cropping_1d()
layer_cropping_2d()
layer_cropping_3d()
layer_dense()

layer_depthwise_conv_1d()
layer_depthwise_conv_2d()
layer_discretization()
layer_dot()
layer_dropout()
layer_einsum_dense()
layer_embedding()
layer_feature_space()
layer_flatten()
layer_flax_module_wrapper()
layer_gaussian_dropout()
layer_gaussian_noise()
layer_global_average_pooling_1d()
layer_global_average_pooling_2d()
layer_global_average_pooling_3d()
layer_global_max_pooling_1d()
layer_global_max_pooling_2d()
layer_global_max_pooling_3d()
layer_group_normalization()
layer_group_query_attention()
layer_gru()
layer_hashed_crossing()
layer_hashing()
layer_identity()
layer_integer_lookup()
layer_jax_model_wrapper()
layer_lambda()
layer_layer_normalization()
layer_lstm()
layer_masking()
layer_max_pooling_1d()
layer_max_pooling_2d()
layer_max_pooling_3d()
layer_maximum()
layer_mel_spectrogram()
layer_minimum()
layer_multi_head_attention()
layer_multiply()
layer_normalization()
layer_permute()
layer_random_brightness()
layer_random_contrast()
layer_random_crop()
layer_random_flip()
layer_random_rotation()
layer_random_translation()
layer_random_zoom()
layer_repeat_vector()

layer_rescaling()
layer_reshape()
layer_resizing()
layer_rnn()
layer_separable_conv_1d()
layer_separable_conv_2d()
layer_simple_rnn()
layer_spatial_dropout_1d()
layer_spatial_dropout_2d()
layer_spatial_dropout_3d()
layer_spectral_normalization()
layer_string_lookup()
layer_subtract()
layer_text_vectorization()
layer_tfsm()
layer_time_distributed()
layer_torch_module_wrapper()
layer_unit_normalization()
layer_upsampling_1d()
layer_upsampling_2d()
layer_upsampling_3d()
layer_zero_padding_1d()
layer_zero_padding_2d()
layer_zero_padding_3d()
rnn_cell_gru()
rnn_cell_lstm()
rnn_cell_simple()
rnn_cells_stack()

layer_conv_lstm_1d *1D Convolutional LSTM.*

Description

Similar to an LSTM layer, but the input transformations and recurrent transformations are both convolutional.

Usage

```
layer_conv_lstm_1d(
  object,
  filters,
  kernel_size,
  strides = 1L,
  padding = "valid",
  data_format = NULL,
  dilation_rate = 1L,
  activation = "tanh",
  recurrent_activation = "sigmoid",
  use_bias = TRUE,
  kernel_initializer = "glorot_uniform",
  recurrent_initializer = "orthogonal",
  bias_initializer = "zeros",
  unit_forget_bias = TRUE,
  kernel_regularizer = NULL,
  recurrent_regularizer = NULL,
  bias_regularizer = NULL,
  activity_regularizer = NULL,
  kernel_constraint = NULL,
  recurrent_constraint = NULL,
  bias_constraint = NULL,
  dropout = 0,
  recurrent_dropout = 0,
  seed = NULL,
  return_sequences = FALSE,
  return_state = FALSE,
  go_backwards = FALSE,
  stateful = FALSE,
  ...,
  unroll = NULL
)
```

Arguments

`object`	Object to compose the layer with. A tensor, array, or sequential model.
`filters`	int, the dimension of the output space (the number of filters in the convolution).
`kernel_size`	int or tuple/list of 1 integer, specifying the size of the convolution window.
`strides`	int or tuple/list of 1 integer, specifying the stride length of the convolution. `strides` > 1 is incompatible with `dilation_rate` > 1.
`padding`	string, `"valid"` or `"same"` (case-insensitive). `"valid"` means no padding. `"same"` results in padding evenly to the left/right or up/down of the input such that output has the same height/width dimension as the input.
`data_format`	string, either `"channels_last"` or `"channels_first"`. The ordering of the dimensions in the inputs. `"channels_last"` corresponds to inputs with shape `(batch, steps, features)` while `"channels_first"` corresponds to inputs with shape `(batch, features, steps)`. It defaults to the `image_data_format` value found in your Keras config file at `~/.keras/keras.json`. If you never set it, then it will be `"channels_last"`.
`dilation_rate`	int or tuple/list of 1 integers, specifying the dilation rate to use for dilated convolution.
`activation`	Activation function to use. By default hyperbolic tangent activation function is applied (`tanh(x)`).

`recurrent_activation`	Activation function to use for the recurrent step.
`use_bias`	Boolean, whether the layer uses a bias vector.
`kernel_initializer`	Initializer for the `kernel` weights matrix, used for the linear transformation of the inputs.
`recurrent_initializer`	Initializer for the `recurrent_kernel` weights matrix, used for the linear transformation of the recurrent state.
`bias_initializer`	Initializer for the bias vector.
`unit_forget_bias`	Boolean. If `TRUE`, add 1 to the bias of the forget gate at initialization. Use in combination with `bias_initializer="zeros"`. This is recommended in Jozefowicz et al., 2015
`kernel_regularizer`	Regularizer function applied to the `kernel` weights matrix.
`recurrent_regularizer`	Regularizer function applied to the `recurrent_kernel` weights matrix.
`bias_regularizer`	Regularizer function applied to the bias vector.
`activity_regularizer`	Regularizer function applied to.
`kernel_constraint`	Constraint function applied to the `kernel` weights matrix.
`recurrent_constraint`	Constraint function applied to the `recurrent_kernel` weights matrix.
`bias_constraint`	Constraint function applied to the bias vector.
`dropout`	Float between 0 and 1. Fraction of the units to drop for the linear transformation of the inputs.
`recurrent_dropout`	Float between 0 and 1. Fraction of the units to drop for the linear transformation of the recurrent state.
`seed`	Random seed for dropout.
`return_sequences`	Boolean. Whether to return the last output in the output sequence, or the full sequence. Default: `FALSE`.
`return_state`	Boolean. Whether to return the last state in addition to the output. Default: `FALSE`.
`go_backwards`	Boolean (default: `FALSE`). If `TRUE`, process the input sequence backwards and return the reversed sequence.
`stateful`	Boolean (default `FALSE`). If `TRUE`, the last state for each sample at index i in a batch will be used as initial state for the sample of index i in the following batch.

`...`	For forward/backward compatability.
`unroll`	Boolean (default: `FALSE`). If `TRUE`, the network will be unrolled, else a symbolic loop will be used. Unrolling can speed-up a RNN, although it tends to be more memory-intensive. Unrolling is only suitable for short sequences.

Value

The return value depends on the value provided for the first argument. If `object` is:

- a `keras_model_sequential()`, then the layer is added to the sequential model (which is modified in place). To enable piping, the sequential model is also returned, invisibly.
- a `keras_input()`, then the output tensor from calling `layer(input)` is returned.
- `NULL` or missing, then a `Layer` instance is returned.

Call Arguments

- `inputs`: A 4D tensor.
- `initial_state`: List of initial state tensors to be passed to the first call of the cell.
- `mask`: Binary tensor of shape `(samples, timesteps)` indicating whether a given timestep should be masked.
- `training`: Python boolean indicating whether the layer should behave in training mode or in inference mode. This is only relevant if `dropout` or `recurrent_dropout` are set.

Input Shape

- If `data_format="channels_first"`: 4D tensor with shape: `(samples, time, channels, rows)`
- If `data_format="channels_last"`: 4D tensor with shape: `(samples, time, rows, channels)`

Output Shape

- If `return_state`: a list of tensors. The first tensor is the output. The remaining tensors are the last states, each 3D tensor with shape: `(samples, filters, new_rows)` if `data_format='channels_first'` or shape: `(samples, new_rows, filters)` if `data_format='channels_last'`. `rows` values might have changed due to padding.
- If `return_sequences`: 4D tensor with shape: `(samples, timesteps, filters, new_rows)` if data_format='channels_first' or shape: `(samples, timesteps, new_rows, filters)` if `data_format='channels_last'`.
- Else, 3D tensor with shape: `(samples, filters, new_rows)` if `data_format='channels_first'` or shape: `(samples, new_rows, filters)` if `data_format='channels_last'`.

References

- Shi et al., 2015 (the current implementation does not include the feedback loop on the cells output).

See Also

- https://keras.io/api/layers/recurrent_layers/conv_lstm1d#convlstm1d-class

Other rnn layers:
layer_bidirectional()
layer_conv_lstm_2d()
layer_conv_lstm_3d()
layer_gru()
layer_lstm()
layer_rnn()
layer_simple_rnn()
layer_time_distributed()
rnn_cell_gru()
rnn_cell_lstm()
rnn_cell_simple()
rnn_cells_stack()

Other layers:
Layer()
layer_activation()
layer_activation_elu()
layer_activation_leaky_relu()
layer_activation_parametric_relu()
layer_activation_relu()
layer_activation_softmax()
layer_activity_regularization()
layer_add()
layer_additive_attention()
layer_alpha_dropout()
layer_attention()
layer_average()
layer_average_pooling_1d()
layer_average_pooling_2d()
layer_average_pooling_3d()
layer_batch_normalization()
layer_bidirectional()
layer_category_encoding()
layer_center_crop()
layer_concatenate()
layer_conv_1d()
layer_conv_1d_transpose()
layer_conv_2d()
layer_conv_2d_transpose()
layer_conv_3d()
layer_conv_3d_transpose()
layer_conv_lstm_2d()
layer_conv_lstm_3d()
layer_cropping_1d()

layer_cropping_2d()
layer_cropping_3d()
layer_dense()
layer_depthwise_conv_1d()
layer_depthwise_conv_2d()
layer_discretization()
layer_dot()
layer_dropout()
layer_einsum_dense()
layer_embedding()
layer_feature_space()
layer_flatten()
layer_flax_module_wrapper()
layer_gaussian_dropout()
layer_gaussian_noise()
layer_global_average_pooling_1d()
layer_global_average_pooling_2d()
layer_global_average_pooling_3d()
layer_global_max_pooling_1d()
layer_global_max_pooling_2d()
layer_global_max_pooling_3d()
layer_group_normalization()
layer_group_query_attention()
layer_gru()
layer_hashed_crossing()
layer_hashing()
layer_identity()
layer_integer_lookup()
layer_jax_model_wrapper()
layer_lambda()
layer_layer_normalization()
layer_lstm()
layer_masking()
layer_max_pooling_1d()
layer_max_pooling_2d()
layer_max_pooling_3d()
layer_maximum()
layer_mel_spectrogram()
layer_minimum()
layer_multi_head_attention()
layer_multiply()
layer_normalization()
layer_permute()
layer_random_brightness()
layer_random_contrast()
layer_random_crop()
layer_random_flip()
layer_random_rotation()

layer_random_translation()
layer_random_zoom()
layer_repeat_vector()
layer_rescaling()
layer_reshape()
layer_resizing()
layer_rnn()
layer_separable_conv_1d()
layer_separable_conv_2d()
layer_simple_rnn()
layer_spatial_dropout_1d()
layer_spatial_dropout_2d()
layer_spatial_dropout_3d()
layer_spectral_normalization()
layer_string_lookup()
layer_subtract()
layer_text_vectorization()
layer_tfsm()
layer_time_distributed()
layer_torch_module_wrapper()
layer_unit_normalization()
layer_upsampling_1d()
layer_upsampling_2d()
layer_upsampling_3d()
layer_zero_padding_1d()
layer_zero_padding_2d()
layer_zero_padding_3d()
rnn_cell_gru()
rnn_cell_lstm()
rnn_cell_simple()
rnn_cells_stack()

layer_conv_lstm_2d *2D Convolutional LSTM.*

Description

Similar to an LSTM layer, but the input transformations and recurrent transformations are both convolutional.

Usage

```
layer_conv_lstm_2d(
  object,
  filters,
  kernel_size,
  strides = 1L,
  padding = "valid",
  data_format = NULL,
  dilation_rate = 1L,
  activation = "tanh",
  recurrent_activation = "sigmoid",
  use_bias = TRUE,
  kernel_initializer = "glorot_uniform",
  recurrent_initializer = "orthogonal",
  bias_initializer = "zeros",
  unit_forget_bias = TRUE,
  kernel_regularizer = NULL,
  recurrent_regularizer = NULL,
  bias_regularizer = NULL,
  activity_regularizer = NULL,
  kernel_constraint = NULL,
  recurrent_constraint = NULL,
  bias_constraint = NULL,
  dropout = 0,
  recurrent_dropout = 0,
  seed = NULL,
  return_sequences = FALSE,
  return_state = FALSE,
  go_backwards = FALSE,
  stateful = FALSE,
  ...,
  unroll = NULL
)
```

Arguments

`object`	Object to compose the layer with. A tensor, array, or sequential model.
`filters`	int, the dimension of the output space (the number of filters in the convolution).
`kernel_size`	int or tuple/list of 2 integers, specifying the size of the convolution window.
`strides`	int or tuple/list of 2 integers, specifying the stride length of the convolution. `strides` > 1 is incompatible with `dilation_rate` > 1.
`padding`	string, `"valid"` or `"same"` (case-insensitive). `"valid"` means no padding. `"same"` results in padding evenly to the left/right or up/down of the input such that output has the same height/width dimension as the input.
`data_format`	string, either `"channels_last"` or `"channels_first"`. The ordering of the dimensions in the inputs. `"channels_last"` corresponds to inputs with shape `(batch, steps, features)` while `"channels_first"` corresponds to inputs with shape `(batch, features, steps)`. It defaults to the `image_data_format` value found in your Keras config file at `~/.keras/keras.json`. If you never set it, then it will be `"channels_last"`.
`dilation_rate`	int or tuple/list of 2 integers, specifying the dilation rate to use for dilated convolution.

`activation`	Activation function to use. By default hyperbolic tangent activation function is applied (`tanh(x)`).
`recurrent_activation`	Activation function to use for the recurrent step.
`use_bias`	Boolean, whether the layer uses a bias vector.
`kernel_initializer`	Initializer for the `kernel` weights matrix, used for the linear transformation of the inputs.
`recurrent_initializer`	Initializer for the `recurrent_kernel` weights matrix, used for the linear transformation of the recurrent state.
`bias_initializer`	Initializer for the bias vector.
`unit_forget_bias`	Boolean. If `TRUE`, add 1 to the bias of the forget gate at initialization. Use in combination with `bias_initializer="zeros"`. This is recommended in Jozefowicz et al., 2015
`kernel_regularizer`	Regularizer function applied to the `kernel` weights matrix.
`recurrent_regularizer`	Regularizer function applied to the `recurrent_kernel` weights matrix.
`bias_regularizer`	Regularizer function applied to the bias vector.
`activity_regularizer`	Regularizer function applied to.
`kernel_constraint`	Constraint function applied to the `kernel` weights matrix.
`recurrent_constraint`	Constraint function applied to the `recurrent_kernel` weights matrix.
`bias_constraint`	Constraint function applied to the bias vector.
`dropout`	Float between 0 and 1. Fraction of the units to drop for the linear transformation of the inputs.
`recurrent_dropout`	Float between 0 and 1. Fraction of the units to drop for the linear transformation of the recurrent state.
`seed`	Random seed for dropout.
`return_sequences`	Boolean. Whether to return the last output in the output sequence, or the full sequence. Default: `FALSE`.
`return_state`	Boolean. Whether to return the last state in addition to the output. Default: `FALSE`.
`go_backwards`	Boolean (default: `FALSE`). If `TRUE`, process the input sequence backwards and return the reversed sequence.

`stateful`	Boolean (default FALSE). If `TRUE`, the last state for each sample at index i in a batch will be used as initial state for the sample of index i in the following batch.
`...`	For forward/backward compatability.
`unroll`	Boolean (default: `FALSE`). If `TRUE`, the network will be unrolled, else a symbolic loop will be used. Unrolling can speed-up a RNN, although it tends to be more memory-intensive. Unrolling is only suitable for short sequences.

Value

The return value depends on the value provided for the first argument. If `object` is:

- a `keras_model_sequential()`, then the layer is added to the sequential model (which is modified in place). To enable piping, the sequential model is also returned, invisibly.
- a `keras_input()`, then the output tensor from calling `layer(input)` is returned.
- `NULL` or missing, then a `Layer` instance is returned.

Call Arguments

- `inputs`: A 5D tensor.
- `mask`: Binary tensor of shape `(samples, timesteps)` indicating whether a given timestep should be masked.
- `training`: Python boolean indicating whether the layer should behave in training mode or in inference mode. This is only relevant if `dropout` or `recurrent_dropout` are set.
- `initial_state`: List of initial state tensors to be passed to the first call of the cell.

Input Shape

- If `data_format='channels_first'`: 5D tensor with shape: `(samples, time, channels, rows, cols)`
- If `data_format='channels_last'`: 5D tensor with shape: `(samples, time, rows, cols, channels)`

Output Shape

- If `return_state`: a list of tensors. The first tensor is the output. The remaining tensors are the last states, each 4D tensor with shape: `(samples, filters, new_rows, new_cols)` if `data_format='channels_first'` or shape: `(samples, new_rows, new_cols, filters)` if `data_format='channels_last'`. `rows` and `cols` values might have changed due to padding.
- If `return_sequences`: 5D tensor with shape: `(samples, timesteps, filters, new_rows, new_cols)` if data_format='channels_first' or shape: `(samples, timesteps, new_rows, new_cols, filters)` if `data_format='channels_last'`.
- Else, 4D tensor with shape: `(samples, filters, new_rows, new_cols)` if `data_format='channels_first'` or shape: `(samples, new_rows, new_cols, filters)` if `data_format='channels_last'`.

References

- Shi et al., 2015 (the current implementation does not include the feedback loop on the cells output).

See Also

- https://keras.io/api/layers/recurrent_layers/conv_lstm2d#convlstm2d-class

Other rnn layers:
layer_bidirectional()
layer_conv_lstm_1d()
layer_conv_lstm_3d()
layer_gru()
layer_lstm()
layer_rnn()
layer_simple_rnn()
layer_time_distributed()
rnn_cell_gru()
rnn_cell_lstm()
rnn_cell_simple()
rnn_cells_stack()

Other layers:
Layer()
layer_activation()
layer_activation_elu()
layer_activation_leaky_relu()
layer_activation_parametric_relu()
layer_activation_relu()
layer_activation_softmax()
layer_activity_regularization()
layer_add()
layer_additive_attention()
layer_alpha_dropout()
layer_attention()
layer_average()
layer_average_pooling_1d()
layer_average_pooling_2d()
layer_average_pooling_3d()
layer_batch_normalization()
layer_bidirectional()
layer_category_encoding()
layer_center_crop()
layer_concatenate()
layer_conv_1d()
layer_conv_1d_transpose()
layer_conv_2d()
layer_conv_2d_transpose()
layer_conv_3d()
layer_conv_3d_transpose()
layer_conv_lstm_1d()
layer_conv_lstm_3d()
layer_cropping_1d()

layer_cropping_2d()
layer_cropping_3d()
layer_dense()
layer_depthwise_conv_1d()
layer_depthwise_conv_2d()
layer_discretization()
layer_dot()
layer_dropout()
layer_einsum_dense()
layer_embedding()
layer_feature_space()
layer_flatten()
layer_flax_module_wrapper()
layer_gaussian_dropout()
layer_gaussian_noise()
layer_global_average_pooling_1d()
layer_global_average_pooling_2d()
layer_global_average_pooling_3d()
layer_global_max_pooling_1d()
layer_global_max_pooling_2d()
layer_global_max_pooling_3d()
layer_group_normalization()
layer_group_query_attention()
layer_gru()
layer_hashed_crossing()
layer_hashing()
layer_identity()
layer_integer_lookup()
layer_jax_model_wrapper()
layer_lambda()
layer_layer_normalization()
layer_lstm()
layer_masking()
layer_max_pooling_1d()
layer_max_pooling_2d()
layer_max_pooling_3d()
layer_maximum()
layer_mel_spectrogram()
layer_minimum()
layer_multi_head_attention()
layer_multiply()
layer_normalization()
layer_permute()
layer_random_brightness()
layer_random_contrast()
layer_random_crop()
layer_random_flip()
layer_random_rotation()

layer_random_translation()
layer_random_zoom()
layer_repeat_vector()
layer_rescaling()
layer_reshape()
layer_resizing()
layer_rnn()
layer_separable_conv_1d()
layer_separable_conv_2d()
layer_simple_rnn()
layer_spatial_dropout_1d()
layer_spatial_dropout_2d()
layer_spatial_dropout_3d()
layer_spectral_normalization()
layer_string_lookup()
layer_subtract()
layer_text_vectorization()
layer_tfsm()
layer_time_distributed()
layer_torch_module_wrapper()
layer_unit_normalization()
layer_upsampling_1d()
layer_upsampling_2d()
layer_upsampling_3d()
layer_zero_padding_1d()
layer_zero_padding_2d()
layer_zero_padding_3d()
rnn_cell_gru()
rnn_cell_lstm()
rnn_cell_simple()
rnn_cells_stack()

---

layer_conv_lstm_3d *3D Convolutional LSTM.*

Description

Similar to an LSTM layer, but the input transformations and recurrent transformations are both convolutional.

Usage

```
layer_conv_lstm_3d(
  object,
  filters,
  kernel_size,
  strides = 1L,
  padding = "valid",
  data_format = NULL,
  dilation_rate = 1L,
  activation = "tanh",
  recurrent_activation = "sigmoid",
  use_bias = TRUE,
  kernel_initializer = "glorot_uniform",
  recurrent_initializer = "orthogonal",
  bias_initializer = "zeros",
  unit_forget_bias = TRUE,
  kernel_regularizer = NULL,
  recurrent_regularizer = NULL,
  bias_regularizer = NULL,
  activity_regularizer = NULL,
  kernel_constraint = NULL,
  recurrent_constraint = NULL,
  bias_constraint = NULL,
  dropout = 0,
  recurrent_dropout = 0,
  seed = NULL,
  return_sequences = FALSE,
  return_state = FALSE,
  go_backwards = FALSE,
  stateful = FALSE,
  ...,
  unroll = NULL
)
```

Arguments

`object`	Object to compose the layer with. A tensor, array, or sequential model.
`filters`	int, the dimension of the output space (the number of filters in the convolution).
`kernel_size`	int or tuple/list of 3 integers, specifying the size of the convolution window.
`strides`	int or tuple/list of 3 integers, specifying the stride length of the convolution. `strides` > 1 is incompatible with `dilation_rate` > 1.
`padding`	string, `"valid"` or `"same"` (case-insensitive). `"valid"` means no padding. `"same"` results in padding evenly to the left/right or up/down of the input such that output has the same height/width dimension as the input.
`data_format`	string, either `"channels_last"` or `"channels_first"`. The ordering of the dimensions in the inputs. `"channels_last"` corresponds to inputs with shape `(batch, steps, features)` while `"channels_first"` corresponds to inputs with shape `(batch, features, steps)`. It defaults to the `image_data_format` value found in your Keras config file at `~/.keras/keras.json`. If you never set it, then it will be `"channels_last"`.
`dilation_rate`	int or tuple/list of 3 integers, specifying the dilation rate to use for dilated convolution.

`activation`	Activation function to use. By default hyperbolic tangent activation function is applied (`tanh(x)`).
`recurrent_activation`	Activation function to use for the recurrent step.
`use_bias`	Boolean, whether the layer uses a bias vector.
`kernel_initializer`	Initializer for the `kernel` weights matrix, used for the linear transformation of the inputs.
`recurrent_initializer`	Initializer for the `recurrent_kernel` weights matrix, used for the linear transformation of the recurrent state.
`bias_initializer`	Initializer for the bias vector.
`unit_forget_bias`	Boolean. If `TRUE`, add 1 to the bias of the forget gate at initialization. Use in combination with `bias_initializer="zeros"`. This is recommended in Jozefowicz et al., 2015
`kernel_regularizer`	Regularizer function applied to the `kernel` weights matrix.
`recurrent_regularizer`	Regularizer function applied to the `recurrent_kernel` weights matrix.
`bias_regularizer`	Regularizer function applied to the bias vector.
`activity_regularizer`	Regularizer function applied to.
`kernel_constraint`	Constraint function applied to the `kernel` weights matrix.
`recurrent_constraint`	Constraint function applied to the `recurrent_kernel` weights matrix.
`bias_constraint`	Constraint function applied to the bias vector.
`dropout`	Float between 0 and 1. Fraction of the units to drop for the linear transformation of the inputs.
`recurrent_dropout`	Float between 0 and 1. Fraction of the units to drop for the linear transformation of the recurrent state.
`seed`	Random seed for dropout.
`return_sequences`	Boolean. Whether to return the last output in the output sequence, or the full sequence. Default: `FALSE`.
`return_state`	Boolean. Whether to return the last state in addition to the output. Default: `FALSE`.
`go_backwards`	Boolean (default: `FALSE`). If `TRUE`, process the input sequence backwards and return the reversed sequence.

`stateful`	Boolean (default `FALSE`). If `TRUE`, the last state for each sample at index i in a batch will be used as initial state for the sample of index i in the following batch.
`...`	For forward/backward compatability.
`unroll`	Boolean (default: `FALSE`). If `TRUE`, the network will be unrolled, else a symbolic loop will be used. Unrolling can speed-up a RNN, although it tends to be more memory-intensive. Unrolling is only suitable for short sequences.

Value

The return value depends on the value provided for the first argument. If `object` is:

- a `keras_model_sequential()`, then the layer is added to the sequential model (which is modified in place). To enable piping, the sequential model is also returned, invisibly.
- a `keras_input()`, then the output tensor from calling `layer(input)` is returned.
- `NULL` or missing, then a `Layer` instance is returned.

Call Arguments

- `inputs`: A 6D tensor.
- `mask`: Binary tensor of shape `(samples, timesteps)` indicating whether a given timestep should be masked.
- `training`: Python boolean indicating whether the layer should behave in training mode or in inference mode. This is only relevant if `dropout` or `recurrent_dropout` are set.
- `initial_state`: List of initial state tensors to be passed to the first call of the cell.

Input Shape

- If `data_format='channels_first'`: 5D tensor with shape: `(samples, time, channels, *spatial_dims)`
- If `data_format='channels_last'`: 5D tensor with shape: `(samples, time, *spatial_dims, channels)`

Output Shape

- If `return_state`: a list of tensors. The first tensor is the output. The remaining tensors are the last states, each 4D tensor with shape: `(samples, filters, *spatial_dims)` if `data_format='channels_first'` or shape: `(samples, *spatial_dims, filters)` if `data_format='channels_last'`.
- If `return_sequences`: 5D tensor with shape: `(samples, timesteps, filters, *spatial_dims)` if data_format='channels_first' or shape: `(samples, timesteps, *spatial_dims, filters)` if `data_format='channels_last'`.
- Else, 4D tensor with shape: `(samples, filters, *spatial_dims)` if `data_format='channels_first'` or shape: `(samples, *spatial_dims, filters)` if `data_format='channels_last'`.

References

- Shi et al., 2015 (the current implementation does not include the feedback loop on the cells output).

See Also

- https://keras.io/api/layers/recurrent_layers/conv_lstm3d#convlstm3d-class

Other rnn layers:
layer_bidirectional()
layer_conv_lstm_1d()
layer_conv_lstm_2d()
layer_gru()
layer_lstm()
layer_rnn()
layer_simple_rnn()
layer_time_distributed()
rnn_cell_gru()
rnn_cell_lstm()
rnn_cell_simple()
rnn_cells_stack()

Other layers:
Layer()
layer_activation()
layer_activation_elu()
layer_activation_leaky_relu()
layer_activation_parametric_relu()
layer_activation_relu()
layer_activation_softmax()
layer_activity_regularization()
layer_add()
layer_additive_attention()
layer_alpha_dropout()
layer_attention()
layer_average()
layer_average_pooling_1d()
layer_average_pooling_2d()
layer_average_pooling_3d()
layer_batch_normalization()
layer_bidirectional()
layer_category_encoding()
layer_center_crop()
layer_concatenate()
layer_conv_1d()
layer_conv_1d_transpose()
layer_conv_2d()
layer_conv_2d_transpose()
layer_conv_3d()
layer_conv_3d_transpose()
layer_conv_lstm_1d()
layer_conv_lstm_2d()
layer_cropping_1d()

layer_cropping_2d()
layer_cropping_3d()
layer_dense()
layer_depthwise_conv_1d()
layer_depthwise_conv_2d()
layer_discretization()
layer_dot()
layer_dropout()
layer_einsum_dense()
layer_embedding()
layer_feature_space()
layer_flatten()
layer_flax_module_wrapper()
layer_gaussian_dropout()
layer_gaussian_noise()
layer_global_average_pooling_1d()
layer_global_average_pooling_2d()
layer_global_average_pooling_3d()
layer_global_max_pooling_1d()
layer_global_max_pooling_2d()
layer_global_max_pooling_3d()
layer_group_normalization()
layer_group_query_attention()
layer_gru()
layer_hashed_crossing()
layer_hashing()
layer_identity()
layer_integer_lookup()
layer_jax_model_wrapper()
layer_lambda()
layer_layer_normalization()
layer_lstm()
layer_masking()
layer_max_pooling_1d()
layer_max_pooling_2d()
layer_max_pooling_3d()
layer_maximum()
layer_mel_spectrogram()
layer_minimum()
layer_multi_head_attention()
layer_multiply()
layer_normalization()
layer_permute()
layer_random_brightness()
layer_random_contrast()
layer_random_crop()
layer_random_flip()
layer_random_rotation()

layer_random_translation()
layer_random_zoom()
layer_repeat_vector()
layer_rescaling()
layer_reshape()
layer_resizing()
layer_rnn()
layer_separable_conv_1d()
layer_separable_conv_2d()
layer_simple_rnn()
layer_spatial_dropout_1d()
layer_spatial_dropout_2d()
layer_spatial_dropout_3d()
layer_spectral_normalization()
layer_string_lookup()
layer_subtract()
layer_text_vectorization()
layer_tfsm()
layer_time_distributed()
layer_torch_module_wrapper()
layer_unit_normalization()
layer_upsampling_1d()
layer_upsampling_2d()
layer_upsampling_3d()
layer_zero_padding_1d()
layer_zero_padding_2d()
layer_zero_padding_3d()
rnn_cell_gru()
rnn_cell_lstm()
rnn_cell_simple()
rnn_cells_stack()

---

layer_cropping_1d *Cropping layer for 1D input (e.g. temporal sequence).*

Description

It crops along the time dimension (axis 2).

Usage

```
layer_cropping_1d(object, cropping = list(1L, 1L), ...)
```

Arguments

object	Object to compose the layer with. A tensor, array, or sequential model.

`cropping`	Int, or list of int (length 2). • If int: how many units should be trimmed off at the beginning and end of the cropping dimension (axis 1). • If list of 2 ints: how many units should be trimmed off at the beginning and end of the cropping dimension (`(left_crop, right_crop)`).
`...`	For forward/backward compatability.

Value

The return value depends on the value provided for the first argument. If `object` is:

- a `keras_model_sequential()`, then the layer is added to the sequential model (which is modified in place). To enable piping, the sequential model is also returned, invisibly.
- a `keras_input()`, then the output tensor from calling `layer(input)` is returned.
- `NULL` or missing, then a `Layer` instance is returned.

Example

```
input_shape <- c(2, 3, 2)
x <- op_arange(prod(input_shape)) |> op_reshape(input_shape)
x

## tf.Tensor(
## [[[ 0.  1.]
##   [ 2.  3.]
##   [ 4.  5.]]
##
##  [[ 6.  7.]
##   [ 8.  9.]
##   [10. 11.]]], shape=(2, 3, 2), dtype=float64)


y <- x |> layer_cropping_1d(cropping = 1)
y

## tf.Tensor(
## [[[2. 3.]]
##
##  [[8. 9.]]], shape=(2, 1, 2), dtype=float32)
```

Input Shape

3D tensor with shape `(batch_size, axis_to_crop, features)`

Output Shape

3D tensor with shape `(batch_size, cropped_axis, features)`

See Also

- https://keras.io/api/layers/reshaping_layers/cropping1d#cropping1d-class

Other reshaping layers:
layer_cropping_2d()
layer_cropping_3d()
layer_flatten()
layer_permute()
layer_repeat_vector()
layer_reshape()
layer_upsampling_1d()
layer_upsampling_2d()
layer_upsampling_3d()
layer_zero_padding_1d()
layer_zero_padding_2d()
layer_zero_padding_3d()

Other layers:
Layer()
layer_activation()
layer_activation_elu()
layer_activation_leaky_relu()
layer_activation_parametric_relu()
layer_activation_relu()
layer_activation_softmax()
layer_activity_regularization()
layer_add()
layer_additive_attention()
layer_alpha_dropout()
layer_attention()
layer_average()
layer_average_pooling_1d()
layer_average_pooling_2d()
layer_average_pooling_3d()
layer_batch_normalization()
layer_bidirectional()
layer_category_encoding()
layer_center_crop()
layer_concatenate()
layer_conv_1d()
layer_conv_1d_transpose()
layer_conv_2d()
layer_conv_2d_transpose()
layer_conv_3d()
layer_conv_3d_transpose()
layer_conv_lstm_1d()
layer_conv_lstm_2d()
layer_conv_lstm_3d()

layer_cropping_2d()
layer_cropping_3d()
layer_dense()
layer_depthwise_conv_1d()
layer_depthwise_conv_2d()
layer_discretization()
layer_dot()
layer_dropout()
layer_einsum_dense()
layer_embedding()
layer_feature_space()
layer_flatten()
layer_flax_module_wrapper()
layer_gaussian_dropout()
layer_gaussian_noise()
layer_global_average_pooling_1d()
layer_global_average_pooling_2d()
layer_global_average_pooling_3d()
layer_global_max_pooling_1d()
layer_global_max_pooling_2d()
layer_global_max_pooling_3d()
layer_group_normalization()
layer_group_query_attention()
layer_gru()
layer_hashed_crossing()
layer_hashing()
layer_identity()
layer_integer_lookup()
layer_jax_model_wrapper()
layer_lambda()
layer_layer_normalization()
layer_lstm()
layer_masking()
layer_max_pooling_1d()
layer_max_pooling_2d()
layer_max_pooling_3d()
layer_maximum()
layer_mel_spectrogram()
layer_minimum()
layer_multi_head_attention()
layer_multiply()
layer_normalization()
layer_permute()
layer_random_brightness()
layer_random_contrast()
layer_random_crop()
layer_random_flip()
layer_random_rotation()

layer_random_translation()
layer_random_zoom()
layer_repeat_vector()
layer_rescaling()
layer_reshape()
layer_resizing()
layer_rnn()
layer_separable_conv_1d()
layer_separable_conv_2d()
layer_simple_rnn()
layer_spatial_dropout_1d()
layer_spatial_dropout_2d()
layer_spatial_dropout_3d()
layer_spectral_normalization()
layer_string_lookup()
layer_subtract()
layer_text_vectorization()
layer_tfsm()
layer_time_distributed()
layer_torch_module_wrapper()
layer_unit_normalization()
layer_upsampling_1d()
layer_upsampling_2d()
layer_upsampling_3d()
layer_zero_padding_1d()
layer_zero_padding_2d()
layer_zero_padding_3d()
rnn_cell_gru()
rnn_cell_lstm()
rnn_cell_simple()
rnn_cells_stack()

---

layer_cropping_2d *Cropping layer for 2D input (e.g. picture).*

Description

It crops along spatial dimensions, i.e. height and width.

Usage

```
layer_cropping_2d(
  object,
  cropping = list(list(0L, 0L), list(0L, 0L)),
  data_format = NULL,
  ...
)
```

Arguments

`object`	Object to compose the layer with. A tensor, array, or sequential model.
`cropping`	Int, or list of 2 ints, or list of 2 lists of 2 ints. • If int: the same symmetric cropping is applied to height and width. • If list of 2 ints: interpreted as two different symmetric cropping values for height and width: `(symmetric_height_crop, symmetric_width_crop)`. • If list of 2 lists of 2 ints: interpreted as `((top_crop, bottom_crop), (left_crop, right_crop))`.
`data_format`	A string, one of `"channels_last"` (default) or `"channels_first"`. The ordering of the dimensions in the inputs. `"channels_last"` corresponds to inputs with shape `(batch_size, height, width, channels)` while `"channels_first"` corresponds to inputs with shape `(batch_size, channels, height, width)`. When unspecified, uses `image_data_format` value found in your Keras config file at `~/.keras/keras.json` (if exists). Defaults to `"channels_last"`.
`...`	For forward/backward compatability.

Value

The return value depends on the value provided for the first argument. If `object` is:

- a `keras_model_sequential()`, then the layer is added to the sequential model (which is modified in place). To enable piping, the sequential model is also returned, invisibly.
- a `keras_input()`, then the output tensor from calling `layer(input)` is returned.
- `NULL` or missing, then a `Layer` instance is returned.

Example

```
input_shape <- c(2, 28, 28, 3)
x <- op_arange(prod(input_shape), dtype ='int32') |> op_reshape(input_shape)
y <- x |> layer_cropping_2d(cropping=list(c(2, 2), c(4, 4)))
shape(y)

## shape(2, 24, 20, 3)
```

Input Shape

4D tensor with shape:

- If `data_format` is `"channels_last"`: `(batch_size, height, width, channels)`
- If `data_format` is `"channels_first"`: `(batch_size, channels, height, width)`

Output Shape

4D tensor with shape:

- If `data_format` is `"channels_last"`: `(batch_size, cropped_height, cropped_width, channels)`
- If `data_format` is `"channels_first"`: `(batch_size, channels, cropped_height, cropped_width)`

See Also

- https://keras.io/api/layers/reshaping_layers/cropping2d#cropping2d-class

Other reshaping layers:
layer_cropping_1d()
layer_cropping_3d()
layer_flatten()
layer_permute()
layer_repeat_vector()
layer_reshape()
layer_upsampling_1d()
layer_upsampling_2d()
layer_upsampling_3d()
layer_zero_padding_1d()
layer_zero_padding_2d()
layer_zero_padding_3d()

Other layers:
Layer()
layer_activation()
layer_activation_elu()
layer_activation_leaky_relu()
layer_activation_parametric_relu()
layer_activation_relu()
layer_activation_softmax()
layer_activity_regularization()
layer_add()
layer_additive_attention()
layer_alpha_dropout()
layer_attention()
layer_average()
layer_average_pooling_1d()
layer_average_pooling_2d()
layer_average_pooling_3d()
layer_batch_normalization()
layer_bidirectional()
layer_category_encoding()
layer_center_crop()
layer_concatenate()
layer_conv_1d()
layer_conv_1d_transpose()
layer_conv_2d()
layer_conv_2d_transpose()
layer_conv_3d()
layer_conv_3d_transpose()
layer_conv_lstm_1d()
layer_conv_lstm_2d()
layer_conv_lstm_3d()

layer_cropping_1d()
layer_cropping_3d()
layer_dense()
layer_depthwise_conv_1d()
layer_depthwise_conv_2d()
layer_discretization()
layer_dot()
layer_dropout()
layer_einsum_dense()
layer_embedding()
layer_feature_space()
layer_flatten()
layer_flax_module_wrapper()
layer_gaussian_dropout()
layer_gaussian_noise()
layer_global_average_pooling_1d()
layer_global_average_pooling_2d()
layer_global_average_pooling_3d()
layer_global_max_pooling_1d()
layer_global_max_pooling_2d()
layer_global_max_pooling_3d()
layer_group_normalization()
layer_group_query_attention()
layer_gru()
layer_hashed_crossing()
layer_hashing()
layer_identity()
layer_integer_lookup()
layer_jax_model_wrapper()
layer_lambda()
layer_layer_normalization()
layer_lstm()
layer_masking()
layer_max_pooling_1d()
layer_max_pooling_2d()
layer_max_pooling_3d()
layer_maximum()
layer_mel_spectrogram()
layer_minimum()
layer_multi_head_attention()
layer_multiply()
layer_normalization()
layer_permute()
layer_random_brightness()
layer_random_contrast()
layer_random_crop()
layer_random_flip()
layer_random_rotation()

layer_random_translation()
layer_random_zoom()
layer_repeat_vector()
layer_rescaling()
layer_reshape()
layer_resizing()
layer_rnn()
layer_separable_conv_1d()
layer_separable_conv_2d()
layer_simple_rnn()
layer_spatial_dropout_1d()
layer_spatial_dropout_2d()
layer_spatial_dropout_3d()
layer_spectral_normalization()
layer_string_lookup()
layer_subtract()
layer_text_vectorization()
layer_tfsm()
layer_time_distributed()
layer_torch_module_wrapper()
layer_unit_normalization()
layer_upsampling_1d()
layer_upsampling_2d()
layer_upsampling_3d()
layer_zero_padding_1d()
layer_zero_padding_2d()
layer_zero_padding_3d()
rnn_cell_gru()
rnn_cell_lstm()
rnn_cell_simple()
rnn_cells_stack()

---

layer_cropping_3d *Cropping layer for 3D data (e.g. spatial or spatio-temporal).*

Description

Cropping layer for 3D data (e.g. spatial or spatio-temporal).

Usage

```
layer_cropping_3d(
  object,
  cropping = list(list(1L, 1L), list(1L, 1L), list(1L, 1L)),
  data_format = NULL,
  ...
)
```

Arguments

`object`	Object to compose the layer with. A tensor, array, or sequential model.
`cropping`	Int, or list of 3 ints, or list of 3 lists of 2 ints. • If int: the same symmetric cropping is applied to depth, height, and width. • If list of 3 ints: interpreted as three different symmetric cropping values for depth, height, and width: `(symmetric_dim1_crop, symmetric_dim2_crop, symmetric_dim3_cro` • If list of 3 lists of 2 ints: interpreted as `((left_dim1_crop, right_dim1_crop), (left_dim2_crop`
`data_format`	A string, one of `"channels_last"` (default) or `"channels_first"`. The ordering of the dimensions in the inputs. `"channels_last"` corresponds to inputs with shape `(batch_size, spatial_dim1, spatial_dim2, spatial_dim3, channels)` while `"channels_first"` corresponds to inputs with shape `(batch_size, channels, spatial_dim1, s` When unspecified, uses `image_data_format` value found in your Keras config file at `~/.keras/keras.json` (if exists). Defaults to `"channels_last"`.
`...`	For forward/backward compatability.

Value

The return value depends on the value provided for the first argument. If `object` is:

- a `keras_model_sequential()`, then the layer is added to the sequential model (which is modified in place). To enable piping, the sequential model is also returned, invisibly.
- a `keras_input()`, then the output tensor from calling `layer(input)` is returned.
- `NULL` or missing, then a `Layer` instance is returned.

Example

```
input_shape <- c(2, 28, 28, 10, 3)
x <- input_shape %>% { op_reshape(seq(prod(.)), .) }
y <- x |> layer_cropping_3d(cropping = c(2, 4, 2))
shape(y)

## shape(2, 24, 20, 6, 3)
```

Input Shape

5D tensor with shape:

- If `data_format` is `"channels_last"`: `(batch_size, first_axis_to_crop, second_axis_to_crop, third_axis_`
- If `data_format` is `"channels_first"`: `(batch_size, channels, first_axis_to_crop, second_axis_to_crop,`

Output Shape

5D tensor with shape:

- If `data_format` is `"channels_last"`: `(batch_size, first_cropped_axis, second_cropped_axis, third_cropp`
- If `data_format` is `"channels_first"`: `(batch_size, channels, first_cropped_axis, second_cropped_axis,`

See Also

- https://keras.io/api/layers/reshaping_layers/cropping3d#cropping3d-class

Other reshaping layers:
layer_cropping_1d()
layer_cropping_2d()
layer_flatten()
layer_permute()
layer_repeat_vector()
layer_reshape()
layer_upsampling_1d()
layer_upsampling_2d()
layer_upsampling_3d()
layer_zero_padding_1d()
layer_zero_padding_2d()
layer_zero_padding_3d()

Other layers:
Layer()
layer_activation()
layer_activation_elu()
layer_activation_leaky_relu()
layer_activation_parametric_relu()
layer_activation_relu()
layer_activation_softmax()
layer_activity_regularization()
layer_add()
layer_additive_attention()
layer_alpha_dropout()
layer_attention()
layer_average()
layer_average_pooling_1d()
layer_average_pooling_2d()
layer_average_pooling_3d()
layer_batch_normalization()
layer_bidirectional()
layer_category_encoding()
layer_center_crop()
layer_concatenate()
layer_conv_1d()
layer_conv_1d_transpose()
layer_conv_2d()
layer_conv_2d_transpose()
layer_conv_3d()
layer_conv_3d_transpose()
layer_conv_lstm_1d()
layer_conv_lstm_2d()
layer_conv_lstm_3d()

layer_cropping_1d()
layer_cropping_2d()
layer_dense()
layer_depthwise_conv_1d()
layer_depthwise_conv_2d()
layer_discretization()
layer_dot()
layer_dropout()
layer_einsum_dense()
layer_embedding()
layer_feature_space()
layer_flatten()
layer_flax_module_wrapper()
layer_gaussian_dropout()
layer_gaussian_noise()
layer_global_average_pooling_1d()
layer_global_average_pooling_2d()
layer_global_average_pooling_3d()
layer_global_max_pooling_1d()
layer_global_max_pooling_2d()
layer_global_max_pooling_3d()
layer_group_normalization()
layer_group_query_attention()
layer_gru()
layer_hashed_crossing()
layer_hashing()
layer_identity()
layer_integer_lookup()
layer_jax_model_wrapper()
layer_lambda()
layer_layer_normalization()
layer_lstm()
layer_masking()
layer_max_pooling_1d()
layer_max_pooling_2d()
layer_max_pooling_3d()
layer_maximum()
layer_mel_spectrogram()
layer_minimum()
layer_multi_head_attention()
layer_multiply()
layer_normalization()
layer_permute()
layer_random_brightness()
layer_random_contrast()
layer_random_crop()
layer_random_flip()
layer_random_rotation()

layer_random_translation()
layer_random_zoom()
layer_repeat_vector()
layer_rescaling()
layer_reshape()
layer_resizing()
layer_rnn()
layer_separable_conv_1d()
layer_separable_conv_2d()
layer_simple_rnn()
layer_spatial_dropout_1d()
layer_spatial_dropout_2d()
layer_spatial_dropout_3d()
layer_spectral_normalization()
layer_string_lookup()
layer_subtract()
layer_text_vectorization()
layer_tfsm()
layer_time_distributed()
layer_torch_module_wrapper()
layer_unit_normalization()
layer_upsampling_1d()
layer_upsampling_2d()
layer_upsampling_3d()
layer_zero_padding_1d()
layer_zero_padding_2d()
layer_zero_padding_3d()
rnn_cell_gru()
rnn_cell_lstm()
rnn_cell_simple()
rnn_cells_stack()

---

layer_dense *Just your regular densely-connected NN layer.*

Description

Dense implements the operation: output = activation(dot(input, kernel) + bias) where activation is the element-wise activation function passed as the activation argument, kernel is a weights matrix created by the layer, and bias is a bias vector created by the layer (only applicable if use_bias is TRUE).

Usage

```
layer_dense(
  object,
  units,
  activation = NULL,
  use_bias = TRUE,
  kernel_initializer = "glorot_uniform",
  bias_initializer = "zeros",
  kernel_regularizer = NULL,
  bias_regularizer = NULL,
  activity_regularizer = NULL,
  kernel_constraint = NULL,
  bias_constraint = NULL,
  lora_rank = NULL,
  ...
)
```

Arguments

`object`	Object to compose the layer with. A tensor, array, or sequential model.
`units`	Positive integer, dimensionality of the output space.
`activation`	Activation function to use. If you don't specify anything, no activation is applied (ie. "linear" activation: `a(x) = x`).
`use_bias`	Boolean, whether the layer uses a bias vector.
`kernel_initializer`	Initializer for the `kernel` weights matrix.
`bias_initializer`	Initializer for the bias vector.
`kernel_regularizer`	Regularizer function applied to the `kernel` weights matrix.
`bias_regularizer`	Regularizer function applied to the bias vector.
`activity_regularizer`	Regularizer function applied to the output of the layer (its "activation").
`kernel_constraint`	Constraint function applied to the `kernel` weights matrix.
`bias_constraint`	Constraint function applied to the bias vector.
`lora_rank`	Optional integer. If set, the layer's forward pass will implement LoRA (Low-Rank Adaptation) with the provided rank. LoRA sets the layer's kernel to non-trainable and replaces it with a delta over the original kernel, obtained via multiplying two lower-rank trainable matrices. This can be useful to reduce the computation cost of fine-tuning large dense layers. You can also enable LoRA on an existing `Dense` layer by calling `layer$enable_lora(rank)`.
`...`	For forward/backward compatability.

Value

The return value depends on the value provided for the first argument. If `object` is:

- a `keras_model_sequential()`, then the layer is added to the sequential model (which is modified in place). To enable piping, the sequential model is also returned, invisibly.
- a `keras_input()`, then the output tensor from calling `layer(input)` is returned.
- `NULL` or missing, then a `Layer` instance is returned.

Note

If the input to the layer has a rank greater than 2, `Dense` computes the dot product between the `inputs` and the `kernel` along the last axis of the `inputs` and axis 0 of the `kernel` (using `tf.tensordot`). For example, if input has dimensions `(batch_size, d0, d1)`, then we create a `kernel` with shape `(d1, units)`, and the `kernel` operates along axis 2 of the `input`, on every sub-tensor of shape `(1, 1, d1)` (there are `batch_size * d0` such sub-tensors). The output in this case will have shape `(batch_size, d0, units)`.

Input Shape

N-D tensor with shape: `(batch_size, ..., input_dim)`. The most common situation would be a 2D input with shape `(batch_size, input_dim)`.

Output Shape

N-D tensor with shape: `(batch_size, ..., units)`. For instance, for a 2D input with shape `(batch_size, input_dim)`, the output would have shape `(batch_size, units)`.

Methods

-
  ```
  enable_lora(
    rank,
    a_initializer = 'he_uniform',
    b_initializer = 'zeros'
  )
  ```

Readonly properties:

- `kernel`

See Also

- https://keras.io/api/layers/core_layers/dense#dense-class

Other core layers:
`layer_einsum_dense()`
`layer_embedding()`
`layer_identity()`
`layer_lambda()`
`layer_masking()`

Other layers:
`Layer()`
`layer_activation()`

layer_activation_elu()
layer_activation_leaky_relu()
layer_activation_parametric_relu()
layer_activation_relu()
layer_activation_softmax()
layer_activity_regularization()
layer_add()
layer_additive_attention()
layer_alpha_dropout()
layer_attention()
layer_average()
layer_average_pooling_1d()
layer_average_pooling_2d()
layer_average_pooling_3d()
layer_batch_normalization()
layer_bidirectional()
layer_category_encoding()
layer_center_crop()
layer_concatenate()
layer_conv_1d()
layer_conv_1d_transpose()
layer_conv_2d()
layer_conv_2d_transpose()
layer_conv_3d()
layer_conv_3d_transpose()
layer_conv_lstm_1d()
layer_conv_lstm_2d()
layer_conv_lstm_3d()
layer_cropping_1d()
layer_cropping_2d()
layer_cropping_3d()
layer_depthwise_conv_1d()
layer_depthwise_conv_2d()
layer_discretization()
layer_dot()
layer_dropout()
layer_einsum_dense()
layer_embedding()
layer_feature_space()
layer_flatten()
layer_flax_module_wrapper()
layer_gaussian_dropout()
layer_gaussian_noise()
layer_global_average_pooling_1d()
layer_global_average_pooling_2d()
layer_global_average_pooling_3d()
layer_global_max_pooling_1d()
layer_global_max_pooling_2d()

layer_global_max_pooling_3d()
layer_group_normalization()
layer_group_query_attention()
layer_gru()
layer_hashed_crossing()
layer_hashing()
layer_identity()
layer_integer_lookup()
layer_jax_model_wrapper()
layer_lambda()
layer_layer_normalization()
layer_lstm()
layer_masking()
layer_max_pooling_1d()
layer_max_pooling_2d()
layer_max_pooling_3d()
layer_maximum()
layer_mel_spectrogram()
layer_minimum()
layer_multi_head_attention()
layer_multiply()
layer_normalization()
layer_permute()
layer_random_brightness()
layer_random_contrast()
layer_random_crop()
layer_random_flip()
layer_random_rotation()
layer_random_translation()
layer_random_zoom()
layer_repeat_vector()
layer_rescaling()
layer_reshape()
layer_resizing()
layer_rnn()
layer_separable_conv_1d()
layer_separable_conv_2d()
layer_simple_rnn()
layer_spatial_dropout_1d()
layer_spatial_dropout_2d()
layer_spatial_dropout_3d()
layer_spectral_normalization()
layer_string_lookup()
layer_subtract()
layer_text_vectorization()
layer_tfsm()
layer_time_distributed()
layer_torch_module_wrapper()

layer_unit_normalization()
layer_upsampling_1d()
layer_upsampling_2d()
layer_upsampling_3d()
layer_zero_padding_1d()
layer_zero_padding_2d()
layer_zero_padding_3d()
rnn_cell_gru()
rnn_cell_lstm()
rnn_cell_simple()
rnn_cells_stack()

layer_depthwise_conv_1d

1D depthwise convolution layer.

Description

Depthwise convolution is a type of convolution in which each input channel is convolved with a different kernel (called a depthwise kernel). You can understand depthwise convolution as the first step in a depthwise separable convolution.

It is implemented via the following steps:

- Split the input into individual channels.
- Convolve each channel with an individual depthwise kernel with depth_multiplier output channels.
- Concatenate the convolved outputs along the channels axis.

Unlike a regular 1D convolution, depthwise convolution does not mix information across different input channels.

The depth_multiplier argument determines how many filters are applied to one input channel. As such, it controls the amount of output channels that are generated per input channel in the depthwise step.

Usage

```
layer_depthwise_conv_1d(
  object,
  kernel_size,
  strides = 1L,
  padding = "valid",
  depth_multiplier = 1L,
  data_format = NULL,
  dilation_rate = 1L,
  activation = NULL,
  use_bias = TRUE,
  depthwise_initializer = "glorot_uniform",
  bias_initializer = "zeros",
  depthwise_regularizer = NULL,
  bias_regularizer = NULL,
  activity_regularizer = NULL,
  depthwise_constraint = NULL,
  bias_constraint = NULL,
  ...
)
```

Arguments

`object`	Object to compose the layer with. A tensor, array, or sequential model.
`kernel_size`	int or list of 1 integer, specifying the size of the depthwise convolution window.
`strides`	int or list of 1 integer, specifying the stride length of the convolution. `strides` > 1 is incompatible with `dilation_rate` > 1.
`padding`	string, either `"valid"` or `"same"` (case-insensitive). `"valid"` means no padding. `"same"` results in padding evenly to the left/right or up/down of the input. When `padding="same"` and `strides=1`, the output has the same size as the input.
`depth_multiplier`	The number of depthwise convolution output channels for each input channel. The total number of depthwise convolution output channels will be equal to `input_channel * depth_multiplier`.
`data_format`	string, either `"channels_last"` or `"channels_first"`. The ordering of the dimensions in the inputs. `"channels_last"` corresponds to inputs with shape `(batch, steps, features)` while `"channels_first"` corresponds to inputs with shape `(batch, features, steps)`. It defaults to the `image_data_format` value found in your Keras config file at `~/.keras/keras.json`. If you never set it, then it will be `"channels_last"`.
`dilation_rate`	int or list of 1 integers, specifying the dilation rate to use for dilated convolution.
`activation`	Activation function. If `NULL`, no activation is applied.
`use_bias`	bool, if `TRUE`, bias will be added to the output.
`depthwise_initializer`	Initializer for the convolution kernel. If `NULL`, the default initializer (`"glorot_uniform"`) will be used.
`bias_initializer`	Initializer for the bias vector. If `NULL`, the default initializer (`"zeros"`) will be used.
`depthwise_regularizer`	Optional regularizer for the convolution kernel.
`bias_regularizer`	Optional regularizer for the bias vector.
`activity_regularizer`	Optional regularizer function for the output.

`depthwise_constraint`	Optional projection function to be applied to the kernel after being updated by an `Optimizer` (e.g. used to implement norm constraints or value constraints for layer weights). The function must take as input the unprojected variable and must return the projected variable (which must have the same shape). Constraints are not safe to use when doing asynchronous distributed training.
`bias_constraint`	Optional projection function to be applied to the bias after being updated by an `Optimizer`.
`...`	For forward/backward compatability.

Value

A 3D tensor representing `activation(depthwise_conv1d(inputs, kernel) + bias)`.

Input Shape

- If `data_format="channels_last"`: A 3D tensor with shape: `(batch_shape, steps, channels)`
- If `data_format="channels_first"`: A 3D tensor with shape: `(batch_shape, channels, steps)`

Output Shape

- If `data_format="channels_last"`: A 3D tensor with shape: `(batch_shape, new_steps, channels * depth_multi`
- If `data_format="channels_first"`: A 3D tensor with shape: `(batch_shape, channels * depth_multiplier, new`

Raises

ValueError: when both `strides` > 1 and `dilation_rate` > 1.

Example

```
x <- random_uniform(c(4, 10, 12))
y <- x |> layer_depthwise_conv_1d(
  kernel_size = 3,
  depth_multiplier = 3,
  activation = 'relu'
)
shape(y)

## shape(4, 8, 36)
```

See Also

Other convolutional layers:
`layer_conv_1d()`
`layer_conv_1d_transpose()`
`layer_conv_2d()`
`layer_conv_2d_transpose()`

layer_conv_3d()
layer_conv_3d_transpose()
layer_depthwise_conv_2d()
layer_separable_conv_1d()
layer_separable_conv_2d()

Other layers:
Layer()
layer_activation()
layer_activation_elu()
layer_activation_leaky_relu()
layer_activation_parametric_relu()
layer_activation_relu()
layer_activation_softmax()
layer_activity_regularization()
layer_add()
layer_additive_attention()
layer_alpha_dropout()
layer_attention()
layer_average()
layer_average_pooling_1d()
layer_average_pooling_2d()
layer_average_pooling_3d()
layer_batch_normalization()
layer_bidirectional()
layer_category_encoding()
layer_center_crop()
layer_concatenate()
layer_conv_1d()
layer_conv_1d_transpose()
layer_conv_2d()
layer_conv_2d_transpose()
layer_conv_3d()
layer_conv_3d_transpose()
layer_conv_lstm_1d()
layer_conv_lstm_2d()
layer_conv_lstm_3d()
layer_cropping_1d()
layer_cropping_2d()
layer_cropping_3d()
layer_dense()
layer_depthwise_conv_2d()
layer_discretization()
layer_dot()
layer_dropout()
layer_einsum_dense()
layer_embedding()
layer_feature_space()

layer_flatten()
layer_flax_module_wrapper()
layer_gaussian_dropout()
layer_gaussian_noise()
layer_global_average_pooling_1d()
layer_global_average_pooling_2d()
layer_global_average_pooling_3d()
layer_global_max_pooling_1d()
layer_global_max_pooling_2d()
layer_global_max_pooling_3d()
layer_group_normalization()
layer_group_query_attention()
layer_gru()
layer_hashed_crossing()
layer_hashing()
layer_identity()
layer_integer_lookup()
layer_jax_model_wrapper()
layer_lambda()
layer_layer_normalization()
layer_lstm()
layer_masking()
layer_max_pooling_1d()
layer_max_pooling_2d()
layer_max_pooling_3d()
layer_maximum()
layer_mel_spectrogram()
layer_minimum()
layer_multi_head_attention()
layer_multiply()
layer_normalization()
layer_permute()
layer_random_brightness()
layer_random_contrast()
layer_random_crop()
layer_random_flip()
layer_random_rotation()
layer_random_translation()
layer_random_zoom()
layer_repeat_vector()
layer_rescaling()
layer_reshape()
layer_resizing()
layer_rnn()
layer_separable_conv_1d()
layer_separable_conv_2d()
layer_simple_rnn()
layer_spatial_dropout_1d()

`layer_spatial_dropout_2d()`
`layer_spatial_dropout_3d()`
`layer_spectral_normalization()`
`layer_string_lookup()`
`layer_subtract()`
`layer_text_vectorization()`
`layer_tfsm()`
`layer_time_distributed()`
`layer_torch_module_wrapper()`
`layer_unit_normalization()`
`layer_upsampling_1d()`
`layer_upsampling_2d()`
`layer_upsampling_3d()`
`layer_zero_padding_1d()`
`layer_zero_padding_2d()`
`layer_zero_padding_3d()`
`rnn_cell_gru()`
`rnn_cell_lstm()`
`rnn_cell_simple()`
`rnn_cells_stack()`

---

layer_depthwise_conv_2d

2D depthwise convolution layer.

Description

Depthwise convolution is a type of convolution in which each input channel is convolved with a different kernel (called a depthwise kernel). You can understand depthwise convolution as the first step in a depthwise separable convolution.

It is implemented via the following steps:

- Split the input into individual channels.
- Convolve each channel with an individual depthwise kernel with `depth_multiplier` output channels.
- Concatenate the convolved outputs along the channels axis.

Unlike a regular 2D convolution, depthwise convolution does not mix information across different input channels.

The `depth_multiplier` argument determines how many filters are applied to one input channel. As such, it controls the amount of output channels that are generated per input channel in the depthwise step.

Usage

```
layer_depthwise_conv_2d(
  object,
  kernel_size,
  strides = list(1L, 1L),
  padding = "valid",
  depth_multiplier = 1L,
  data_format = NULL,
  dilation_rate = list(1L, 1L),
  activation = NULL,
  use_bias = TRUE,
  depthwise_initializer = "glorot_uniform",
  bias_initializer = "zeros",
  depthwise_regularizer = NULL,
  bias_regularizer = NULL,
  activity_regularizer = NULL,
  depthwise_constraint = NULL,
  bias_constraint = NULL,
  ...
)
```

Arguments

`object`	Object to compose the layer with. A tensor, array, or sequential model.
`kernel_size`	int or list of 2 integer, specifying the size of the depthwise convolution window.
`strides`	int or list of 2 integer, specifying the stride length of the depthwise convolution. `strides` > 1 is incompatible with `dilation_rate` > 1.
`padding`	string, either `"valid"` or `"same"` (case-insensitive). `"valid"` means no padding. `"same"` results in padding evenly to the left/right or up/down of the input. When `padding="same"` and `strides=1`, the output has the same size as the input.
`depth_multiplier`	The number of depthwise convolution output channels for each input channel. The total number of depthwise convolution output channels will be equal to `input_channel * depth_multiplier`.
`data_format`	string, either `"channels_last"` or `"channels_first"`. The ordering of the dimensions in the inputs. `"channels_last"` corresponds to inputs with shape `(batch, height, width, channels)` while `"channels_first"` corresponds to inputs with shape `(batch, channels, height, width)`. It defaults to the `image_data_format` value found in your Keras config file at `~/.keras/keras.json`. If you never set it, then it will be `"channels_last"`.
`dilation_rate`	int or list of 2 integers, specifying the dilation rate to use for dilated convolution.
`activation`	Activation function. If `NULL`, no activation is applied.
`use_bias`	bool, if `TRUE`, bias will be added to the output.
`depthwise_initializer`	Initializer for the convolution kernel. If `NULL`, the default initializer (`"glorot_uniform"`) will be used.

bias_initializer
: Initializer for the bias vector. If NULL, the default initializer ("zeros") will be used.

depthwise_regularizer
: Optional regularizer for the convolution kernel.

bias_regularizer
: Optional regularizer for the bias vector.

activity_regularizer
: Optional regularizer function for the output.

depthwise_constraint
: Optional projection function to be applied to the kernel after being updated by an Optimizer (e.g. used to implement norm constraints or value constraints for layer weights). The function must take as input the unprojected variable and must return the projected variable (which must have the same shape). Constraints are not safe to use when doing asynchronous distributed training.

bias_constraint
: Optional projection function to be applied to the bias after being updated by an Optimizer.

...
: For forward/backward compatability.

Value

A 4D tensor representing `activation(depthwise_conv2d(inputs, kernel) + bias)`.

Input Shape

- If `data_format="channels_last"`: A 4D tensor with shape: `(batch_size, height, width, channels)`
- If `data_format="channels_first"`: A 4D tensor with shape: `(batch_size, channels, height, width)`

Output Shape

- If `data_format="channels_last"`: A 4D tensor with shape: `(batch_size, new_height, new_width, channels *`
- If `data_format="channels_first"`: A 4D tensor with shape: `(batch_size, channels * depth_multiplier, new_`

Raises

ValueError: when both `strides > 1` and `dilation_rate > 1`.

Example

```
x <- random_uniform(c(4, 10, 10, 12))
y <- x |> layer_depthwise_conv_2d(3, 3, activation = 'relu')
shape(y)

## shape(4, 3, 3, 12)
```

See Also

- https://keras.io/api/layers/convolution_layers/depthwise_convolution2d#depthwiseconv2d-class

Other convolutional layers:
layer_conv_1d()
layer_conv_1d_transpose()
layer_conv_2d()
layer_conv_2d_transpose()
layer_conv_3d()
layer_conv_3d_transpose()
layer_depthwise_conv_1d()
layer_separable_conv_1d()
layer_separable_conv_2d()

Other layers:
Layer()
layer_activation()
layer_activation_elu()
layer_activation_leaky_relu()
layer_activation_parametric_relu()
layer_activation_relu()
layer_activation_softmax()
layer_activity_regularization()
layer_add()
layer_additive_attention()
layer_alpha_dropout()
layer_attention()
layer_average()
layer_average_pooling_1d()
layer_average_pooling_2d()
layer_average_pooling_3d()
layer_batch_normalization()
layer_bidirectional()
layer_category_encoding()
layer_center_crop()
layer_concatenate()
layer_conv_1d()
layer_conv_1d_transpose()
layer_conv_2d()
layer_conv_2d_transpose()
layer_conv_3d()
layer_conv_3d_transpose()
layer_conv_lstm_1d()
layer_conv_lstm_2d()
layer_conv_lstm_3d()
layer_cropping_1d()
layer_cropping_2d()
layer_cropping_3d()

layer_dense()
layer_depthwise_conv_1d()
layer_discretization()
layer_dot()
layer_dropout()
layer_einsum_dense()
layer_embedding()
layer_feature_space()
layer_flatten()
layer_flax_module_wrapper()
layer_gaussian_dropout()
layer_gaussian_noise()
layer_global_average_pooling_1d()
layer_global_average_pooling_2d()
layer_global_average_pooling_3d()
layer_global_max_pooling_1d()
layer_global_max_pooling_2d()
layer_global_max_pooling_3d()
layer_group_normalization()
layer_group_query_attention()
layer_gru()
layer_hashed_crossing()
layer_hashing()
layer_identity()
layer_integer_lookup()
layer_jax_model_wrapper()
layer_lambda()
layer_layer_normalization()
layer_lstm()
layer_masking()
layer_max_pooling_1d()
layer_max_pooling_2d()
layer_max_pooling_3d()
layer_maximum()
layer_mel_spectrogram()
layer_minimum()
layer_multi_head_attention()
layer_multiply()
layer_normalization()
layer_permute()
layer_random_brightness()
layer_random_contrast()
layer_random_crop()
layer_random_flip()
layer_random_rotation()
layer_random_translation()
layer_random_zoom()
layer_repeat_vector()

layer_rescaling()
layer_reshape()
layer_resizing()
layer_rnn()
layer_separable_conv_1d()
layer_separable_conv_2d()
layer_simple_rnn()
layer_spatial_dropout_1d()
layer_spatial_dropout_2d()
layer_spatial_dropout_3d()
layer_spectral_normalization()
layer_string_lookup()
layer_subtract()
layer_text_vectorization()
layer_tfsm()
layer_time_distributed()
layer_torch_module_wrapper()
layer_unit_normalization()
layer_upsampling_1d()
layer_upsampling_2d()
layer_upsampling_3d()
layer_zero_padding_1d()
layer_zero_padding_2d()
layer_zero_padding_3d()
rnn_cell_gru()
rnn_cell_lstm()
rnn_cell_simple()
rnn_cells_stack()

---

layer_discretization *A preprocessing layer which buckets continuous features by ranges.*

Description

This layer will place each element of its input data into one of several contiguous ranges and output an integer index indicating which range each element was placed in.

Note: This layer is safe to use inside a `tf.data` pipeline (independently of which backend you're using).

Usage

```
layer_discretization(
  object,
  bin_boundaries = NULL,
  num_bins = NULL,
  epsilon = 0.01,
  output_mode = "int",
  sparse = FALSE,
  dtype = NULL,
  name = NULL
)
```

Arguments

`object`	Object to compose the layer with. A tensor, array, or sequential model.
`bin_boundaries`	A list of bin boundaries. The leftmost and rightmost bins will always extend to `-Inf` and `Inf`, so `bin_boundaries = c(0, 1, 2)` generates bins `(-Inf, 0)`, `[0, 1)`, `[1, 2)`, and `[2, +Inf)`. If this option is set, `adapt()` should not be called.
`num_bins`	The integer number of bins to compute. If this option is set, `adapt()` should be called to learn the bin boundaries.
`epsilon`	Error tolerance, typically a small fraction close to zero (e.g. 0.01). Higher values of epsilon increase the quantile approximation, and hence result in more unequal buckets, but could improve performance and resource consumption.
`output_mode`	Specification for the output of the layer. Values can be `"int"`, `"one_hot"`, `"multi_hot"`, or `"count"` configuring the layer as follows: • `"int"`: Return the discretized bin indices directly. • `"one_hot"`: Encodes each individual element in the input into an array the same size as `num_bins`, containing a 1 at the input's bin index. If the last dimension is size 1, will encode on that dimension. If the last dimension is not size 1, will append a new dimension for the encoded output. • `"multi_hot"`: Encodes each sample in the input into a single array the same size as `num_bins`, containing a 1 for each bin index index present in the sample. Treats the last dimension as the sample dimension, if input shape is `(..., sample_length)`, output shape will be `(..., num_tokens)`. • `"count"`: As `"multi_hot"`, but the int array contains a count of the number of times the bin index appeared in the sample. Defaults to `"int"`.
`sparse`	Boolean. Only applicable to `"one_hot"`, `"multi_hot"`, and `"count"` output modes. Only supported with TensorFlow backend. If `TRUE`, returns a `SparseTensor` instead of a dense `Tensor`. Defaults to `FALSE`.
`dtype`	datatype (e.g., `"float32"`).
`name`	String, name for the object

Value

The return value depends on the value provided for the first argument. If `object` is:

- a `keras_model_sequential()`, then the layer is added to the sequential model (which is modified in place). To enable piping, the sequential model is also returned, invisibly.
- a `keras_input()`, then the output tensor from calling `layer(input)` is returned.
- `NULL` or missing, then a `Layer` instance is returned.

Input Shape

Any array of dimension 2 or higher.

Output Shape

Same as input shape.

Examples

Discretize float values based on provided buckets.

```
input <- op_array(rbind(c(-1.5, 1, 3.4, 0.5),
                        c(0, 3, 1.3, 0),
                        c(-.5, 0, .5, 1),
                        c(1.5, 2, 2.5, 3)))
output <- input |> layer_discretization(bin_boundaries = c(0, 1, 2))
output
```

```
## tf.Tensor(
## [[0 2 3 1]
##  [1 3 2 1]
##  [0 1 1 2]
##  [2 3 3 3]], shape=(4, 4), dtype=int64)
```

Discretize float values based on a number of buckets to compute.

```
layer <- layer_discretization(num_bins = 4, epsilon = 0.01)
layer |> adapt(input)
layer(input)
```

```
## tf.Tensor(
## [[0 2 3 1]
##  [1 3 2 1]
##  [0 1 1 2]
##  [2 3 3 3]], shape=(4, 4), dtype=int64)
```

See Also

- https://keras.io/api/layers/preprocessing_layers/numerical/discretization#discretization-class

Other numerical features preprocessing layers:
`layer_normalization()`

Other preprocessing layers:
`layer_category_encoding()`
`layer_center_crop()`

layer_feature_space()
layer_hashed_crossing()
layer_hashing()
layer_integer_lookup()
layer_mel_spectrogram()
layer_normalization()
layer_random_brightness()
layer_random_contrast()
layer_random_crop()
layer_random_flip()
layer_random_rotation()
layer_random_translation()
layer_random_zoom()
layer_rescaling()
layer_resizing()
layer_string_lookup()
layer_text_vectorization()

Other layers:
Layer()
layer_activation()
layer_activation_elu()
layer_activation_leaky_relu()
layer_activation_parametric_relu()
layer_activation_relu()
layer_activation_softmax()
layer_activity_regularization()
layer_add()
layer_additive_attention()
layer_alpha_dropout()
layer_attention()
layer_average()
layer_average_pooling_1d()
layer_average_pooling_2d()
layer_average_pooling_3d()
layer_batch_normalization()
layer_bidirectional()
layer_category_encoding()
layer_center_crop()
layer_concatenate()
layer_conv_1d()
layer_conv_1d_transpose()
layer_conv_2d()
layer_conv_2d_transpose()
layer_conv_3d()
layer_conv_3d_transpose()
layer_conv_lstm_1d()
layer_conv_lstm_2d()

layer_conv_lstm_3d()
layer_cropping_1d()
layer_cropping_2d()
layer_cropping_3d()
layer_dense()
layer_depthwise_conv_1d()
layer_depthwise_conv_2d()
layer_dot()
layer_dropout()
layer_einsum_dense()
layer_embedding()
layer_feature_space()
layer_flatten()
layer_flax_module_wrapper()
layer_gaussian_dropout()
layer_gaussian_noise()
layer_global_average_pooling_1d()
layer_global_average_pooling_2d()
layer_global_average_pooling_3d()
layer_global_max_pooling_1d()
layer_global_max_pooling_2d()
layer_global_max_pooling_3d()
layer_group_normalization()
layer_group_query_attention()
layer_gru()
layer_hashed_crossing()
layer_hashing()
layer_identity()
layer_integer_lookup()
layer_jax_model_wrapper()
layer_lambda()
layer_layer_normalization()
layer_lstm()
layer_masking()
layer_max_pooling_1d()
layer_max_pooling_2d()
layer_max_pooling_3d()
layer_maximum()
layer_mel_spectrogram()
layer_minimum()
layer_multi_head_attention()
layer_multiply()
layer_normalization()
layer_permute()
layer_random_brightness()
layer_random_contrast()
layer_random_crop()
layer_random_flip()

layer_random_rotation()
layer_random_translation()
layer_random_zoom()
layer_repeat_vector()
layer_rescaling()
layer_reshape()
layer_resizing()
layer_rnn()
layer_separable_conv_1d()
layer_separable_conv_2d()
layer_simple_rnn()
layer_spatial_dropout_1d()
layer_spatial_dropout_2d()
layer_spatial_dropout_3d()
layer_spectral_normalization()
layer_string_lookup()
layer_subtract()
layer_text_vectorization()
layer_tfsm()
layer_time_distributed()
layer_torch_module_wrapper()
layer_unit_normalization()
layer_upsampling_1d()
layer_upsampling_2d()
layer_upsampling_3d()
layer_zero_padding_1d()
layer_zero_padding_2d()
layer_zero_padding_3d()
rnn_cell_gru()
rnn_cell_lstm()
rnn_cell_simple()
rnn_cells_stack()

layer_dot *Computes element-wise dot product of two tensors.*

Description

It takes a list of inputs of size 2, and the axes corresponding to each input along with the dot product is to be performed.

Let's say `x` and `y` are the two input tensors with shapes `(2, 3, 5)` and `(2, 10, 3)`. The batch dimension should be of same size for both the inputs, and `axes` should correspond to the dimensions that have the same size in the corresponding inputs. e.g. with `axes = c(1, 2)`, the dot product of `x`, and `y` will result in a tensor with shape `(2, 5, 10)`

Usage

```
layer_dot(inputs, ..., axes, normalize = FALSE)
```

Arguments

`inputs`	layers to combine
`...`	Standard layer keyword arguments.
`axes`	Integer or list of integers, axis or axes along which to take the dot product. If a list, should be two integers corresponding to the desired axis from the first input and the desired axis from the second input, respectively. Note that the size of the two selected axes must match.
`normalize`	Whether to L2-normalize samples along the dot product axis before taking the dot product. If set to `TRUE`, then the output of the dot product is the cosine proximity between the two samples.

Value

A tensor, the dot product of the samples from the inputs.

Examples

```
x <- op_reshape(0:9,   c(1, 5, 2))
y <- op_reshape(10:19, c(1, 2, 5))
layer_dot(x, y, axes=c(2, 3))

## tf.Tensor(
## [[[260 360]
##   [320 445]]], shape=(1, 2, 2), dtype=int32)
```

Usage in a Keras model:

```
x1 <- op_reshape(0:9, c(5, 2)) |> layer_dense(8)
x2 <- op_reshape(10:19, c(5, 2)) |> layer_dense(8)
shape(x1)

## shape(5, 8)


shape(x2)

## shape(5, 8)


y <- layer_dot(x1, x2, axes=2)
shape(y)

## shape(5, 1)
```

See Also

- https://keras.io/api/layers/merging_layers/dot#dot-class

Other merging layers:
layer_add()
layer_average()
layer_concatenate()
layer_maximum()
layer_minimum()
layer_multiply()
layer_subtract()

Other layers:
Layer()
layer_activation()
layer_activation_elu()
layer_activation_leaky_relu()
layer_activation_parametric_relu()
layer_activation_relu()
layer_activation_softmax()
layer_activity_regularization()
layer_add()
layer_additive_attention()
layer_alpha_dropout()
layer_attention()
layer_average()
layer_average_pooling_1d()
layer_average_pooling_2d()
layer_average_pooling_3d()
layer_batch_normalization()
layer_bidirectional()
layer_category_encoding()
layer_center_crop()
layer_concatenate()
layer_conv_1d()
layer_conv_1d_transpose()
layer_conv_2d()
layer_conv_2d_transpose()
layer_conv_3d()
layer_conv_3d_transpose()
layer_conv_lstm_1d()
layer_conv_lstm_2d()
layer_conv_lstm_3d()
layer_cropping_1d()
layer_cropping_2d()
layer_cropping_3d()
layer_dense()
layer_depthwise_conv_1d()

layer_depthwise_conv_2d()
layer_discretization()
layer_dropout()
layer_einsum_dense()
layer_embedding()
layer_feature_space()
layer_flatten()
layer_flax_module_wrapper()
layer_gaussian_dropout()
layer_gaussian_noise()
layer_global_average_pooling_1d()
layer_global_average_pooling_2d()
layer_global_average_pooling_3d()
layer_global_max_pooling_1d()
layer_global_max_pooling_2d()
layer_global_max_pooling_3d()
layer_group_normalization()
layer_group_query_attention()
layer_gru()
layer_hashed_crossing()
layer_hashing()
layer_identity()
layer_integer_lookup()
layer_jax_model_wrapper()
layer_lambda()
layer_layer_normalization()
layer_lstm()
layer_masking()
layer_max_pooling_1d()
layer_max_pooling_2d()
layer_max_pooling_3d()
layer_maximum()
layer_mel_spectrogram()
layer_minimum()
layer_multi_head_attention()
layer_multiply()
layer_normalization()
layer_permute()
layer_random_brightness()
layer_random_contrast()
layer_random_crop()
layer_random_flip()
layer_random_rotation()
layer_random_translation()
layer_random_zoom()
layer_repeat_vector()
layer_rescaling()
layer_reshape()

layer_resizing()
layer_rnn()
layer_separable_conv_1d()
layer_separable_conv_2d()
layer_simple_rnn()
layer_spatial_dropout_1d()
layer_spatial_dropout_2d()
layer_spatial_dropout_3d()
layer_spectral_normalization()
layer_string_lookup()
layer_subtract()
layer_text_vectorization()
layer_tfsm()
layer_time_distributed()
layer_torch_module_wrapper()
layer_unit_normalization()
layer_upsampling_1d()
layer_upsampling_2d()
layer_upsampling_3d()
layer_zero_padding_1d()
layer_zero_padding_2d()
layer_zero_padding_3d()
rnn_cell_gru()
rnn_cell_lstm()
rnn_cell_simple()
rnn_cells_stack()

---

layer_dropout *Applies dropout to the input.*

Description

The Dropout layer randomly sets input units to 0 with a frequency of rate at each step during training time, which helps prevent overfitting. Inputs not set to 0 are scaled up by 1 / (1 - rate) such that the sum over all inputs is unchanged.

Note that the Dropout layer only applies when training is set to TRUE in call(), such that no values are dropped during inference. When using model.fit, training will be appropriately set to TRUE automatically. In other contexts, you can set the argument explicitly to TRUE when calling the layer.

(This is in contrast to setting trainable=FALSE for a Dropout layer. trainable does not affect the layer's behavior, as Dropout does not have any variables/weights that can be frozen during training.)

Usage

```
layer_dropout(object, rate, noise_shape = NULL, seed = NULL, ...)
```

Arguments

`object`	Object to compose the layer with. A tensor, array, or sequential model.
`rate`	Float between 0 and 1. Fraction of the input units to drop.
`noise_shape`	1D integer tensor representing the shape of the binary dropout mask that will be multiplied with the input. For instance, if your inputs have shape `(batch_size, timesteps, features)` and you want the dropout mask to be the same for all timesteps, you can use `noise_shape=(batch_size, 1, features)`.
`seed`	An R integer to use as random seed.
`...`	For forward/backward compatability.

Value

The return value depends on the value provided for the first argument. If `object` is:

- a `keras_model_sequential()`, then the layer is added to the sequential model (which is modified in place). To enable piping, the sequential model is also returned, invisibly.
- a `keras_input()`, then the output tensor from calling `layer(input)` is returned.
- `NULL` or missing, then a `Layer` instance is returned.

Call Arguments

- `inputs`: Input tensor (of any rank).
- `training`: Python boolean indicating whether the layer should behave in training mode (adding dropout) or in inference mode (doing nothing).

See Also

- `https://keras.io/api/layers/regularization_layers/dropout#dropout-class`

Other regularization layers:
`layer_activity_regularization()`
`layer_alpha_dropout()`
`layer_gaussian_dropout()`
`layer_gaussian_noise()`
`layer_spatial_dropout_1d()`
`layer_spatial_dropout_2d()`
`layer_spatial_dropout_3d()`

Other layers:
`Layer()`
`layer_activation()`
`layer_activation_elu()`
`layer_activation_leaky_relu()`
`layer_activation_parametric_relu()`
`layer_activation_relu()`
`layer_activation_softmax()`
`layer_activity_regularization()`

layer_add()
layer_additive_attention()
layer_alpha_dropout()
layer_attention()
layer_average()
layer_average_pooling_1d()
layer_average_pooling_2d()
layer_average_pooling_3d()
layer_batch_normalization()
layer_bidirectional()
layer_category_encoding()
layer_center_crop()
layer_concatenate()
layer_conv_1d()
layer_conv_1d_transpose()
layer_conv_2d()
layer_conv_2d_transpose()
layer_conv_3d()
layer_conv_3d_transpose()
layer_conv_lstm_1d()
layer_conv_lstm_2d()
layer_conv_lstm_3d()
layer_cropping_1d()
layer_cropping_2d()
layer_cropping_3d()
layer_dense()
layer_depthwise_conv_1d()
layer_depthwise_conv_2d()
layer_discretization()
layer_dot()
layer_einsum_dense()
layer_embedding()
layer_feature_space()
layer_flatten()
layer_flax_module_wrapper()
layer_gaussian_dropout()
layer_gaussian_noise()
layer_global_average_pooling_1d()
layer_global_average_pooling_2d()
layer_global_average_pooling_3d()
layer_global_max_pooling_1d()
layer_global_max_pooling_2d()
layer_global_max_pooling_3d()
layer_group_normalization()
layer_group_query_attention()
layer_gru()
layer_hashed_crossing()
layer_hashing()

layer_identity()
layer_integer_lookup()
layer_jax_model_wrapper()
layer_lambda()
layer_layer_normalization()
layer_lstm()
layer_masking()
layer_max_pooling_1d()
layer_max_pooling_2d()
layer_max_pooling_3d()
layer_maximum()
layer_mel_spectrogram()
layer_minimum()
layer_multi_head_attention()
layer_multiply()
layer_normalization()
layer_permute()
layer_random_brightness()
layer_random_contrast()
layer_random_crop()
layer_random_flip()
layer_random_rotation()
layer_random_translation()
layer_random_zoom()
layer_repeat_vector()
layer_rescaling()
layer_reshape()
layer_resizing()
layer_rnn()
layer_separable_conv_1d()
layer_separable_conv_2d()
layer_simple_rnn()
layer_spatial_dropout_1d()
layer_spatial_dropout_2d()
layer_spatial_dropout_3d()
layer_spectral_normalization()
layer_string_lookup()
layer_subtract()
layer_text_vectorization()
layer_tfsm()
layer_time_distributed()
layer_torch_module_wrapper()
layer_unit_normalization()
layer_upsampling_1d()
layer_upsampling_2d()
layer_upsampling_3d()
layer_zero_padding_1d()
layer_zero_padding_2d()

layer_zero_padding_3d()
rnn_cell_gru()
rnn_cell_lstm()
rnn_cell_simple()
rnn_cells_stack()

---

layer_einsum_dense *A layer that uses* einsum *as the backing computation.*

Description

This layer can perform einsum calculations of arbitrary dimensionality.

Usage

```
layer_einsum_dense(
  object,
  equation,
  output_shape,
  activation = NULL,
  bias_axes = NULL,
  kernel_initializer = "glorot_uniform",
  bias_initializer = "zeros",
  kernel_regularizer = NULL,
  bias_regularizer = NULL,
  kernel_constraint = NULL,
  bias_constraint = NULL,
  lora_rank = NULL,
  ...
)
```

Arguments

object	Object to compose the layer with. A tensor, array, or sequential model.
equation	An equation describing the einsum to perform. This equation must be a valid einsum string of the form `ab,bc->ac`, `...ab,bc->...ac`, or `ab...,bc->ac...` where 'ab', 'bc', and 'ac' can be any valid einsum axis expression sequence.
output_shape	The expected shape of the output tensor (excluding the batch dimension and any dimensions represented by ellipses). You can specify `NA` or `NULL` for any dimension that is unknown or can be inferred from the input shape.
activation	Activation function to use. If you don't specify anything, no activation is applied (that is, a "linear" activation: `a(x) = x`).
bias_axes	A string containing the output dimension(s) to apply a bias to. Each character in the `bias_axes` string should correspond to a character in the output portion of the `equation` string.

`kernel_initializer`	Initializer for the `kernel` weights matrix.
`bias_initializer`	Initializer for the bias vector.
`kernel_regularizer`	Regularizer function applied to the `kernel` weights matrix.
`bias_regularizer`	Regularizer function applied to the bias vector.
`kernel_constraint`	Constraint function applied to the `kernel` weights matrix.
`bias_constraint`	Constraint function applied to the bias vector.
`lora_rank`	Optional integer. If set, the layer's forward pass will implement LoRA (Low-Rank Adaptation) with the provided rank. LoRA sets the layer's kernel to non-trainable and replaces it with a delta over the original kernel, obtained via multiplying two lower-rank trainable matrices (the factorization happens on the last dimension). This can be useful to reduce the computation cost of fine-tuning large dense layers. You can also enable LoRA on an existing `EinsumDense` layer by calling `layer$enable_lora(rank)`.
`...`	Base layer keyword arguments, such as `name` and `dtype`.

Value

The return value depends on the value provided for the first argument. If `object` is:

- a `keras_model_sequential()`, then the layer is added to the sequential model (which is modified in place). To enable piping, the sequential model is also returned, invisibly.
- a `keras_input()`, then the output tensor from calling `layer(input)` is returned.
- `NULL` or missing, then a `Layer` instance is returned.

Examples

Biased dense layer with einsums

This example shows how to instantiate a standard Keras dense layer using einsum operations. This example is equivalent to `layer_Dense(64, use_bias=TRUE)`.

```
input <- layer_input(shape = c(32))
output <- input |>
  layer_einsum_dense("ab,bc->ac",
                     output_shape = 64,
                     bias_axes = "c")
output # shape(NA, 64)
```

```
## <KerasTensor shape=(None, 64), dtype=float32, sparse=False, name=keras_tensor_1>
```

Applying a dense layer to a sequence

This example shows how to instantiate a layer that applies the same dense operation to every element in a sequence. Here, the `output_shape` has two values (since there are two non-batch dimensions in the output); the first dimension in the `output_shape` is `NA`, because the sequence dimension `b` has an unknown shape.

```
input <- layer_input(shape = c(32, 128))
output <- input |>
  layer_einsum_dense("abc,cd->abd",
                     output_shape = c(NA, 64),
                     bias_axes = "d")
output  # shape(NA, 32, 64)
```

```
## <KerasTensor shape=(None, None, 64), dtype=float32, sparse=False, name=keras_tensor_3>
```

Applying a dense layer to a sequence using ellipses

This example shows how to instantiate a layer that applies the same dense operation to every element in a sequence, but uses the ellipsis notation instead of specifying the batch and sequence dimensions.

Because we are using ellipsis notation and have specified only one axis, the `output_shape` arg is a single value. When instantiated in this way, the layer can handle any number of sequence dimensions - including the case where no sequence dimension exists.

```
input <- layer_input(shape = c(32, 128))
output <- input |>
  layer_einsum_dense("...x,xy->...y",
                     output_shape = 64,
                     bias_axes = "y")

output  # shape(NA, 32, 64)
```

```
## <KerasTensor shape=(None, 32, 64), dtype=float32, sparse=False, name=keras_tensor_5>
```

Methods

-
  ```
  enable_lora(
    rank,
    a_initializer = 'he_uniform',
    b_initializer = 'zeros'
  )
  ```
- `quantize(mode)`

Readonly properties:

- `kernel`

See Also

Other core layers:
layer_dense()
layer_embedding()
layer_identity()
layer_lambda()
layer_masking()

Other layers:
Layer()
layer_activation()
layer_activation_elu()
layer_activation_leaky_relu()
layer_activation_parametric_relu()
layer_activation_relu()
layer_activation_softmax()
layer_activity_regularization()
layer_add()
layer_additive_attention()
layer_alpha_dropout()
layer_attention()
layer_average()
layer_average_pooling_1d()
layer_average_pooling_2d()
layer_average_pooling_3d()
layer_batch_normalization()
layer_bidirectional()
layer_category_encoding()
layer_center_crop()
layer_concatenate()
layer_conv_1d()
layer_conv_1d_transpose()
layer_conv_2d()
layer_conv_2d_transpose()
layer_conv_3d()
layer_conv_3d_transpose()
layer_conv_lstm_1d()
layer_conv_lstm_2d()
layer_conv_lstm_3d()
layer_cropping_1d()
layer_cropping_2d()
layer_cropping_3d()
layer_dense()
layer_depthwise_conv_1d()
layer_depthwise_conv_2d()
layer_discretization()
layer_dot()
layer_dropout()

layer_embedding()
layer_feature_space()
layer_flatten()
layer_flax_module_wrapper()
layer_gaussian_dropout()
layer_gaussian_noise()
layer_global_average_pooling_1d()
layer_global_average_pooling_2d()
layer_global_average_pooling_3d()
layer_global_max_pooling_1d()
layer_global_max_pooling_2d()
layer_global_max_pooling_3d()
layer_group_normalization()
layer_group_query_attention()
layer_gru()
layer_hashed_crossing()
layer_hashing()
layer_identity()
layer_integer_lookup()
layer_jax_model_wrapper()
layer_lambda()
layer_layer_normalization()
layer_lstm()
layer_masking()
layer_max_pooling_1d()
layer_max_pooling_2d()
layer_max_pooling_3d()
layer_maximum()
layer_mel_spectrogram()
layer_minimum()
layer_multi_head_attention()
layer_multiply()
layer_normalization()
layer_permute()
layer_random_brightness()
layer_random_contrast()
layer_random_crop()
layer_random_flip()
layer_random_rotation()
layer_random_translation()
layer_random_zoom()
layer_repeat_vector()
layer_rescaling()
layer_reshape()
layer_resizing()
layer_rnn()
layer_separable_conv_1d()
layer_separable_conv_2d()

layer_simple_rnn()
layer_spatial_dropout_1d()
layer_spatial_dropout_2d()
layer_spatial_dropout_3d()
layer_spectral_normalization()
layer_string_lookup()
layer_subtract()
layer_text_vectorization()
layer_tfsm()
layer_time_distributed()
layer_torch_module_wrapper()
layer_unit_normalization()
layer_upsampling_1d()
layer_upsampling_2d()
layer_upsampling_3d()
layer_zero_padding_1d()
layer_zero_padding_2d()
layer_zero_padding_3d()
rnn_cell_gru()
rnn_cell_lstm()
rnn_cell_simple()
rnn_cells_stack()

layer_embedding *Turns positive integers (indexes) into dense vectors of fixed size.*

Description

e.g. `rbind(4L, 20L)` $\rightarrow$ `rbind(c(0.25, 0.1), c(0.6, -0.2))`

This layer can only be used on positive integer inputs of a fixed range.

Usage

```
layer_embedding(
  object,
  input_dim,
  output_dim,
  embeddings_initializer = "uniform",
  embeddings_regularizer = NULL,
  embeddings_constraint = NULL,
  mask_zero = FALSE,
  lora_rank = NULL,
  ...
)
```

Arguments

`object`	Object to compose the layer with. A tensor, array, or sequential model.
`input_dim`	Integer. Size of the vocabulary, i.e. maximum integer index + 1.
`output_dim`	Integer. Dimension of the dense embedding.
`embeddings_initializer`	Initializer for the `embeddings` matrix (see `keras3::initializer_*`).
`embeddings_regularizer`	Regularizer function applied to the `embeddings` matrix (see `keras3::regularizer_*`).
`embeddings_constraint`	Constraint function applied to the `embeddings` matrix (see `keras3::constraint_*`).
`mask_zero`	Boolean, whether or not the input value 0 is a special "padding" value that should be masked out. This is useful when using recurrent layers which may take variable length input. If this is `TRUE`, then all subsequent layers in the model need to support masking or an exception will be raised. If `mask_zero` is set to `TRUE`, as a consequence, index 0 cannot be used in the vocabulary (`input_dim` should equal size of vocabulary + 1).
`lora_rank`	Optional integer. If set, the layer's forward pass will implement LoRA (Low-Rank Adaptation) with the provided rank. LoRA sets the layer's embeddings matrix to non-trainable and replaces it with a delta over the original matrix, obtained via multiplying two lower-rank trainable matrices. This can be useful to reduce the computation cost of fine-tuning large embedding layers. You can also enable LoRA on an existing `Embedding` layer instance by calling `layer$enable_lora(rank)`.
`...`	For forward/backward compatability.

Value

The return value depends on the value provided for the first argument. If `object` is:

- a `keras_model_sequential()`, then the layer is added to the sequential model (which is modified in place). To enable piping, the sequential model is also returned, invisibly.
- a `keras_input()`, then the output tensor from calling `layer(input)` is returned.
- `NULL` or missing, then a `Layer` instance is returned.

Example

```
model <- keras_model_sequential() |>
  layer_embedding(1000, 64)

# The model will take as input an integer matrix of size (batch,input_length),
# and the largest integer (i.e. word index) in the input
# should be no larger than 999 (vocabulary size).
# Now model$output_shape is (NA, 10, 64), where `NA` is the batch
# dimension.

input_array <- random_integer(shape = c(32, 10), minval = 0, maxval = 1000)
model |> compile('rmsprop', 'mse')
```

```
output_array <- model |> predict(input_array, verbose = 0)
dim(output_array)    # (32, 10, 64)

## [1] 32 10 64
```

Input Shape

2D tensor with shape: `(batch_size, input_length)`.

Output Shape

3D tensor with shape: `(batch_size, input_length, output_dim)`.

Methods

-
  ```
  enable_lora(
    rank,
    a_initializer = 'he_uniform',
    b_initializer = 'zeros'
  )
  ```
- `quantize(mode)`
- `quantized_build(input_shape, mode)`
- `quantized_call(inputs)`

Readonly properties:

- `embeddings`

See Also

- https://keras.io/api/layers/core_layers/embedding#embedding-class

Other core layers:
`layer_dense()`
`layer_einsum_dense()`
`layer_identity()`
`layer_lambda()`
`layer_masking()`

Other layers:
`Layer()`
`layer_activation()`
`layer_activation_elu()`
`layer_activation_leaky_relu()`
`layer_activation_parametric_relu()`
`layer_activation_relu()`
`layer_activation_softmax()`
`layer_activity_regularization()`

layer_add()
layer_additive_attention()
layer_alpha_dropout()
layer_attention()
layer_average()
layer_average_pooling_1d()
layer_average_pooling_2d()
layer_average_pooling_3d()
layer_batch_normalization()
layer_bidirectional()
layer_category_encoding()
layer_center_crop()
layer_concatenate()
layer_conv_1d()
layer_conv_1d_transpose()
layer_conv_2d()
layer_conv_2d_transpose()
layer_conv_3d()
layer_conv_3d_transpose()
layer_conv_lstm_1d()
layer_conv_lstm_2d()
layer_conv_lstm_3d()
layer_cropping_1d()
layer_cropping_2d()
layer_cropping_3d()
layer_dense()
layer_depthwise_conv_1d()
layer_depthwise_conv_2d()
layer_discretization()
layer_dot()
layer_dropout()
layer_einsum_dense()
layer_feature_space()
layer_flatten()
layer_flax_module_wrapper()
layer_gaussian_dropout()
layer_gaussian_noise()
layer_global_average_pooling_1d()
layer_global_average_pooling_2d()
layer_global_average_pooling_3d()
layer_global_max_pooling_1d()
layer_global_max_pooling_2d()
layer_global_max_pooling_3d()
layer_group_normalization()
layer_group_query_attention()
layer_gru()
layer_hashed_crossing()
layer_hashing()

layer_identity()
layer_integer_lookup()
layer_jax_model_wrapper()
layer_lambda()
layer_layer_normalization()
layer_lstm()
layer_masking()
layer_max_pooling_1d()
layer_max_pooling_2d()
layer_max_pooling_3d()
layer_maximum()
layer_mel_spectrogram()
layer_minimum()
layer_multi_head_attention()
layer_multiply()
layer_normalization()
layer_permute()
layer_random_brightness()
layer_random_contrast()
layer_random_crop()
layer_random_flip()
layer_random_rotation()
layer_random_translation()
layer_random_zoom()
layer_repeat_vector()
layer_rescaling()
layer_reshape()
layer_resizing()
layer_rnn()
layer_separable_conv_1d()
layer_separable_conv_2d()
layer_simple_rnn()
layer_spatial_dropout_1d()
layer_spatial_dropout_2d()
layer_spatial_dropout_3d()
layer_spectral_normalization()
layer_string_lookup()
layer_subtract()
layer_text_vectorization()
layer_tfsm()
layer_time_distributed()
layer_torch_module_wrapper()
layer_unit_normalization()
layer_upsampling_1d()
layer_upsampling_2d()
layer_upsampling_3d()
layer_zero_padding_1d()
layer_zero_padding_2d()

layer_zero_padding_3d()
rnn_cell_gru()
rnn_cell_lstm()
rnn_cell_simple()
rnn_cells_stack()

---

layer_feature_space *One-stop utility for preprocessing and encoding structured data.*

Description

Available feature types:

Note that all features can be referred to by their string name, e.g. "integer_categorical". When using the string name, the default argument values are used.

```
# Plain float values.
feature_float(name = NULL)

# Float values to be preprocessed via featurewise standardization
# (i.e. via a `layer_normalization()` layer).
feature_float_normalized(name = NULL)

# Float values to be preprocessed via linear rescaling
# (i.e. via a `layer_rescaling` layer).
feature_float_rescaled(scale = 1., offset = 0., name = NULL)

# Float values to be discretized. By default, the discrete
# representation will then be one-hot encoded.
feature_float_discretized(
  num_bins,
  bin_boundaries = NULL,
  output_mode = "one_hot",
  name = NULL
)

# Integer values to be indexed. By default, the discrete
# representation will then be one-hot encoded.
feature_integer_categorical(
  max_tokens = NULL,
  num_oov_indices = 1,
  output_mode = "one_hot",
  name = NULL
)

# String values to be indexed. By default, the discrete
# representation will then be one-hot encoded.
feature_string_categorical(
  max_tokens = NULL,
  num_oov_indices = 1,
  output_mode = "one_hot",
  name = NULL
)

# Integer values to be hashed into a fixed number of bins.
# By default, the discrete representation will then be one-hot encoded.
feature_integer_hashed(num_bins, output_mode = "one_hot", name = NULL)

# String values to be hashed into a fixed number of bins.
# By default, the discrete representation will then be one-hot encoded.
feature_string_hashed(num_bins, output_mode = "one_hot", name = NULL)
```

Usage

```
layer_feature_space(
  object,
  features,
  output_mode = "concat",
  crosses = NULL,
  crossing_dim = 32L,
  hashing_dim = 32L,
  num_discretization_bins = 32L,
  name = NULL,
  feature_names = NULL
)

feature_cross(feature_names, crossing_dim, output_mode = "one_hot")

feature_custom(dtype, preprocessor, output_mode)

feature_float(name = NULL)

feature_float_rescaled(scale = 1, offset = 0, name = NULL)

feature_float_normalized(name = NULL)

feature_float_discretized(
  num_bins,
  bin_boundaries = NULL,
  output_mode = "one_hot",
  name = NULL
)

feature_integer_categorical(
  max_tokens = NULL,
  num_oov_indices = 1,
  output_mode = "one_hot",
  name = NULL
)

feature_string_categorical(
  max_tokens = NULL,
  num_oov_indices = 1,
  output_mode = "one_hot",
  name = NULL
)

feature_string_hashed(num_bins, output_mode = "one_hot", name = NULL)

feature_integer_hashed(num_bins, output_mode = "one_hot", name = NULL)
```

Arguments

`object`	see description
`features`	see description
`output_mode`	A string. • For `layer_feature_space()`, one of `"concat"` or `"dict"`. In concat mode, all features get concatenated together into a single vector. In dict mode, the `FeatureSpace` returns a named list of individually encoded features (with the same names as the input list names). • For the `feature_*` functions, one of: `"int"` `"one_hot"` or `"float"`.
`crosses`	List of features to be crossed together, e.g. `crosses=list(c("feature_1", "feature_2"))`. The features will be "crossed" by hashing their combined value into a fixed-length vector.
`crossing_dim`	Default vector size for hashing crossed features. Defaults to `32`.
`hashing_dim`	Default vector size for hashing features of type `"integer_hashed"` and `"string_hashed"`. Defaults to `32`.
`num_discretization_bins`	Default number of bins to be used for discretizing features of type `"float_discretized"`. Defaults to `32`.
`name`	String, name for the object
`feature_names`	Named list mapping the names of your features to their type specification, e.g. `list(my_feature = "integer_categorical")` or `list(my_feature = feature_integer_categorica` For a complete list of all supported types, see "Available feature types" paragraph below.
`dtype`	string, the output dtype of the feature. E.g., "float32".
`preprocessor`	A callable.
`scale, offset`	Passed on to `layer_rescaling()`
`num_bins, bin_boundaries`	Passed on to `layer_discretization()`

`max_tokens, num_oov_indices`
: Passed on to `layer_integer_lookup()` by `feature_integer_categorical()` or to `layer_string_lookup()` by `feature_string_categorical()`.

Value

The return value depends on the value provided for the first argument. If `object` is:

- a `keras_model_sequential()`, then the layer is added to the sequential model (which is modified in place). To enable piping, the sequential model is also returned, invisibly.
- a `keras_input()`, then the output tensor from calling `layer(input)` is returned.
- `NULL` or missing, then a `Layer` instance is returned.

Examples

Basic usage with a named list of input data:

```
raw_data <- list(
  float_values = c(0.0, 0.1, 0.2, 0.3),
  string_values = c("zero", "one", "two", "three"),
  int_values = as.integer(c(0, 1, 2, 3))
)

dataset <- tfdatasets::tensor_slices_dataset(raw_data)

feature_space <- layer_feature_space(
  features = list(
    float_values = "float_normalized",
    string_values = "string_categorical",
    int_values = "integer_categorical"
  ),
  crosses = list(c("string_values", "int_values")),
  output_mode = "concat"
)

# Before you start using the feature_space(),
# you must `adapt()` it on some data.
feature_space |> adapt(dataset)

# You can call the feature_space() on a named list of
# data (batched or unbatched).
output_vector <- feature_space(raw_data)
```

Basic usage with `tf.data`:

```
library(tfdatasets)
# Unlabeled data
preprocessed_ds <- unlabeled_dataset |>
  dataset_map(feature_space)
```

```
# Labeled data
preprocessed_ds <- labeled_dataset |>
  dataset_map(function(x, y) tuple(feature_space(x), y))
```

Basic usage with the Keras Functional API:

```
# Retrieve a named list of Keras layer_input() objects
(inputs <- feature_space$get_inputs())
```

```
## $float_values
## <KerasTensor shape=(None, 1), dtype=float32, sparse=None, name=float_values>
##
## $string_values
## <KerasTensor shape=(None, 1), dtype=string, sparse=None, name=string_values>
##
## $int_values
## <KerasTensor shape=(None, 1), dtype=int32, sparse=None, name=int_values>
```

```
# Retrieve the corresponding encoded Keras tensors
(encoded_features <- feature_space$get_encoded_features())
```

```
## <KerasTensor shape=(None, 43), dtype=float32, sparse=False, name=keras_tensor_7>
```

```
# Build a Functional model
outputs <- encoded_features |> layer_dense(1, activation = "sigmoid")
model <- keras_model(inputs, outputs)
```

Customizing each feature or feature cross:

```
feature_space <- layer_feature_space(
  features = list(
    float_values = feature_float_normalized(),
    string_values = feature_string_categorical(max_tokens = 10),
    int_values = feature_integer_categorical(max_tokens = 10)
  ),
  crosses = list(
    feature_cross(c("string_values", "int_values"), crossing_dim = 32)
  ),
  output_mode = "concat"
)
```

Returning a dict (a named list) of integer-encoded features:

```
feature_space <- layer_feature_space(
  features = list(
    "string_values" = feature_string_categorical(output_mode = "int"),
    "int_values" = feature_integer_categorical(output_mode = "int")
  ),
  crosses = list(
    feature_cross(
      feature_names = c("string_values", "int_values"),
      crossing_dim = 32,
      output_mode = "int"
    )
  ),
  output_mode = "dict"
)
```

Specifying your own Keras preprocessing layer:

```
# Let's say that one of the features is a short text paragraph that
# we want to encode as a vector (one vector per paragraph) via TF-IDF.
data <- list(text = c("1st string", "2nd string", "3rd string"))

# There's a Keras layer for this: layer_text_vectorization()
custom_layer <- layer_text_vectorization(output_mode = "tf_idf")

# We can use feature_custom() to create a custom feature
# that will use our preprocessing layer.
feature_space <- layer_feature_space(
  features = list(
    text = feature_custom(preprocessor = custom_layer,
                          dtype = "string",
                          output_mode = "float"
    )
  ),
  output_mode = "concat"
)
feature_space |> adapt(tfdatasets::tensor_slices_dataset(data))
output_vector <- feature_space(data)
```

Retrieving the underlying Keras preprocessing layers:

```
# The preprocessing layer of each feature is available in `$preprocessors`.
preprocessing_layer <- feature_space$preprocessors$feature1

# The crossing layer of each feature cross is available in `$crossers`.
# It's an instance of layer_hashed_crossing()
crossing_layer <- feature_space$crossers[["feature1_X_feature2"]]
```

Saving and reloading a FeatureSpace:

```
feature_space$save("featurespace.keras")
reloaded_feature_space <- keras$models$load_model("featurespace.keras")
```

See Also

- https://keras.io/api/utils/feature_space#featurespace-class

Other preprocessing layers:
layer_category_encoding()
layer_center_crop()
layer_discretization()
layer_hashed_crossing()
layer_hashing()
layer_integer_lookup()
layer_mel_spectrogram()
layer_normalization()
layer_random_brightness()
layer_random_contrast()
layer_random_crop()
layer_random_flip()
layer_random_rotation()
layer_random_translation()
layer_random_zoom()
layer_rescaling()
layer_resizing()
layer_string_lookup()
layer_text_vectorization()

Other layers:
Layer()
layer_activation()
layer_activation_elu()
layer_activation_leaky_relu()
layer_activation_parametric_relu()
layer_activation_relu()
layer_activation_softmax()
layer_activity_regularization()
layer_add()
layer_additive_attention()
layer_alpha_dropout()
layer_attention()
layer_average()
layer_average_pooling_1d()
layer_average_pooling_2d()
layer_average_pooling_3d()
layer_batch_normalization()
layer_bidirectional()
layer_category_encoding()
layer_center_crop()

layer_concatenate()
layer_conv_1d()
layer_conv_1d_transpose()
layer_conv_2d()
layer_conv_2d_transpose()
layer_conv_3d()
layer_conv_3d_transpose()
layer_conv_lstm_1d()
layer_conv_lstm_2d()
layer_conv_lstm_3d()
layer_cropping_1d()
layer_cropping_2d()
layer_cropping_3d()
layer_dense()
layer_depthwise_conv_1d()
layer_depthwise_conv_2d()
layer_discretization()
layer_dot()
layer_dropout()
layer_einsum_dense()
layer_embedding()
layer_flatten()
layer_flax_module_wrapper()
layer_gaussian_dropout()
layer_gaussian_noise()
layer_global_average_pooling_1d()
layer_global_average_pooling_2d()
layer_global_average_pooling_3d()
layer_global_max_pooling_1d()
layer_global_max_pooling_2d()
layer_global_max_pooling_3d()
layer_group_normalization()
layer_group_query_attention()
layer_gru()
layer_hashed_crossing()
layer_hashing()
layer_identity()
layer_integer_lookup()
layer_jax_model_wrapper()
layer_lambda()
layer_layer_normalization()
layer_lstm()
layer_masking()
layer_max_pooling_1d()
layer_max_pooling_2d()
layer_max_pooling_3d()
layer_maximum()
layer_mel_spectrogram()

layer_minimum()
layer_multi_head_attention()
layer_multiply()
layer_normalization()
layer_permute()
layer_random_brightness()
layer_random_contrast()
layer_random_crop()
layer_random_flip()
layer_random_rotation()
layer_random_translation()
layer_random_zoom()
layer_repeat_vector()
layer_rescaling()
layer_reshape()
layer_resizing()
layer_rnn()
layer_separable_conv_1d()
layer_separable_conv_2d()
layer_simple_rnn()
layer_spatial_dropout_1d()
layer_spatial_dropout_2d()
layer_spatial_dropout_3d()
layer_spectral_normalization()
layer_string_lookup()
layer_subtract()
layer_text_vectorization()
layer_tfsm()
layer_time_distributed()
layer_torch_module_wrapper()
layer_unit_normalization()
layer_upsampling_1d()
layer_upsampling_2d()
layer_upsampling_3d()
layer_zero_padding_1d()
layer_zero_padding_2d()
layer_zero_padding_3d()
rnn_cell_gru()
rnn_cell_lstm()
rnn_cell_simple()
rnn_cells_stack()

Other utils:
audio_dataset_from_directory()
clear_session()
config_disable_interactive_logging()
config_disable_traceback_filtering()
config_enable_interactive_logging()

config_enable_traceback_filtering()
config_is_interactive_logging_enabled()
config_is_traceback_filtering_enabled()
get_file()
get_source_inputs()
image_array_save()
image_dataset_from_directory()
image_from_array()
image_load()
image_smart_resize()
image_to_array()
normalize()
pack_x_y_sample_weight()
pad_sequences()
set_random_seed()
split_dataset()
text_dataset_from_directory()
timeseries_dataset_from_array()
to_categorical()
unpack_x_y_sample_weight()
zip_lists()

---

layer_flatten *Flattens the input. Does not affect the batch size.*

Description

Flattens the input. Does not affect the batch size.

Usage

```
layer_flatten(object, data_format = NULL, ...)
```

Arguments

object	Object to compose the layer with. A tensor, array, or sequential model.
data_format	A string, one of "channels_last" (default) or "channels_first". The ordering of the dimensions in the inputs. "channels_last" corresponds to inputs with shape (batch, ..., channels) while "channels_first" corresponds to inputs with shape (batch, channels, ...). When unspecified, uses image_data_format value found in your Keras config file at ~/.keras/keras.json (if exists). Defaults to "channels_last".
...	For forward/backward compatability.

Value

The return value depends on the value provided for the first argument. If `object` is:

- a `keras_model_sequential()`, then the layer is added to the sequential model (which is modified in place). To enable piping, the sequential model is also returned, invisibly.
- a `keras_input()`, then the output tensor from calling `layer(input)` is returned.
- `NULL` or missing, then a `Layer` instance is returned.

Note

If inputs are shaped `(batch)` without a feature axis, then flattening adds an extra channel dimension and output shape is `(batch, 1)`.

Example

```
x <- layer_input(shape=c(10, 64))
y <- x |> layer_flatten()
shape(y)

## shape(NA, 640)
```

See Also

- https://keras.io/api/layers/reshaping_layers/flatten#flatten-class

Other reshaping layers:
`layer_cropping_1d()`
`layer_cropping_2d()`
`layer_cropping_3d()`
`layer_permute()`
`layer_repeat_vector()`
`layer_reshape()`
`layer_upsampling_1d()`
`layer_upsampling_2d()`
`layer_upsampling_3d()`
`layer_zero_padding_1d()`
`layer_zero_padding_2d()`
`layer_zero_padding_3d()`

Other layers:
`Layer()`
`layer_activation()`
`layer_activation_elu()`
`layer_activation_leaky_relu()`
`layer_activation_parametric_relu()`
`layer_activation_relu()`
`layer_activation_softmax()`
`layer_activity_regularization()`

layer_add()
layer_additive_attention()
layer_alpha_dropout()
layer_attention()
layer_average()
layer_average_pooling_1d()
layer_average_pooling_2d()
layer_average_pooling_3d()
layer_batch_normalization()
layer_bidirectional()
layer_category_encoding()
layer_center_crop()
layer_concatenate()
layer_conv_1d()
layer_conv_1d_transpose()
layer_conv_2d()
layer_conv_2d_transpose()
layer_conv_3d()
layer_conv_3d_transpose()
layer_conv_lstm_1d()
layer_conv_lstm_2d()
layer_conv_lstm_3d()
layer_cropping_1d()
layer_cropping_2d()
layer_cropping_3d()
layer_dense()
layer_depthwise_conv_1d()
layer_depthwise_conv_2d()
layer_discretization()
layer_dot()
layer_dropout()
layer_einsum_dense()
layer_embedding()
layer_feature_space()
layer_flax_module_wrapper()
layer_gaussian_dropout()
layer_gaussian_noise()
layer_global_average_pooling_1d()
layer_global_average_pooling_2d()
layer_global_average_pooling_3d()
layer_global_max_pooling_1d()
layer_global_max_pooling_2d()
layer_global_max_pooling_3d()
layer_group_normalization()
layer_group_query_attention()
layer_gru()
layer_hashed_crossing()
layer_hashing()

layer_identity()
layer_integer_lookup()
layer_jax_model_wrapper()
layer_lambda()
layer_layer_normalization()
layer_lstm()
layer_masking()
layer_max_pooling_1d()
layer_max_pooling_2d()
layer_max_pooling_3d()
layer_maximum()
layer_mel_spectrogram()
layer_minimum()
layer_multi_head_attention()
layer_multiply()
layer_normalization()
layer_permute()
layer_random_brightness()
layer_random_contrast()
layer_random_crop()
layer_random_flip()
layer_random_rotation()
layer_random_translation()
layer_random_zoom()
layer_repeat_vector()
layer_rescaling()
layer_reshape()
layer_resizing()
layer_rnn()
layer_separable_conv_1d()
layer_separable_conv_2d()
layer_simple_rnn()
layer_spatial_dropout_1d()
layer_spatial_dropout_2d()
layer_spatial_dropout_3d()
layer_spectral_normalization()
layer_string_lookup()
layer_subtract()
layer_text_vectorization()
layer_tfsm()
layer_time_distributed()
layer_torch_module_wrapper()
layer_unit_normalization()
layer_upsampling_1d()
layer_upsampling_2d()
layer_upsampling_3d()
layer_zero_padding_1d()
layer_zero_padding_2d()

layer_zero_padding_3d()
rnn_cell_gru()
rnn_cell_lstm()
rnn_cell_simple()
rnn_cells_stack()

---

layer_flax_module_wrapper
Keras Layer that wraps a R*hrefhttps://flax.readthedocs.ioFlax module.*

Description

This layer enables the use of Flax components in the form of `flax.linen.Module` instances within Keras when using JAX as the backend for Keras.

The module method to use for the forward pass can be specified via the `method` argument and is `__call__` by default. This method must take the following arguments with these exact names:

- `self` if the method is bound to the module, which is the case for the default of `__call__`, and `module` otherwise to pass the module.
- `inputs`: the inputs to the model, a JAX array or a `PyTree` of arrays.
- `training` *(optional)*: an argument specifying if we're in training mode or inference mode, `TRUE` is passed in training mode.

`FlaxLayer` handles the non-trainable state of your model and required RNGs automatically. Note that the `mutable` parameter of `flax.linen.Module.apply()` is set to `DenyList(["params"])`, therefore making the assumption that all the variables outside of the "params" collection are non-trainable weights.

This example shows how to create a `FlaxLayer` from a Flax `Module` with the default `__call__` method and no training argument:

```
# keras3::use_backend("jax")
# py_install("flax", "r-keras")

if(config_backend() == "jax" &&
   reticulate::py_module_available("flax")) {

flax <- import("flax")

MyFlaxModule(flax$linen$Module) %py_class% {
  `__call__` <- flax$linen$compact(\(self, inputs) {
    inputs |>
      (flax$linen$Conv(features = 32L, kernel_size = tuple(3L, 3L)))() |>
      flax$linen$relu() |>
      flax$linen$avg_pool(window_shape = tuple(2L, 2L),
                          strides = tuple(2L, 2L)) |>
      # flatten all except batch_size axis
      (\(x) x$reshape(tuple(x$shape[[1]], -1L)))() |>
      (flax$linen$Dense(features = 200L))() |>
      flax$linen$relu() |>
      (flax$linen$Dense(features = 10L))() |>
      flax$linen$softmax()
  })
}

# typical usage:
input <- keras_input(c(28, 28, 3))
output <- input |>
  layer_flax_module_wrapper(MyFlaxModule())

model <- keras_model(input, output)

# to instantiate the layer before composing:
flax_module <- MyFlaxModule()
keras_layer <- layer_flax_module_wrapper(module = flax_module)

input <- keras_input(c(28, 28, 3))
output <- input |>
  keras_layer()

model <- keras_model(input, output)

}
```

This example shows how to wrap the module method to conform to the required signature. This allows having multiple input arguments and a training argument that has a different name and values. This additionally shows how to use a function that is not bound to the module.

```
flax <- import("flax")

MyFlaxModule(flax$linen$Module) \%py_class\% {
  forward <-
    flax$linen$compact(\(self, inputs1, input2, deterministic) {
      # do work ....
      outputs # return
    })
}

my_flax_module_wrapper <- function(module, inputs, training) {
  c(input1, input2) \%<-\% inputs
  module$forward(input1, input2,!training)
}

flax_module <- MyFlaxModule()
```

```
keras_layer <- layer_flax_module_wrapper(module = flax_module,
                                         method = my_flax_module_wrapper)
```

Usage

```
layer_flax_module_wrapper(object, module, method = NULL, variables = NULL, ...)
```

Arguments

`object`	Object to compose the layer with. A tensor, array, or sequential model.
`module`	An instance of `flax.linen.Module` or subclass.
`method`	The method to call the model. This is generally a method in the `Module`. If not provided, the `__call__` method is used. `method` can also be a function not defined in the `Module`, in which case it must take the `Module` as the first argument. It is used for both `Module.init` and `Module.apply`. Details are documented in the `method` argument of `flax.linen.Module.apply()`.
`variables`	A `dict` (named R list) containing all the variables of the module in the same format as what is returned by `flax.linen.Module.init()`. It should contain a `"params"` key and, if applicable, other keys for collections of variables for non-trainable state. This allows passing trained parameters and learned non-trainable state or controlling the initialization. If `NULL` is passed, the module's `init` function is called at build time to initialize the variables of the model.
`...`	For forward/backward compatability.

Value

The return value depends on the value provided for the first argument. If `object` is:

- a `keras_model_sequential()`, then the layer is added to the sequential model (which is modified in place). To enable piping, the sequential model is also returned, invisibly.
- a `keras_input()`, then the output tensor from calling `layer(input)` is returned.
- `NULL` or missing, then a `Layer` instance is returned.

See Also

- `https://www.tensorflow.org/api_docs/python/tf/keras/layers/FlaxLayer`

Other wrapping layers:
`layer_jax_model_wrapper()`
`layer_torch_module_wrapper()`

Other layers:
`Layer()`
`layer_activation()`
`layer_activation_elu()`
`layer_activation_leaky_relu()`
`layer_activation_parametric_relu()`
`layer_activation_relu()`

layer_activation_softmax()
layer_activity_regularization()
layer_add()
layer_additive_attention()
layer_alpha_dropout()
layer_attention()
layer_average()
layer_average_pooling_1d()
layer_average_pooling_2d()
layer_average_pooling_3d()
layer_batch_normalization()
layer_bidirectional()
layer_category_encoding()
layer_center_crop()
layer_concatenate()
layer_conv_1d()
layer_conv_1d_transpose()
layer_conv_2d()
layer_conv_2d_transpose()
layer_conv_3d()
layer_conv_3d_transpose()
layer_conv_lstm_1d()
layer_conv_lstm_2d()
layer_conv_lstm_3d()
layer_cropping_1d()
layer_cropping_2d()
layer_cropping_3d()
layer_dense()
layer_depthwise_conv_1d()
layer_depthwise_conv_2d()
layer_discretization()
layer_dot()
layer_dropout()
layer_einsum_dense()
layer_embedding()
layer_feature_space()
layer_flatten()
layer_gaussian_dropout()
layer_gaussian_noise()
layer_global_average_pooling_1d()
layer_global_average_pooling_2d()
layer_global_average_pooling_3d()
layer_global_max_pooling_1d()
layer_global_max_pooling_2d()
layer_global_max_pooling_3d()
layer_group_normalization()
layer_group_query_attention()
layer_gru()

layer_hashed_crossing()
layer_hashing()
layer_identity()
layer_integer_lookup()
layer_jax_model_wrapper()
layer_lambda()
layer_layer_normalization()
layer_lstm()
layer_masking()
layer_max_pooling_1d()
layer_max_pooling_2d()
layer_max_pooling_3d()
layer_maximum()
layer_mel_spectrogram()
layer_minimum()
layer_multi_head_attention()
layer_multiply()
layer_normalization()
layer_permute()
layer_random_brightness()
layer_random_contrast()
layer_random_crop()
layer_random_flip()
layer_random_rotation()
layer_random_translation()
layer_random_zoom()
layer_repeat_vector()
layer_rescaling()
layer_reshape()
layer_resizing()
layer_rnn()
layer_separable_conv_1d()
layer_separable_conv_2d()
layer_simple_rnn()
layer_spatial_dropout_1d()
layer_spatial_dropout_2d()
layer_spatial_dropout_3d()
layer_spectral_normalization()
layer_string_lookup()
layer_subtract()
layer_text_vectorization()
layer_tfsm()
layer_time_distributed()
layer_torch_module_wrapper()
layer_unit_normalization()
layer_upsampling_1d()
layer_upsampling_2d()
layer_upsampling_3d()

layer_zero_padding_1d()
layer_zero_padding_2d()
layer_zero_padding_3d()
rnn_cell_gru()
rnn_cell_lstm()
rnn_cell_simple()
rnn_cells_stack()

---

layer_gaussian_dropout

Apply multiplicative 1-centered Gaussian noise.

Description

As it is a regularization layer, it is only active at training time.

Usage

```
layer_gaussian_dropout(object, rate, seed = NULL, ...)
```

Arguments

object	Object to compose the layer with. A tensor, array, or sequential model.
rate	Float, drop probability (as with `Dropout`). The multiplicative noise will have standard deviation `sqrt(rate / (1 - rate))`.
seed	Integer, optional random seed to enable deterministic behavior.
...	For forward/backward compatability.

Value

The return value depends on the value provided for the first argument. If `object` is:

- a `keras_model_sequential()`, then the layer is added to the sequential model (which is modified in place). To enable piping, the sequential model is also returned, invisibly.
- a `keras_input()`, then the output tensor from calling `layer(input)` is returned.
- `NULL` or missing, then a `Layer` instance is returned.

Call Arguments

- `inputs`: Input tensor (of any rank).
- `training`: Python boolean indicating whether the layer should behave in training mode (adding dropout) or in inference mode (doing nothing).

See Also

- https://keras.io/api/layers/regularization_layers/gaussian_dropout#gaussiandropout-class

Other regularization layers:
layer_activity_regularization()
layer_alpha_dropout()
layer_dropout()
layer_gaussian_noise()
layer_spatial_dropout_1d()
layer_spatial_dropout_2d()
layer_spatial_dropout_3d()

Other layers:
Layer()
layer_activation()
layer_activation_elu()
layer_activation_leaky_relu()
layer_activation_parametric_relu()
layer_activation_relu()
layer_activation_softmax()
layer_activity_regularization()
layer_add()
layer_additive_attention()
layer_alpha_dropout()
layer_attention()
layer_average()
layer_average_pooling_1d()
layer_average_pooling_2d()
layer_average_pooling_3d()
layer_batch_normalization()
layer_bidirectional()
layer_category_encoding()
layer_center_crop()
layer_concatenate()
layer_conv_1d()
layer_conv_1d_transpose()
layer_conv_2d()
layer_conv_2d_transpose()
layer_conv_3d()
layer_conv_3d_transpose()
layer_conv_lstm_1d()
layer_conv_lstm_2d()
layer_conv_lstm_3d()
layer_cropping_1d()
layer_cropping_2d()
layer_cropping_3d()
layer_dense()
layer_depthwise_conv_1d()

layer_depthwise_conv_2d()
layer_discretization()
layer_dot()
layer_dropout()
layer_einsum_dense()
layer_embedding()
layer_feature_space()
layer_flatten()
layer_flax_module_wrapper()
layer_gaussian_noise()
layer_global_average_pooling_1d()
layer_global_average_pooling_2d()
layer_global_average_pooling_3d()
layer_global_max_pooling_1d()
layer_global_max_pooling_2d()
layer_global_max_pooling_3d()
layer_group_normalization()
layer_group_query_attention()
layer_gru()
layer_hashed_crossing()
layer_hashing()
layer_identity()
layer_integer_lookup()
layer_jax_model_wrapper()
layer_lambda()
layer_layer_normalization()
layer_lstm()
layer_masking()
layer_max_pooling_1d()
layer_max_pooling_2d()
layer_max_pooling_3d()
layer_maximum()
layer_mel_spectrogram()
layer_minimum()
layer_multi_head_attention()
layer_multiply()
layer_normalization()
layer_permute()
layer_random_brightness()
layer_random_contrast()
layer_random_crop()
layer_random_flip()
layer_random_rotation()
layer_random_translation()
layer_random_zoom()
layer_repeat_vector()
layer_rescaling()
layer_reshape()

layer_resizing()
layer_rnn()
layer_separable_conv_1d()
layer_separable_conv_2d()
layer_simple_rnn()
layer_spatial_dropout_1d()
layer_spatial_dropout_2d()
layer_spatial_dropout_3d()
layer_spectral_normalization()
layer_string_lookup()
layer_subtract()
layer_text_vectorization()
layer_tfsm()
layer_time_distributed()
layer_torch_module_wrapper()
layer_unit_normalization()
layer_upsampling_1d()
layer_upsampling_2d()
layer_upsampling_3d()
layer_zero_padding_1d()
layer_zero_padding_2d()
layer_zero_padding_3d()
rnn_cell_gru()
rnn_cell_lstm()
rnn_cell_simple()
rnn_cells_stack()

layer_gaussian_noise *Apply additive zero-centered Gaussian noise.*

Description

This is useful to mitigate overfitting (you could see it as a form of random data augmentation). Gaussian Noise (GS) is a natural choice as corruption process for real valued inputs.

As it is a regularization layer, it is only active at training time.

Usage

```
layer_gaussian_noise(object, stddev, seed = NULL, ...)
```

Arguments

object	Object to compose the layer with. A tensor, array, or sequential model.
stddev	Float, standard deviation of the noise distribution.
seed	Integer, optional random seed to enable deterministic behavior.
...	For forward/backward compatability.

Value

The return value depends on the value provided for the first argument. If `object` is:

- a `keras_model_sequential()`, then the layer is added to the sequential model (which is modified in place). To enable piping, the sequential model is also returned, invisibly.
- a `keras_input()`, then the output tensor from calling `layer(input)` is returned.
- `NULL` or missing, then a `Layer` instance is returned.

Call Arguments

- `inputs`: Input tensor (of any rank).
- `training`: Python boolean indicating whether the layer should behave in training mode (adding noise) or in inference mode (doing nothing).

See Also

- https://keras.io/api/layers/regularization_layers/gaussian_noise#gaussiannoise-class

Other regularization layers:
`layer_activity_regularization()`
`layer_alpha_dropout()`
`layer_dropout()`
`layer_gaussian_dropout()`
`layer_spatial_dropout_1d()`
`layer_spatial_dropout_2d()`
`layer_spatial_dropout_3d()`

Other layers:
`Layer()`
`layer_activation()`
`layer_activation_elu()`
`layer_activation_leaky_relu()`
`layer_activation_parametric_relu()`
`layer_activation_relu()`
`layer_activation_softmax()`
`layer_activity_regularization()`
`layer_add()`
`layer_additive_attention()`
`layer_alpha_dropout()`
`layer_attention()`
`layer_average()`
`layer_average_pooling_1d()`
`layer_average_pooling_2d()`
`layer_average_pooling_3d()`
`layer_batch_normalization()`
`layer_bidirectional()`
`layer_category_encoding()`
`layer_center_crop()`

layer_concatenate()
layer_conv_1d()
layer_conv_1d_transpose()
layer_conv_2d()
layer_conv_2d_transpose()
layer_conv_3d()
layer_conv_3d_transpose()
layer_conv_lstm_1d()
layer_conv_lstm_2d()
layer_conv_lstm_3d()
layer_cropping_1d()
layer_cropping_2d()
layer_cropping_3d()
layer_dense()
layer_depthwise_conv_1d()
layer_depthwise_conv_2d()
layer_discretization()
layer_dot()
layer_dropout()
layer_einsum_dense()
layer_embedding()
layer_feature_space()
layer_flatten()
layer_flax_module_wrapper()
layer_gaussian_dropout()
layer_global_average_pooling_1d()
layer_global_average_pooling_2d()
layer_global_average_pooling_3d()
layer_global_max_pooling_1d()
layer_global_max_pooling_2d()
layer_global_max_pooling_3d()
layer_group_normalization()
layer_group_query_attention()
layer_gru()
layer_hashed_crossing()
layer_hashing()
layer_identity()
layer_integer_lookup()
layer_jax_model_wrapper()
layer_lambda()
layer_layer_normalization()
layer_lstm()
layer_masking()
layer_max_pooling_1d()
layer_max_pooling_2d()
layer_max_pooling_3d()
layer_maximum()
layer_mel_spectrogram()

layer_minimum()
layer_multi_head_attention()
layer_multiply()
layer_normalization()
layer_permute()
layer_random_brightness()
layer_random_contrast()
layer_random_crop()
layer_random_flip()
layer_random_rotation()
layer_random_translation()
layer_random_zoom()
layer_repeat_vector()
layer_rescaling()
layer_reshape()
layer_resizing()
layer_rnn()
layer_separable_conv_1d()
layer_separable_conv_2d()
layer_simple_rnn()
layer_spatial_dropout_1d()
layer_spatial_dropout_2d()
layer_spatial_dropout_3d()
layer_spectral_normalization()
layer_string_lookup()
layer_subtract()
layer_text_vectorization()
layer_tfsm()
layer_time_distributed()
layer_torch_module_wrapper()
layer_unit_normalization()
layer_upsampling_1d()
layer_upsampling_2d()
layer_upsampling_3d()
layer_zero_padding_1d()
layer_zero_padding_2d()
layer_zero_padding_3d()
rnn_cell_gru()
rnn_cell_lstm()
rnn_cell_simple()
rnn_cells_stack()

layer_global_average_pooling_1d

Global average pooling operation for temporal data.

Description

Global average pooling operation for temporal data.

Usage

```
layer_global_average_pooling_1d(
  object,
  data_format = NULL,
  keepdims = FALSE,
  ...
)
```

Arguments

`object`	Object to compose the layer with. A tensor, array, or sequential model.
`data_format`	string, either `"channels_last"` or `"channels_first"`. The ordering of the dimensions in the inputs. `"channels_last"` corresponds to inputs with shape `(batch, steps, features)` while `"channels_first"` corresponds to inputs with shape `(batch, features, steps)`. It defaults to the `image_data_format` value found in your Keras config file at `~/.keras/keras.json`. If you never set it, then it will be `"channels_last"`.
`keepdims`	A boolean, whether to keep the temporal dimension or not. If `keepdims` is `FALSE` (default), the rank of the tensor is reduced for spatial dimensions. If `keepdims` is `TRUE`, the temporal dimension are retained with length 1. The behavior is the same as for `tf$reduce_mean()` or `op_mean()`.
`...`	For forward/backward compatability.

Value

The return value depends on the value provided for the first argument. If `object` is:

- a `keras_model_sequential()`, then the layer is added to the sequential model (which is modified in place). To enable piping, the sequential model is also returned, invisibly.
- a `keras_input()`, then the output tensor from calling `layer(input)` is returned.
- `NULL` or missing, then a `Layer` instance is returned.

Call Arguments

- `inputs`: A 3D tensor.
- `mask`: Binary tensor of shape `(batch_size, steps)` indicating whether a given step should be masked (excluded from the average).

Input Shape

- If `data_format='channels_last'`: 3D tensor with shape: `(batch_size, steps, features)`
- If `data_format='channels_first'`: 3D tensor with shape: `(batch_size, features, steps)`

Output Shape

- If keepdims=FALSE: 2D tensor with shape (batch_size, features).
- If keepdims=TRUE:
  - If data_format="channels_last": 3D tensor with shape (batch_size, 1, features)
  - If data_format="channels_first": 3D tensor with shape (batch_size, features, 1)

Examples

```
x <- random_uniform(c(2, 3, 4))
y <- x |> layer_global_average_pooling_1d()
shape(y)

## shape(2, 4)
```

See Also

- https://keras.io/api/layers/pooling_layers/global_average_pooling1d#globalaveragepooling1d-clas

Other pooling layers:
layer_average_pooling_1d()
layer_average_pooling_2d()
layer_average_pooling_3d()
layer_global_average_pooling_2d()
layer_global_average_pooling_3d()
layer_global_max_pooling_1d()
layer_global_max_pooling_2d()
layer_global_max_pooling_3d()
layer_max_pooling_1d()
layer_max_pooling_2d()
layer_max_pooling_3d()

Other layers:
Layer()
layer_activation()
layer_activation_elu()
layer_activation_leaky_relu()
layer_activation_parametric_relu()
layer_activation_relu()
layer_activation_softmax()
layer_activity_regularization()
layer_add()
layer_additive_attention()
layer_alpha_dropout()
layer_attention()
layer_average()
layer_average_pooling_1d()
layer_average_pooling_2d()

layer_average_pooling_3d()
layer_batch_normalization()
layer_bidirectional()
layer_category_encoding()
layer_center_crop()
layer_concatenate()
layer_conv_1d()
layer_conv_1d_transpose()
layer_conv_2d()
layer_conv_2d_transpose()
layer_conv_3d()
layer_conv_3d_transpose()
layer_conv_lstm_1d()
layer_conv_lstm_2d()
layer_conv_lstm_3d()
layer_cropping_1d()
layer_cropping_2d()
layer_cropping_3d()
layer_dense()
layer_depthwise_conv_1d()
layer_depthwise_conv_2d()
layer_discretization()
layer_dot()
layer_dropout()
layer_einsum_dense()
layer_embedding()
layer_feature_space()
layer_flatten()
layer_flax_module_wrapper()
layer_gaussian_dropout()
layer_gaussian_noise()
layer_global_average_pooling_2d()
layer_global_average_pooling_3d()
layer_global_max_pooling_1d()
layer_global_max_pooling_2d()
layer_global_max_pooling_3d()
layer_group_normalization()
layer_group_query_attention()
layer_gru()
layer_hashed_crossing()
layer_hashing()
layer_identity()
layer_integer_lookup()
layer_jax_model_wrapper()
layer_lambda()
layer_layer_normalization()
layer_lstm()
layer_masking()

layer_max_pooling_1d()
layer_max_pooling_2d()
layer_max_pooling_3d()
layer_maximum()
layer_mel_spectrogram()
layer_minimum()
layer_multi_head_attention()
layer_multiply()
layer_normalization()
layer_permute()
layer_random_brightness()
layer_random_contrast()
layer_random_crop()
layer_random_flip()
layer_random_rotation()
layer_random_translation()
layer_random_zoom()
layer_repeat_vector()
layer_rescaling()
layer_reshape()
layer_resizing()
layer_rnn()
layer_separable_conv_1d()
layer_separable_conv_2d()
layer_simple_rnn()
layer_spatial_dropout_1d()
layer_spatial_dropout_2d()
layer_spatial_dropout_3d()
layer_spectral_normalization()
layer_string_lookup()
layer_subtract()
layer_text_vectorization()
layer_tfsm()
layer_time_distributed()
layer_torch_module_wrapper()
layer_unit_normalization()
layer_upsampling_1d()
layer_upsampling_2d()
layer_upsampling_3d()
layer_zero_padding_1d()
layer_zero_padding_2d()
layer_zero_padding_3d()
rnn_cell_gru()
rnn_cell_lstm()
rnn_cell_simple()
rnn_cells_stack()

layer_global_average_pooling_2d

Global average pooling operation for 2D data.

Description

Global average pooling operation for 2D data.

Usage

```
layer_global_average_pooling_2d(
  object,
  data_format = NULL,
  keepdims = FALSE,
  ...
)
```

Arguments

`object`	Object to compose the layer with. A tensor, array, or sequential model.
`data_format`	string, either `"channels_last"` or `"channels_first"`. The ordering of the dimensions in the inputs. `"channels_last"` corresponds to inputs with shape `(batch, height, width, channels)` while `"channels_first"` corresponds to inputs with shape `(batch, features, height, weight)`. It defaults to the `image_data_format` value found in your Keras config file at `~/.keras/keras.json`. If you never set it, then it will be `"channels_last"`.
`keepdims`	A boolean, whether to keep the temporal dimension or not. If `keepdims` is `FALSE` (default), the rank of the tensor is reduced for spatial dimensions. If `keepdims` is `TRUE`, the spatial dimension are retained with length 1. The behavior is the same as for `tf$reduce_mean()` or `op_mean()`.
`...`	For forward/backward compatability.

Value

The return value depends on the value provided for the first argument. If `object` is:

- a `keras_model_sequential()`, then the layer is added to the sequential model (which is modified in place). To enable piping, the sequential model is also returned, invisibly.
- a `keras_input()`, then the output tensor from calling `layer(input)` is returned.
- `NULL` or missing, then a `Layer` instance is returned.

Input Shape

- If `data_format='channels_last'`: 4D tensor with shape: `(batch_size, height, width, channels)`
- If `data_format='channels_first'`: 4D tensor with shape: `(batch_size, channels, height, width)`

Output Shape

- If keepdims=FALSE: 2D tensor with shape (batch_size, channels).
- If keepdims=TRUE:
  - If data_format="channels_last": 4D tensor with shape (batch_size, 1, 1, channels)
  - If data_format="channels_first": 4D tensor with shape (batch_size, channels, 1, 1)

Examples

```
x <- random_uniform(c(2, 4, 5, 3))
y <- x |> layer_global_average_pooling_2d()
shape(y)

## shape(2, 3)
```

See Also

- https://keras.io/api/layers/pooling_layers/global_average_pooling2d#globalaveragepooling2d-clas

Other pooling layers:
layer_average_pooling_1d()
layer_average_pooling_2d()
layer_average_pooling_3d()
layer_global_average_pooling_1d()
layer_global_average_pooling_3d()
layer_global_max_pooling_1d()
layer_global_max_pooling_2d()
layer_global_max_pooling_3d()
layer_max_pooling_1d()
layer_max_pooling_2d()
layer_max_pooling_3d()

Other layers:
Layer()
layer_activation()
layer_activation_elu()
layer_activation_leaky_relu()
layer_activation_parametric_relu()
layer_activation_relu()
layer_activation_softmax()
layer_activity_regularization()
layer_add()
layer_additive_attention()
layer_alpha_dropout()
layer_attention()
layer_average()
layer_average_pooling_1d()
layer_average_pooling_2d()

layer_average_pooling_3d()
layer_batch_normalization()
layer_bidirectional()
layer_category_encoding()
layer_center_crop()
layer_concatenate()
layer_conv_1d()
layer_conv_1d_transpose()
layer_conv_2d()
layer_conv_2d_transpose()
layer_conv_3d()
layer_conv_3d_transpose()
layer_conv_lstm_1d()
layer_conv_lstm_2d()
layer_conv_lstm_3d()
layer_cropping_1d()
layer_cropping_2d()
layer_cropping_3d()
layer_dense()
layer_depthwise_conv_1d()
layer_depthwise_conv_2d()
layer_discretization()
layer_dot()
layer_dropout()
layer_einsum_dense()
layer_embedding()
layer_feature_space()
layer_flatten()
layer_flax_module_wrapper()
layer_gaussian_dropout()
layer_gaussian_noise()
layer_global_average_pooling_1d()
layer_global_average_pooling_3d()
layer_global_max_pooling_1d()
layer_global_max_pooling_2d()
layer_global_max_pooling_3d()
layer_group_normalization()
layer_group_query_attention()
layer_gru()
layer_hashed_crossing()
layer_hashing()
layer_identity()
layer_integer_lookup()
layer_jax_model_wrapper()
layer_lambda()
layer_layer_normalization()
layer_lstm()
layer_masking()

layer_max_pooling_1d()
layer_max_pooling_2d()
layer_max_pooling_3d()
layer_maximum()
layer_mel_spectrogram()
layer_minimum()
layer_multi_head_attention()
layer_multiply()
layer_normalization()
layer_permute()
layer_random_brightness()
layer_random_contrast()
layer_random_crop()
layer_random_flip()
layer_random_rotation()
layer_random_translation()
layer_random_zoom()
layer_repeat_vector()
layer_rescaling()
layer_reshape()
layer_resizing()
layer_rnn()
layer_separable_conv_1d()
layer_separable_conv_2d()
layer_simple_rnn()
layer_spatial_dropout_1d()
layer_spatial_dropout_2d()
layer_spatial_dropout_3d()
layer_spectral_normalization()
layer_string_lookup()
layer_subtract()
layer_text_vectorization()
layer_tfsm()
layer_time_distributed()
layer_torch_module_wrapper()
layer_unit_normalization()
layer_upsampling_1d()
layer_upsampling_2d()
layer_upsampling_3d()
layer_zero_padding_1d()
layer_zero_padding_2d()
layer_zero_padding_3d()
rnn_cell_gru()
rnn_cell_lstm()
rnn_cell_simple()
rnn_cells_stack()

layer_global_average_pooling_3d

Global average pooling operation for 3D data.

Description

Global average pooling operation for 3D data.

Usage

```
layer_global_average_pooling_3d(
  object,
  data_format = NULL,
  keepdims = FALSE,
  ...
)
```

Arguments

`object`	Object to compose the layer with. A tensor, array, or sequential model.
`data_format`	string, either `"channels_last"` or `"channels_first"`. The ordering of the dimensions in the inputs. `"channels_last"` corresponds to inputs with shape `(batch, spatial_dim1, spatial_dim2, spatial_dim3, channels)` while `"channels_first"` corresponds to inputs with shape `(batch, channels, spatial_dim1, spatial_di` It defaults to the `image_data_format` value found in your Keras config file at `~/.keras/keras.json`. If you never set it, then it will be `"channels_last"`.
`keepdims`	A boolean, whether to keep the temporal dimension or not. If `keepdims` is `FALSE` (default), the rank of the tensor is reduced for spatial dimensions. If `keepdims` is `TRUE`, the spatial dimension are retained with length 1. The behavior is the same as for `tf$reduce_mean()` or `op_mean()`.
`...`	For forward/backward compatability.

Value

The return value depends on the value provided for the first argument. If `object` is:

- a `keras_model_sequential()`, then the layer is added to the sequential model (which is modified in place). To enable piping, the sequential model is also returned, invisibly.
- a `keras_input()`, then the output tensor from calling `layer(input)` is returned.
- `NULL` or missing, then a `Layer` instance is returned.

Input Shape

- If `data_format='channels_last'`: 5D tensor with shape: `(batch_size, spatial_dim1, spatial_dim2, spatial`
- If `data_format='channels_first'`: 5D tensor with shape: `(batch_size, channels, spatial_dim1, spatial_di`

Output Shape

- If keepdims=FALSE: 2D tensor with shape (batch_size, channels).
- If keepdims=TRUE:
  - If data_format="channels_last": 5D tensor with shape (batch_size, 1, 1, 1, channels)
  - If data_format="channels_first": 5D tensor with shape (batch_size, channels, 1, 1, 1)

Examples

```
x <- random_uniform(c(2, 4, 5, 4, 3))
y <- x |> layer_global_average_pooling_3d()
shape(y)

## shape(2, 3)
```

See Also

- https://keras.io/api/layers/pooling_layers/global_average_pooling3d#globalaveragepooling3d-clas

Other pooling layers:
layer_average_pooling_1d()
layer_average_pooling_2d()
layer_average_pooling_3d()
layer_global_average_pooling_1d()
layer_global_average_pooling_2d()
layer_global_max_pooling_1d()
layer_global_max_pooling_2d()
layer_global_max_pooling_3d()
layer_max_pooling_1d()
layer_max_pooling_2d()
layer_max_pooling_3d()

Other layers:
Layer()
layer_activation()
layer_activation_elu()
layer_activation_leaky_relu()
layer_activation_parametric_relu()
layer_activation_relu()
layer_activation_softmax()
layer_activity_regularization()
layer_add()
layer_additive_attention()
layer_alpha_dropout()
layer_attention()
layer_average()
layer_average_pooling_1d()
layer_average_pooling_2d()

layer_average_pooling_3d()
layer_batch_normalization()
layer_bidirectional()
layer_category_encoding()
layer_center_crop()
layer_concatenate()
layer_conv_1d()
layer_conv_1d_transpose()
layer_conv_2d()
layer_conv_2d_transpose()
layer_conv_3d()
layer_conv_3d_transpose()
layer_conv_lstm_1d()
layer_conv_lstm_2d()
layer_conv_lstm_3d()
layer_cropping_1d()
layer_cropping_2d()
layer_cropping_3d()
layer_dense()
layer_depthwise_conv_1d()
layer_depthwise_conv_2d()
layer_discretization()
layer_dot()
layer_dropout()
layer_einsum_dense()
layer_embedding()
layer_feature_space()
layer_flatten()
layer_flax_module_wrapper()
layer_gaussian_dropout()
layer_gaussian_noise()
layer_global_average_pooling_1d()
layer_global_average_pooling_2d()
layer_global_max_pooling_1d()
layer_global_max_pooling_2d()
layer_global_max_pooling_3d()
layer_group_normalization()
layer_group_query_attention()
layer_gru()
layer_hashed_crossing()
layer_hashing()
layer_identity()
layer_integer_lookup()
layer_jax_model_wrapper()
layer_lambda()
layer_layer_normalization()
layer_lstm()
layer_masking()

layer_max_pooling_1d()
layer_max_pooling_2d()
layer_max_pooling_3d()
layer_maximum()
layer_mel_spectrogram()
layer_minimum()
layer_multi_head_attention()
layer_multiply()
layer_normalization()
layer_permute()
layer_random_brightness()
layer_random_contrast()
layer_random_crop()
layer_random_flip()
layer_random_rotation()
layer_random_translation()
layer_random_zoom()
layer_repeat_vector()
layer_rescaling()
layer_reshape()
layer_resizing()
layer_rnn()
layer_separable_conv_1d()
layer_separable_conv_2d()
layer_simple_rnn()
layer_spatial_dropout_1d()
layer_spatial_dropout_2d()
layer_spatial_dropout_3d()
layer_spectral_normalization()
layer_string_lookup()
layer_subtract()
layer_text_vectorization()
layer_tfsm()
layer_time_distributed()
layer_torch_module_wrapper()
layer_unit_normalization()
layer_upsampling_1d()
layer_upsampling_2d()
layer_upsampling_3d()
layer_zero_padding_1d()
layer_zero_padding_2d()
layer_zero_padding_3d()
rnn_cell_gru()
rnn_cell_lstm()
rnn_cell_simple()
rnn_cells_stack()

layer_global_max_pooling_1d

Global max pooling operation for temporal data.

Description

Global max pooling operation for temporal data.

Usage

```
layer_global_max_pooling_1d(object, data_format = NULL, keepdims = FALSE, ...)
```

Arguments

`object`	Object to compose the layer with. A tensor, array, or sequential model.
`data_format`	string, either `"channels_last"` or `"channels_first"`. The ordering of the dimensions in the inputs. `"channels_last"` corresponds to inputs with shape `(batch, steps, features)` while `"channels_first"` corresponds to inputs with shape `(batch, features, steps)`. It defaults to the `image_data_format` value found in your Keras config file at `~/.keras/keras.json`. If you never set it, then it will be `"channels_last"`.
`keepdims`	A boolean, whether to keep the temporal dimension or not. If `keepdims` is `FALSE` (default), the rank of the tensor is reduced for spatial dimensions. If `keepdims` is `TRUE`, the temporal dimension are retained with length 1. The behavior is the same as for `tf$reduce_mean()` or `op_mean()`.
`...`	For forward/backward compatability.

Value

The return value depends on the value provided for the first argument. If `object` is:

- a `keras_model_sequential()`, then the layer is added to the sequential model (which is modified in place). To enable piping, the sequential model is also returned, invisibly.
- a `keras_input()`, then the output tensor from calling `layer(input)` is returned.
- `NULL` or missing, then a `Layer` instance is returned.

Input Shape

- If `data_format='channels_last'`: 3D tensor with shape: `(batch_size, steps, features)`
- If `data_format='channels_first'`: 3D tensor with shape: `(batch_size, features, steps)`

Output Shape

- If `keepdims=FALSE`: 2D tensor with shape `(batch_size, features)`.
- If `keepdims=TRUE`:
  - If `data_format="channels_last"`: 3D tensor with shape `(batch_size, 1, features)`
  - If `data_format="channels_first"`: 3D tensor with shape `(batch_size, features, 1)`

Examples

```
x <- random_uniform(c(2, 3, 4))
y <- x |> layer_global_max_pooling_1d()
shape(y)

## shape(2, 4)
```

See Also

- https://keras.io/api/layers/pooling_layers/global_max_pooling1d#globalmaxpooling1d-class

Other pooling layers:
layer_average_pooling_1d()
layer_average_pooling_2d()
layer_average_pooling_3d()
layer_global_average_pooling_1d()
layer_global_average_pooling_2d()
layer_global_average_pooling_3d()
layer_global_max_pooling_2d()
layer_global_max_pooling_3d()
layer_max_pooling_1d()
layer_max_pooling_2d()
layer_max_pooling_3d()

Other layers:
Layer()
layer_activation()
layer_activation_elu()
layer_activation_leaky_relu()
layer_activation_parametric_relu()
layer_activation_relu()
layer_activation_softmax()
layer_activity_regularization()
layer_add()
layer_additive_attention()
layer_alpha_dropout()
layer_attention()
layer_average()
layer_average_pooling_1d()
layer_average_pooling_2d()
layer_average_pooling_3d()
layer_batch_normalization()
layer_bidirectional()
layer_category_encoding()
layer_center_crop()
layer_concatenate()
layer_conv_1d()

layer_conv_1d_transpose()
layer_conv_2d()
layer_conv_2d_transpose()
layer_conv_3d()
layer_conv_3d_transpose()
layer_conv_lstm_1d()
layer_conv_lstm_2d()
layer_conv_lstm_3d()
layer_cropping_1d()
layer_cropping_2d()
layer_cropping_3d()
layer_dense()
layer_depthwise_conv_1d()
layer_depthwise_conv_2d()
layer_discretization()
layer_dot()
layer_dropout()
layer_einsum_dense()
layer_embedding()
layer_feature_space()
layer_flatten()
layer_flax_module_wrapper()
layer_gaussian_dropout()
layer_gaussian_noise()
layer_global_average_pooling_1d()
layer_global_average_pooling_2d()
layer_global_average_pooling_3d()
layer_global_max_pooling_2d()
layer_global_max_pooling_3d()
layer_group_normalization()
layer_group_query_attention()
layer_gru()
layer_hashed_crossing()
layer_hashing()
layer_identity()
layer_integer_lookup()
layer_jax_model_wrapper()
layer_lambda()
layer_layer_normalization()
layer_lstm()
layer_masking()
layer_max_pooling_1d()
layer_max_pooling_2d()
layer_max_pooling_3d()
layer_maximum()
layer_mel_spectrogram()
layer_minimum()
layer_multi_head_attention()

layer_multiply()
layer_normalization()
layer_permute()
layer_random_brightness()
layer_random_contrast()
layer_random_crop()
layer_random_flip()
layer_random_rotation()
layer_random_translation()
layer_random_zoom()
layer_repeat_vector()
layer_rescaling()
layer_reshape()
layer_resizing()
layer_rnn()
layer_separable_conv_1d()
layer_separable_conv_2d()
layer_simple_rnn()
layer_spatial_dropout_1d()
layer_spatial_dropout_2d()
layer_spatial_dropout_3d()
layer_spectral_normalization()
layer_string_lookup()
layer_subtract()
layer_text_vectorization()
layer_tfsm()
layer_time_distributed()
layer_torch_module_wrapper()
layer_unit_normalization()
layer_upsampling_1d()
layer_upsampling_2d()
layer_upsampling_3d()
layer_zero_padding_1d()
layer_zero_padding_2d()
layer_zero_padding_3d()
rnn_cell_gru()
rnn_cell_lstm()
rnn_cell_simple()
rnn_cells_stack()

`layer_global_max_pooling_2d`

Global max pooling operation for 2D data.

Description

Global max pooling operation for 2D data.

Usage

```
layer_global_max_pooling_2d(object, data_format = NULL, keepdims = FALSE, ...)
```

Arguments

`object`	Object to compose the layer with. A tensor, array, or sequential model.
`data_format`	string, either `"channels_last"` or `"channels_first"`. The ordering of the dimensions in the inputs. `"channels_last"` corresponds to inputs with shape `(batch, height, width, channels)` while `"channels_first"` corresponds to inputs with shape `(batch, features, height, weight)`. It defaults to the `image_data_format` value found in your Keras config file at `~/.keras/keras.json`. If you never set it, then it will be `"channels_last"`.
`keepdims`	A boolean, whether to keep the temporal dimension or not. If `keepdims` is `FALSE` (default), the rank of the tensor is reduced for spatial dimensions. If `keepdims` is `TRUE`, the spatial dimension are retained with length 1. The behavior is the same as for `tf$reduce_mean()` or `op_mean()`.
`...`	For forward/backward compatability.

Value

The return value depends on the value provided for the first argument. If `object` is:

- a `keras_model_sequential()`, then the layer is added to the sequential model (which is modified in place). To enable piping, the sequential model is also returned, invisibly.
- a `keras_input()`, then the output tensor from calling `layer(input)` is returned.
- `NULL` or missing, then a `Layer` instance is returned.

Input Shape

- If `data_format='channels_last'`: 4D tensor with shape: `(batch_size, height, width, channels)`
- If `data_format='channels_first'`: 4D tensor with shape: `(batch_size, channels, height, width)`

Output Shape

- If `keepdims=FALSE`: 2D tensor with shape `(batch_size, channels)`.
- If `keepdims=TRUE`:
  - If `data_format="channels_last"`: 4D tensor with shape `(batch_size, 1, 1, channels)`
  - If `data_format="channels_first"`: 4D tensor with shape `(batch_size, channels, 1, 1)`

Examples

```
x <- random_uniform(c(2, 4, 5, 3))
y <- x |> layer_global_max_pooling_2d()
shape(y)

## shape(2, 3)
```

See Also

- https://keras.io/api/layers/pooling_layers/global_max_pooling2d#globalmaxpooling2d-class

Other pooling layers:
layer_average_pooling_1d()
layer_average_pooling_2d()
layer_average_pooling_3d()
layer_global_average_pooling_1d()
layer_global_average_pooling_2d()
layer_global_average_pooling_3d()
layer_global_max_pooling_1d()
layer_global_max_pooling_3d()
layer_max_pooling_1d()
layer_max_pooling_2d()
layer_max_pooling_3d()

Other layers:
Layer()
layer_activation()
layer_activation_elu()
layer_activation_leaky_relu()
layer_activation_parametric_relu()
layer_activation_relu()
layer_activation_softmax()
layer_activity_regularization()
layer_add()
layer_additive_attention()
layer_alpha_dropout()
layer_attention()
layer_average()
layer_average_pooling_1d()
layer_average_pooling_2d()
layer_average_pooling_3d()
layer_batch_normalization()
layer_bidirectional()
layer_category_encoding()
layer_center_crop()
layer_concatenate()
layer_conv_1d()
layer_conv_1d_transpose()
layer_conv_2d()
layer_conv_2d_transpose()
layer_conv_3d()
layer_conv_3d_transpose()
layer_conv_lstm_1d()
layer_conv_lstm_2d()
layer_conv_lstm_3d()
layer_cropping_1d()

layer_cropping_2d()
layer_cropping_3d()
layer_dense()
layer_depthwise_conv_1d()
layer_depthwise_conv_2d()
layer_discretization()
layer_dot()
layer_dropout()
layer_einsum_dense()
layer_embedding()
layer_feature_space()
layer_flatten()
layer_flax_module_wrapper()
layer_gaussian_dropout()
layer_gaussian_noise()
layer_global_average_pooling_1d()
layer_global_average_pooling_2d()
layer_global_average_pooling_3d()
layer_global_max_pooling_1d()
layer_global_max_pooling_3d()
layer_group_normalization()
layer_group_query_attention()
layer_gru()
layer_hashed_crossing()
layer_hashing()
layer_identity()
layer_integer_lookup()
layer_jax_model_wrapper()
layer_lambda()
layer_layer_normalization()
layer_lstm()
layer_masking()
layer_max_pooling_1d()
layer_max_pooling_2d()
layer_max_pooling_3d()
layer_maximum()
layer_mel_spectrogram()
layer_minimum()
layer_multi_head_attention()
layer_multiply()
layer_normalization()
layer_permute()
layer_random_brightness()
layer_random_contrast()
layer_random_crop()
layer_random_flip()
layer_random_rotation()
layer_random_translation()

layer_random_zoom()
layer_repeat_vector()
layer_rescaling()
layer_reshape()
layer_resizing()
layer_rnn()
layer_separable_conv_1d()
layer_separable_conv_2d()
layer_simple_rnn()
layer_spatial_dropout_1d()
layer_spatial_dropout_2d()
layer_spatial_dropout_3d()
layer_spectral_normalization()
layer_string_lookup()
layer_subtract()
layer_text_vectorization()
layer_tfsm()
layer_time_distributed()
layer_torch_module_wrapper()
layer_unit_normalization()
layer_upsampling_1d()
layer_upsampling_2d()
layer_upsampling_3d()
layer_zero_padding_1d()
layer_zero_padding_2d()
layer_zero_padding_3d()
rnn_cell_gru()
rnn_cell_lstm()
rnn_cell_simple()
rnn_cells_stack()

---

layer_global_max_pooling_3d

Global max pooling operation for 3D data.

Description

Global max pooling operation for 3D data.

Usage

```
layer_global_max_pooling_3d(object, data_format = NULL, keepdims = FALSE, ...)
```

Arguments

object	Object to compose the layer with. A tensor, array, or sequential model.

`data_format`	string, either `"channels_last"` or `"channels_first"`. The ordering of the dimensions in the inputs. `"channels_last"` corresponds to inputs with shape `(batch, spatial_dim1, spatial_dim2, spatial_dim3, channels)` while `"channels_first"` corresponds to inputs with shape `(batch, channels, spatial_dim1, spatial_di` It defaults to the `image_data_format` value found in your Keras config file at `~/.keras/keras.json`. If you never set it, then it will be `"channels_last"`.
`keepdims`	A boolean, whether to keep the temporal dimension or not. If `keepdims` is `FALSE` (default), the rank of the tensor is reduced for spatial dimensions. If `keepdims` is `TRUE`, the spatial dimension are retained with length 1. The behavior is the same as for `tf$reduce_mean()` or `op_mean()`.
`...`	For forward/backward compatability.

Value

The return value depends on the value provided for the first argument. If `object` is:

- a `keras_model_sequential()`, then the layer is added to the sequential model (which is modified in place). To enable piping, the sequential model is also returned, invisibly.
- a `keras_input()`, then the output tensor from calling `layer(input)` is returned.
- `NULL` or missing, then a `Layer` instance is returned.

Input Shape

- If `data_format='channels_last'`: 5D tensor with shape: `(batch_size, spatial_dim1, spatial_dim2, spatial_`
- If `data_format='channels_first'`: 5D tensor with shape: `(batch_size, channels, spatial_dim1, spatial_di`

Output Shape

- If `keepdims=FALSE`: 2D tensor with shape `(batch_size, channels)`.
- If `keepdims=TRUE`:
  - If `data_format="channels_last"`: 5D tensor with shape `(batch_size, 1, 1, 1, channels)`
  - If `data_format="channels_first"`: 5D tensor with shape `(batch_size, channels, 1, 1, 1)`

Examples

```
x <- random_uniform(c(2, 4, 5, 4, 3))
y <- x |> layer_global_max_pooling_3d()
shape(y)
```

```
## shape(2, 3)
```

See Also

- https://keras.io/api/layers/pooling_layers/global_max_pooling3d#globalmaxpooling3d-class

Other pooling layers:
layer_average_pooling_1d()
layer_average_pooling_2d()
layer_average_pooling_3d()
layer_global_average_pooling_1d()
layer_global_average_pooling_2d()
layer_global_average_pooling_3d()
layer_global_max_pooling_1d()
layer_global_max_pooling_2d()
layer_max_pooling_1d()
layer_max_pooling_2d()
layer_max_pooling_3d()

Other layers:
Layer()
layer_activation()
layer_activation_elu()
layer_activation_leaky_relu()
layer_activation_parametric_relu()
layer_activation_relu()
layer_activation_softmax()
layer_activity_regularization()
layer_add()
layer_additive_attention()
layer_alpha_dropout()
layer_attention()
layer_average()
layer_average_pooling_1d()
layer_average_pooling_2d()
layer_average_pooling_3d()
layer_batch_normalization()
layer_bidirectional()
layer_category_encoding()
layer_center_crop()
layer_concatenate()
layer_conv_1d()
layer_conv_1d_transpose()
layer_conv_2d()
layer_conv_2d_transpose()
layer_conv_3d()
layer_conv_3d_transpose()
layer_conv_lstm_1d()
layer_conv_lstm_2d()
layer_conv_lstm_3d()
layer_cropping_1d()

layer_cropping_2d()
layer_cropping_3d()
layer_dense()
layer_depthwise_conv_1d()
layer_depthwise_conv_2d()
layer_discretization()
layer_dot()
layer_dropout()
layer_einsum_dense()
layer_embedding()
layer_feature_space()
layer_flatten()
layer_flax_module_wrapper()
layer_gaussian_dropout()
layer_gaussian_noise()
layer_global_average_pooling_1d()
layer_global_average_pooling_2d()
layer_global_average_pooling_3d()
layer_global_max_pooling_1d()
layer_global_max_pooling_2d()
layer_group_normalization()
layer_group_query_attention()
layer_gru()
layer_hashed_crossing()
layer_hashing()
layer_identity()
layer_integer_lookup()
layer_jax_model_wrapper()
layer_lambda()
layer_layer_normalization()
layer_lstm()
layer_masking()
layer_max_pooling_1d()
layer_max_pooling_2d()
layer_max_pooling_3d()
layer_maximum()
layer_mel_spectrogram()
layer_minimum()
layer_multi_head_attention()
layer_multiply()
layer_normalization()
layer_permute()
layer_random_brightness()
layer_random_contrast()
layer_random_crop()
layer_random_flip()
layer_random_rotation()
layer_random_translation()

layer_random_zoom()
layer_repeat_vector()
layer_rescaling()
layer_reshape()
layer_resizing()
layer_rnn()
layer_separable_conv_1d()
layer_separable_conv_2d()
layer_simple_rnn()
layer_spatial_dropout_1d()
layer_spatial_dropout_2d()
layer_spatial_dropout_3d()
layer_spectral_normalization()
layer_string_lookup()
layer_subtract()
layer_text_vectorization()
layer_tfsm()
layer_time_distributed()
layer_torch_module_wrapper()
layer_unit_normalization()
layer_upsampling_1d()
layer_upsampling_2d()
layer_upsampling_3d()
layer_zero_padding_1d()
layer_zero_padding_2d()
layer_zero_padding_3d()
rnn_cell_gru()
rnn_cell_lstm()
rnn_cell_simple()
rnn_cells_stack()

`layer_group_normalization`

Group normalization layer.

Description

Group Normalization divides the channels into groups and computes within each group the mean and variance for normalization. Empirically, its accuracy is more stable than batch norm in a wide range of small batch sizes, if learning rate is adjusted linearly with batch sizes.

Relation to Layer Normalization: If the number of groups is set to 1, then this operation becomes nearly identical to Layer Normalization (see Layer Normalization docs for details).

Relation to Instance Normalization: If the number of groups is set to the input dimension (number of groups is equal to number of channels), then this operation becomes identical to Instance Normalization. You can achieve this via `groups=-1`.

Usage

```
layer_group_normalization(
  object,
  groups = 32L,
  axis = -1L,
  epsilon = 0.001,
  center = TRUE,
  scale = TRUE,
  beta_initializer = "zeros",
  gamma_initializer = "ones",
  beta_regularizer = NULL,
  gamma_regularizer = NULL,
  beta_constraint = NULL,
  gamma_constraint = NULL,
  ...
)
```

Arguments

`object`	Object to compose the layer with. A tensor, array, or sequential model.
`groups`	Integer, the number of groups for Group Normalization. Can be in the range `[1, N]` where N is the input dimension. The input dimension must be divisible by the number of groups. Defaults to 32.
`axis`	Integer or List/Tuple. The axis or axes to normalize across. Typically, this is the features axis/axes. The left-out axes are typically the batch axis/axes. -1 is the last dimension in the input. Defaults to `-1`.
`epsilon`	Small float added to variance to avoid dividing by zero. Defaults to 1e-3.
`center`	If `TRUE`, add offset of `beta` to normalized tensor. If `FALSE`, `beta` is ignored. Defaults to `TRUE`.
`scale`	If `TRUE`, multiply by `gamma`. If `FALSE`, `gamma` is not used. When the next layer is linear (also e.g. `relu`), this can be disabled since the scaling will be done by the next layer. Defaults to `TRUE`.
`beta_initializer`	Initializer for the beta weight. Defaults to zeros.
`gamma_initializer`	Initializer for the gamma weight. Defaults to ones.
`beta_regularizer`	Optional regularizer for the beta weight. `NULL` by default.
`gamma_regularizer`	Optional regularizer for the gamma weight. `NULL` by default.
`beta_constraint`	Optional constraint for the beta weight. `NULL` by default.
`gamma_constraint`	Optional constraint for the gamma weight. `NULL` by default.
`...`	For forward/backward compatability.

Value

The return value depends on the value provided for the first argument. If `object` is:

- a `keras_model_sequential()`, then the layer is added to the sequential model (which is modified in place). To enable piping, the sequential model is also returned, invisibly.
- a `keras_input()`, then the output tensor from calling `layer(input)` is returned.
- `NULL` or missing, then a `Layer` instance is returned.

Input Shape

Arbitrary. Use the keyword argument `input_shape` (tuple of integers, does not include the samples axis) when using this layer as the first layer in a model.

Output Shape

Same shape as input. **kwargs: Base layer keyword arguments (e.g. `name` and `dtype`).

Reference

- Yuxin Wu & Kaiming He, 2018

See Also

- `https://keras.io/api/layers/normalization_layers/group_normalization#groupnormalization-class`

Other normalization layers:
`layer_batch_normalization()`
`layer_layer_normalization()`
`layer_spectral_normalization()`
`layer_unit_normalization()`

Other layers:
`Layer()`
`layer_activation()`
`layer_activation_elu()`
`layer_activation_leaky_relu()`
`layer_activation_parametric_relu()`
`layer_activation_relu()`
`layer_activation_softmax()`
`layer_activity_regularization()`
`layer_add()`
`layer_additive_attention()`
`layer_alpha_dropout()`
`layer_attention()`
`layer_average()`
`layer_average_pooling_1d()`
`layer_average_pooling_2d()`
`layer_average_pooling_3d()`
`layer_batch_normalization()`

layer_bidirectional()
layer_category_encoding()
layer_center_crop()
layer_concatenate()
layer_conv_1d()
layer_conv_1d_transpose()
layer_conv_2d()
layer_conv_2d_transpose()
layer_conv_3d()
layer_conv_3d_transpose()
layer_conv_lstm_1d()
layer_conv_lstm_2d()
layer_conv_lstm_3d()
layer_cropping_1d()
layer_cropping_2d()
layer_cropping_3d()
layer_dense()
layer_depthwise_conv_1d()
layer_depthwise_conv_2d()
layer_discretization()
layer_dot()
layer_dropout()
layer_einsum_dense()
layer_embedding()
layer_feature_space()
layer_flatten()
layer_flax_module_wrapper()
layer_gaussian_dropout()
layer_gaussian_noise()
layer_global_average_pooling_1d()
layer_global_average_pooling_2d()
layer_global_average_pooling_3d()
layer_global_max_pooling_1d()
layer_global_max_pooling_2d()
layer_global_max_pooling_3d()
layer_group_query_attention()
layer_gru()
layer_hashed_crossing()
layer_hashing()
layer_identity()
layer_integer_lookup()
layer_jax_model_wrapper()
layer_lambda()
layer_layer_normalization()
layer_lstm()
layer_masking()
layer_max_pooling_1d()
layer_max_pooling_2d()

layer_max_pooling_3d()
layer_maximum()
layer_mel_spectrogram()
layer_minimum()
layer_multi_head_attention()
layer_multiply()
layer_normalization()
layer_permute()
layer_random_brightness()
layer_random_contrast()
layer_random_crop()
layer_random_flip()
layer_random_rotation()
layer_random_translation()
layer_random_zoom()
layer_repeat_vector()
layer_rescaling()
layer_reshape()
layer_resizing()
layer_rnn()
layer_separable_conv_1d()
layer_separable_conv_2d()
layer_simple_rnn()
layer_spatial_dropout_1d()
layer_spatial_dropout_2d()
layer_spatial_dropout_3d()
layer_spectral_normalization()
layer_string_lookup()
layer_subtract()
layer_text_vectorization()
layer_tfsm()
layer_time_distributed()
layer_torch_module_wrapper()
layer_unit_normalization()
layer_upsampling_1d()
layer_upsampling_2d()
layer_upsampling_3d()
layer_zero_padding_1d()
layer_zero_padding_2d()
layer_zero_padding_3d()
rnn_cell_gru()
rnn_cell_lstm()
rnn_cell_simple()
rnn_cells_stack()

layer_group_query_attention
Grouped Query Attention layer.

Description

This is an implementation of grouped-query attention introduced by Ainslie et al., 2023. Here num_key_value_heads denotes number of groups, setting num_key_value_heads to 1 is equivalent to multi-query attention, and when num_key_value_heads is equal to num_query_heads it is equivalent to multi-head attention.

This layer first projects query, key, and value tensors. Then, key and value are repeated to match the number of heads of query.

Then, the query is scaled and dot-producted with key tensors. These are softmaxed to obtain attention probabilities. The value tensors are then interpolated by these probabilities and concatenated back to a single tensor.

Usage

```
layer_group_query_attention(
  object,
  head_dim,
  num_query_heads,
  num_key_value_heads,
  dropout = 0,
  use_bias = TRUE,
  kernel_initializer = "glorot_uniform",
  bias_initializer = "zeros",
  kernel_regularizer = NULL,
  bias_regularizer = NULL,
  activity_regularizer = NULL,
  kernel_constraint = NULL,
  bias_constraint = NULL,
  ...
)
```

Arguments

object	Object to compose the layer with. A tensor, array, or sequential model.
head_dim	Size of each attention head.
num_query_heads	Number of query attention heads.
num_key_value_heads	Number of key and value attention heads.
dropout	Dropout probability.
use_bias	Boolean, whether the dense layers use bias vectors/matrices.
kernel_initializer	Initializer for dense layer kernels.

`bias_initializer`
: Initializer for dense layer biases.

`kernel_regularizer`
: Regularizer for dense layer kernels.

`bias_regularizer`
: Regularizer for dense layer biases.

`activity_regularizer`
: Regularizer for dense layer activity.

`kernel_constraint`
: Constraint for dense layer kernels.

`bias_constraint`
: Constraint for dense layer kernels.

`...` For forward/backward compatability.

Value

attention_output: Result of the computation, of shape `(batch_dim, target_seq_len, feature_dim)`, where `target_seq_len` is for target sequence length and `feature_dim` is the query input last dim. attention_scores: (Optional) attention coefficients of shape `(batch_dim, num_query_heads, target_seq_len, source_se`

Call Arguments

- `query`: Query tensor of shape `(batch_dim, target_seq_len, feature_dim)`, where `batch_dim` is batch size, `target_seq_len` is the length of target sequence, and `feature_dim` is dimension of feature.
- `value`: Value tensor of shape `(batch_dim, source_seq_len, feature_dim)`, where `batch_dim` is batch size, `source_seq_len` is the length of source sequence, and `feature_dim` is dimension of feature.
- `key`: Optional key tensor of shape `(batch_dim, source_seq_len, feature_dim)`. If not given, will use `value` for both `key` and `value`, which is most common case.
- `attention_mask`: A boolean mask of shape `(batch_dim, target_seq_len, source_seq_len)`, that prevents attention to certain positions. The boolean mask specifies which query elements can attend to which key elements, where 1 indicates attention and 0 indicates no attention. Broadcasting can happen for the missing batch dimensions and the head dimension.
- `return_attention_scores`: A boolean to indicate whether the output should be `(attention_output, attention_sc` if `TRUE`, or `attention_output` if `FALSE`. Defaults to `FALSE`.
- `training`: Python boolean indicating whether the layer should behave in training mode (adding dropout) or in inference mode (no dropout). Will go with either using the training mode of the parent layer/model or `FALSE` (inference) if there is no parent layer.
- `use_causal_mask`: A boolean to indicate whether to apply a causal mask to prevent tokens from attending to future tokens (e.g., used in a decoder Transformer).

See Also

Other attention layers:
`layer_additive_attention()`

layer_attention()
layer_multi_head_attention()

Other layers:
Layer()
layer_activation()
layer_activation_elu()
layer_activation_leaky_relu()
layer_activation_parametric_relu()
layer_activation_relu()
layer_activation_softmax()
layer_activity_regularization()
layer_add()
layer_additive_attention()
layer_alpha_dropout()
layer_attention()
layer_average()
layer_average_pooling_1d()
layer_average_pooling_2d()
layer_average_pooling_3d()
layer_batch_normalization()
layer_bidirectional()
layer_category_encoding()
layer_center_crop()
layer_concatenate()
layer_conv_1d()
layer_conv_1d_transpose()
layer_conv_2d()
layer_conv_2d_transpose()
layer_conv_3d()
layer_conv_3d_transpose()
layer_conv_lstm_1d()
layer_conv_lstm_2d()
layer_conv_lstm_3d()
layer_cropping_1d()
layer_cropping_2d()
layer_cropping_3d()
layer_dense()
layer_depthwise_conv_1d()
layer_depthwise_conv_2d()
layer_discretization()
layer_dot()
layer_dropout()
layer_einsum_dense()
layer_embedding()
layer_feature_space()
layer_flatten()
layer_flax_module_wrapper()

layer_gaussian_dropout()
layer_gaussian_noise()
layer_global_average_pooling_1d()
layer_global_average_pooling_2d()
layer_global_average_pooling_3d()
layer_global_max_pooling_1d()
layer_global_max_pooling_2d()
layer_global_max_pooling_3d()
layer_group_normalization()
layer_gru()
layer_hashed_crossing()
layer_hashing()
layer_identity()
layer_integer_lookup()
layer_jax_model_wrapper()
layer_lambda()
layer_layer_normalization()
layer_lstm()
layer_masking()
layer_max_pooling_1d()
layer_max_pooling_2d()
layer_max_pooling_3d()
layer_maximum()
layer_mel_spectrogram()
layer_minimum()
layer_multi_head_attention()
layer_multiply()
layer_normalization()
layer_permute()
layer_random_brightness()
layer_random_contrast()
layer_random_crop()
layer_random_flip()
layer_random_rotation()
layer_random_translation()
layer_random_zoom()
layer_repeat_vector()
layer_rescaling()
layer_reshape()
layer_resizing()
layer_rnn()
layer_separable_conv_1d()
layer_separable_conv_2d()
layer_simple_rnn()
layer_spatial_dropout_1d()
layer_spatial_dropout_2d()
layer_spatial_dropout_3d()
layer_spectral_normalization()

layer_string_lookup()
layer_subtract()
layer_text_vectorization()
layer_tfsm()
layer_time_distributed()
layer_torch_module_wrapper()
layer_unit_normalization()
layer_upsampling_1d()
layer_upsampling_2d()
layer_upsampling_3d()
layer_zero_padding_1d()
layer_zero_padding_2d()
layer_zero_padding_3d()
rnn_cell_gru()
rnn_cell_lstm()
rnn_cell_simple()
rnn_cells_stack()

layer_gru *Gated Recurrent Unit - Cho et al. 2014.*

Description

Based on available runtime hardware and constraints, this layer will choose different implementations (cuDNN-based or backend-native) to maximize the performance. If a GPU is available and all the arguments to the layer meet the requirement of the cuDNN kernel (see below for details), the layer will use a fast cuDNN implementation when using the TensorFlow backend.

The requirements to use the cuDNN implementation are:

1. `activation` == `tanh`
2. `recurrent_activation` == `sigmoid`
3. `dropout` == 0 and `recurrent_dropout` == 0
4. `unroll` is `FALSE`
5. `use_bias` is `TRUE`
6. `reset_after` is `TRUE`
7. Inputs, if use masking, are strictly right-padded.
8. Eager execution is enabled in the outermost context.

There are two variants of the GRU implementation. The default one is based on v3 and has reset gate applied to hidden state before matrix multiplication. The other one is based on original and has the order reversed.

The second variant is compatible with CuDNNGRU (GPU-only) and allows inference on CPU. Thus it has separate biases for `kernel` and `recurrent_kernel`. To use this variant, set `reset_after=TRUE` and `recurrent_activation='sigmoid'`.

For example:

```
inputs <- random_uniform(c(32, 10, 8))
outputs <- inputs |> layer_gru(4)
shape(outputs)
```

```
## shape(32, 4)
```

```
# (32, 4)
gru <- layer_gru(, 4, return_sequences = TRUE, return_state = TRUE)
c(whole_sequence_output, final_state) %<-% gru(inputs)
shape(whole_sequence_output)
```

```
## shape(32, 10, 4)
```

```
shape(final_state)
```

```
## shape(32, 4)
```

Usage

```
layer_gru(
  object,
  units,
  activation = "tanh",
  recurrent_activation = "sigmoid",
  use_bias = TRUE,
  kernel_initializer = "glorot_uniform",
  recurrent_initializer = "orthogonal",
  bias_initializer = "zeros",
  kernel_regularizer = NULL,
  recurrent_regularizer = NULL,
  bias_regularizer = NULL,
  activity_regularizer = NULL,
  kernel_constraint = NULL,
  recurrent_constraint = NULL,
  bias_constraint = NULL,
  dropout = 0,
  recurrent_dropout = 0,
  seed = NULL,
  return_sequences = FALSE,
  return_state = FALSE,
  go_backwards = FALSE,
  stateful = FALSE,
  unroll = FALSE,
  reset_after = TRUE,
  use_cudnn = "auto",
  ...
)
```

Arguments

`object`	Object to compose the layer with. A tensor, array, or sequential model.
`units`	Positive integer, dimensionality of the output space.
`activation`	Activation function to use. Default: hyperbolic tangent (`tanh`). If you pass `NULL`, no activation is applied (ie. "linear" activation: `a(x) = x`).
`recurrent_activation`	Activation function to use for the recurrent step. Default: sigmoid (`sigmoid`). If you pass `NULL`, no activation is applied (ie. "linear" activation: `a(x) = x`).
`use_bias`	Boolean, (default `TRUE`), whether the layer should use a bias vector.
`kernel_initializer`	Initializer for the `kernel` weights matrix, used for the linear transformation of the inputs. Default: `"glorot_uniform"`.
`recurrent_initializer`	Initializer for the `recurrent_kernel` weights matrix, used for the linear transformation of the recurrent state. Default: `"orthogonal"`.
`bias_initializer`	Initializer for the bias vector. Default: `"zeros"`.
`kernel_regularizer`	Regularizer function applied to the `kernel` weights matrix. Default: `NULL`.
`recurrent_regularizer`	Regularizer function applied to the `recurrent_kernel` weights matrix. Default: `NULL`.
`bias_regularizer`	Regularizer function applied to the bias vector. Default: `NULL`.
`activity_regularizer`	Regularizer function applied to the output of the layer (its "activation"). Default: `NULL`.
`kernel_constraint`	Constraint function applied to the `kernel` weights matrix. Default: `NULL`.
`recurrent_constraint`	Constraint function applied to the `recurrent_kernel` weights matrix. Default: `NULL`.
`bias_constraint`	Constraint function applied to the bias vector. Default: `NULL`.
`dropout`	Float between 0 and 1. Fraction of the units to drop for the linear transformation of the inputs. Default: 0.
`recurrent_dropout`	Float between 0 and 1. Fraction of the units to drop for the linear transformation of the recurrent state. Default: 0.
`seed`	Random seed for dropout.
`return_sequences`	Boolean. Whether to return the last output in the output sequence, or the full sequence. Default: `FALSE`.
`return_state`	Boolean. Whether to return the last state in addition to the output. Default: `FALSE`.

`go_backwards`	Boolean (default `FALSE`). If `TRUE`, process the input sequence backwards and return the reversed sequence.
`stateful`	Boolean (default: `FALSE`). If `TRUE`, the last state for each sample at index i in a batch will be used as initial state for the sample of index i in the following batch.
`unroll`	Boolean (default: `FALSE`). If `TRUE`, the network will be unrolled, else a symbolic loop will be used. Unrolling can speed-up a RNN, although it tends to be more memory-intensive. Unrolling is only suitable for short sequences.
`reset_after`	GRU convention (whether to apply reset gate after or before matrix multiplication). `FALSE` is `"before"`, `TRUE` is `"after"` (default and cuDNN compatible).
`use_cudnn`	Whether to use a cuDNN-backed implementation. `"auto"` will attempt to use cuDNN when feasible, and will fallback to the default implementation if not.
`...`	For forward/backward compatability.

Value

The return value depends on the value provided for the first argument. If `object` is:

- a `keras_model_sequential()`, then the layer is added to the sequential model (which is modified in place). To enable piping, the sequential model is also returned, invisibly.
- a `keras_input()`, then the output tensor from calling `layer(input)` is returned.
- `NULL` or missing, then a `Layer` instance is returned.

Call Arguments

- `inputs`: A 3D tensor, with shape `(batch, timesteps, feature)`.
- `mask`: Binary tensor of shape `(samples, timesteps)` indicating whether a given timestep should be masked (optional). An individual `TRUE` entry indicates that the corresponding timestep should be utilized, while a `FALSE` entry indicates that the corresponding timestep should be ignored. Defaults to `NULL`.
- `training`: Python boolean indicating whether the layer should behave in training mode or in inference mode. This argument is passed to the cell when calling it. This is only relevant if `dropout` or `recurrent_dropout` is used (optional). Defaults to `NULL`.
- `initial_state`: List of initial state tensors to be passed to the first call of the cell (optional, `NULL` causes creation of zero-filled initial state tensors). Defaults to `NULL`.

See Also

- https://keras.io/api/layers/recurrent_layers/gru#gru-class

Other gru rnn layers:
`rnn_cell_gru()`

Other rnn layers:
`layer_bidirectional()`
`layer_conv_lstm_1d()`
`layer_conv_lstm_2d()`
`layer_conv_lstm_3d()`

layer_lstm()
layer_rnn()
layer_simple_rnn()
layer_time_distributed()
rnn_cell_gru()
rnn_cell_lstm()
rnn_cell_simple()
rnn_cells_stack()

Other layers:
Layer()
layer_activation()
layer_activation_elu()
layer_activation_leaky_relu()
layer_activation_parametric_relu()
layer_activation_relu()
layer_activation_softmax()
layer_activity_regularization()
layer_add()
layer_additive_attention()
layer_alpha_dropout()
layer_attention()
layer_average()
layer_average_pooling_1d()
layer_average_pooling_2d()
layer_average_pooling_3d()
layer_batch_normalization()
layer_bidirectional()
layer_category_encoding()
layer_center_crop()
layer_concatenate()
layer_conv_1d()
layer_conv_1d_transpose()
layer_conv_2d()
layer_conv_2d_transpose()
layer_conv_3d()
layer_conv_3d_transpose()
layer_conv_lstm_1d()
layer_conv_lstm_2d()
layer_conv_lstm_3d()
layer_cropping_1d()
layer_cropping_2d()
layer_cropping_3d()
layer_dense()
layer_depthwise_conv_1d()
layer_depthwise_conv_2d()
layer_discretization()
layer_dot()

layer_dropout()
layer_einsum_dense()
layer_embedding()
layer_feature_space()
layer_flatten()
layer_flax_module_wrapper()
layer_gaussian_dropout()
layer_gaussian_noise()
layer_global_average_pooling_1d()
layer_global_average_pooling_2d()
layer_global_average_pooling_3d()
layer_global_max_pooling_1d()
layer_global_max_pooling_2d()
layer_global_max_pooling_3d()
layer_group_normalization()
layer_group_query_attention()
layer_hashed_crossing()
layer_hashing()
layer_identity()
layer_integer_lookup()
layer_jax_model_wrapper()
layer_lambda()
layer_layer_normalization()
layer_lstm()
layer_masking()
layer_max_pooling_1d()
layer_max_pooling_2d()
layer_max_pooling_3d()
layer_maximum()
layer_mel_spectrogram()
layer_minimum()
layer_multi_head_attention()
layer_multiply()
layer_normalization()
layer_permute()
layer_random_brightness()
layer_random_contrast()
layer_random_crop()
layer_random_flip()
layer_random_rotation()
layer_random_translation()
layer_random_zoom()
layer_repeat_vector()
layer_rescaling()
layer_reshape()
layer_resizing()
layer_rnn()
layer_separable_conv_1d()

layer_separable_conv_2d()
layer_simple_rnn()
layer_spatial_dropout_1d()
layer_spatial_dropout_2d()
layer_spatial_dropout_3d()
layer_spectral_normalization()
layer_string_lookup()
layer_subtract()
layer_text_vectorization()
layer_tfsm()
layer_time_distributed()
layer_torch_module_wrapper()
layer_unit_normalization()
layer_upsampling_1d()
layer_upsampling_2d()
layer_upsampling_3d()
layer_zero_padding_1d()
layer_zero_padding_2d()
layer_zero_padding_3d()
rnn_cell_gru()
rnn_cell_lstm()
rnn_cell_simple()
rnn_cells_stack()

layer_hashed_crossing *A preprocessing layer which crosses features using the "hashing trick".*

Description

This layer performs crosses of categorical features using the "hashing trick". Conceptually, the transformation can be thought of as: `hash(concatenate(features)) %% num_bins`.

This layer currently only performs crosses of scalar inputs and batches of scalar inputs. Valid input shapes are `(batch_size, 1)`, `(batch_size)` and `()`.

Note: This layer wraps `tf.keras.layers.HashedCrossing`. It cannot be used as part of the compiled computation graph of a model with any backend other than TensorFlow. It can however be used with any backend when running eagerly. It can also always be used as part of an input preprocessing pipeline with any backend (outside the model itself), which is how we recommend to use this layer.

Note: This layer is safe to use inside a `tfdatasets` pipeline (independently of which backend you're using).

Usage

```
layer_hashed_crossing(
  object,
  num_bins,
  output_mode = "int",
  sparse = FALSE,
  name = NULL,
  dtype = NULL,
  ...
)
```

Arguments

`object`	Object to compose the layer with. A tensor, array, or sequential model.
`num_bins`	Number of hash bins.
`output_mode`	Specification for the output of the layer. Values can be `"int"`, or `"one_hot"` configuring the layer as follows: • `"int"`: Return the integer bin indices directly. • `"one_hot"`: Encodes each individual element in the input into an array the same size as `num_bins`, containing a 1 at the input's bin index. Defaults to `"int"`.
`sparse`	Boolean. Only applicable to `"one_hot"` mode and only valid when using the TensorFlow backend. If `TRUE`, returns a `SparseTensor` instead of a dense `Tensor`. Defaults to `FALSE`.
`name`	String, name for the object
`dtype`	datatype (e.g., `"float32"`).
`...`	Keyword arguments to construct a layer.

Value

The return value depends on the value provided for the first argument. If `object` is:

- a `keras_model_sequential()`, then the layer is added to the sequential model (which is modified in place). To enable piping, the sequential model is also returned, invisibly.
- a `keras_input()`, then the output tensor from calling `layer(input)` is returned.
- `NULL` or missing, then a `Layer` instance is returned.

Examples

```
feat1 <- c('A', 'B', 'A', 'B', 'A') |> as.array()
feat2 <- c(101, 101, 101, 102, 102) |> as.integer() |> as.array()
```

Crossing two scalar features.

```
layer <- layer_hashed_crossing(num_bins = 5)
layer(list(feat1, feat2))
```

```
## tf.Tensor([1 4 1 1 3], shape=(5), dtype=int64)
```

Crossing and one-hotting two scalar features.

```
layer <- layer_hashed_crossing(num_bins = 5, output_mode = 'one_hot')
layer(list(feat1, feat2))

## tf.Tensor(
## [[0. 1. 0. 0. 0.]
##  [0. 0. 0. 0. 1.]
##  [0. 1. 0. 0. 0.]
##  [0. 1. 0. 0. 0.]
##  [0. 0. 0. 1. 0.]], shape=(5, 5), dtype=float32)
```

See Also

- https://keras.io/api/layers/preprocessing_layers/categorical/hashed_crossing#hashedcrossing-class

Other categorical features preprocessing layers:
layer_category_encoding()
layer_hashing()
layer_integer_lookup()
layer_string_lookup()

Other preprocessing layers:
layer_category_encoding()
layer_center_crop()
layer_discretization()
layer_feature_space()
layer_hashing()
layer_integer_lookup()
layer_mel_spectrogram()
layer_normalization()
layer_random_brightness()
layer_random_contrast()
layer_random_crop()
layer_random_flip()
layer_random_rotation()
layer_random_translation()
layer_random_zoom()
layer_rescaling()
layer_resizing()
layer_string_lookup()
layer_text_vectorization()

Other layers:
Layer()
layer_activation()
layer_activation_elu()

layer_activation_leaky_relu()
layer_activation_parametric_relu()
layer_activation_relu()
layer_activation_softmax()
layer_activity_regularization()
layer_add()
layer_additive_attention()
layer_alpha_dropout()
layer_attention()
layer_average()
layer_average_pooling_1d()
layer_average_pooling_2d()
layer_average_pooling_3d()
layer_batch_normalization()
layer_bidirectional()
layer_category_encoding()
layer_center_crop()
layer_concatenate()
layer_conv_1d()
layer_conv_1d_transpose()
layer_conv_2d()
layer_conv_2d_transpose()
layer_conv_3d()
layer_conv_3d_transpose()
layer_conv_lstm_1d()
layer_conv_lstm_2d()
layer_conv_lstm_3d()
layer_cropping_1d()
layer_cropping_2d()
layer_cropping_3d()
layer_dense()
layer_depthwise_conv_1d()
layer_depthwise_conv_2d()
layer_discretization()
layer_dot()
layer_dropout()
layer_einsum_dense()
layer_embedding()
layer_feature_space()
layer_flatten()
layer_flax_module_wrapper()
layer_gaussian_dropout()
layer_gaussian_noise()
layer_global_average_pooling_1d()
layer_global_average_pooling_2d()
layer_global_average_pooling_3d()
layer_global_max_pooling_1d()
layer_global_max_pooling_2d()

layer_global_max_pooling_3d()
layer_group_normalization()
layer_group_query_attention()
layer_gru()
layer_hashing()
layer_identity()
layer_integer_lookup()
layer_jax_model_wrapper()
layer_lambda()
layer_layer_normalization()
layer_lstm()
layer_masking()
layer_max_pooling_1d()
layer_max_pooling_2d()
layer_max_pooling_3d()
layer_maximum()
layer_mel_spectrogram()
layer_minimum()
layer_multi_head_attention()
layer_multiply()
layer_normalization()
layer_permute()
layer_random_brightness()
layer_random_contrast()
layer_random_crop()
layer_random_flip()
layer_random_rotation()
layer_random_translation()
layer_random_zoom()
layer_repeat_vector()
layer_rescaling()
layer_reshape()
layer_resizing()
layer_rnn()
layer_separable_conv_1d()
layer_separable_conv_2d()
layer_simple_rnn()
layer_spatial_dropout_1d()
layer_spatial_dropout_2d()
layer_spatial_dropout_3d()
layer_spectral_normalization()
layer_string_lookup()
layer_subtract()
layer_text_vectorization()
layer_tfsm()
layer_time_distributed()
layer_torch_module_wrapper()
layer_unit_normalization()

layer_upsampling_1d()
layer_upsampling_2d()
layer_upsampling_3d()
layer_zero_padding_1d()
layer_zero_padding_2d()
layer_zero_padding_3d()
rnn_cell_gru()
rnn_cell_lstm()
rnn_cell_simple()
rnn_cells_stack()

`layer_hashing` *A preprocessing layer which hashes and bins categorical features.*

Description

This layer transforms categorical inputs to hashed output. It element-wise converts a ints or strings to ints in a fixed range. The stable hash function uses `tensorflow::ops::Fingerprint` to produce the same output consistently across all platforms.

This layer uses FarmHash64 by default, which provides a consistent hashed output across different platforms and is stable across invocations, regardless of device and context, by mixing the input bits thoroughly.

If you want to obfuscate the hashed output, you can also pass a random `salt` argument in the constructor. In that case, the layer will use the SipHash64 hash function, with the `salt` value serving as additional input to the hash function.

Note: This layer internally uses TensorFlow. It cannot be used as part of the compiled computation graph of a model with any backend other than TensorFlow. It can however be used with any backend when running eagerly. It can also always be used as part of an input preprocessing pipeline with any backend (outside the model itself), which is how we recommend to use this layer.

Note: This layer is safe to use inside a `tf.data` pipeline (independently of which backend you're using).

Example (FarmHash64)

```
layer <- layer_hashing(num_bins = 3)
inp <- c('A', 'B', 'C', 'D', 'E') |> array(dim = c(5, 1))
layer(inp)

## tf.Tensor(
## [[1]
##  [0]
##  [1]
##  [1]
##  [2]], shape=(5, 1), dtype=int64)
```

Example (FarmHash64) with a mask value

```
layer <- layer_hashing(num_bins=3, mask_value='')
inp <- c('A', 'B', '', 'C', 'D') |> array(dim = c(5, 1))
layer(inp)
```

```
## tf.Tensor(
## [[1]
##  [1]
##  [0]
##  [2]
##  [2]], shape=(5, 1), dtype=int64)
```

Example (SipHash64)

```
layer <- layer_hashing(num_bins=3, salt=c(133, 137))
inp <- c('A', 'B', 'C', 'D', 'E') |> array(dim = c(5, 1))
layer(inp)
```

```
## tf.Tensor(
## [[1]
##  [2]
##  [1]
##  [0]
##  [2]], shape=(5, 1), dtype=int64)
```

Example (Siphash64 with a single integer, same as `salt=[133, 133]`)

```
layer <- layer_hashing(num_bins=3, salt=133)
inp <- c('A', 'B', 'C', 'D', 'E') |> array(dim = c(5, 1))
layer(inp)
```

```
## tf.Tensor(
## [[0]
##  [0]
##  [2]
##  [1]
##  [0]], shape=(5, 1), dtype=int64)
```

Usage

```
layer_hashing(
  object,
  num_bins,
  mask_value = NULL,
  salt = NULL,
  output_mode = "int",
  sparse = FALSE,
  ...
)
```

Arguments

`object`	Object to compose the layer with. A tensor, array, or sequential model.
`num_bins`	Number of hash bins. Note that this includes the `mask_value` bin, so the effective number of bins is `(num_bins - 1)` if `mask_value` is set.
`mask_value`	A value that represents masked inputs, which are mapped to index 0. `NULL` means no mask term will be added and the hashing will start at index 0. Defaults to `NULL`.
`salt`	A single unsigned integer or `NULL`. If passed, the hash function used will be SipHash64, with these values used as an additional input (known as a "salt" in cryptography). These should be non-zero. If `NULL`, uses the FarmHash64 hash function. It also supports list of 2 unsigned integer numbers, see reference paper for details. Defaults to `NULL`.
`output_mode`	Specification for the output of the layer. Values can be `"int"`, `"one_hot"`, `"multi_hot"`, or `"count"` configuring the layer as follows: • `"int"`: Return the integer bin indices directly. • `"one_hot"`: Encodes each individual element in the input into an array the same size as `num_bins`, containing a 1 at the input's bin index. If the last dimension is size 1, will encode on that dimension. If the last dimension is not size 1, will append a new dimension for the encoded output. • `"multi_hot"`: Encodes each sample in the input into a single array the same size as `num_bins`, containing a 1 for each bin index index present in the sample. Treats the last dimension as the sample dimension, if input shape is `(..., sample_length)`, output shape will be `(..., num_tokens)`. • `"count"`: As `"multi_hot"`, but the int array contains a count of the number of times the bin index appeared in the sample. Defaults to `"int"`.
`sparse`	Boolean. Only applicable to `"one_hot"`, `"multi_hot"`, and `"count"` output modes. Only supported with TensorFlow backend. If `TRUE`, returns a `SparseTensor` instead of a dense `Tensor`. Defaults to `FALSE`.
`...`	Keyword arguments to construct a layer.

Value

The return value depends on the value provided for the first argument. If `object` is:

- a `keras_model_sequential()`, then the layer is added to the sequential model (which is modified in place). To enable piping, the sequential model is also returned, invisibly.
- a `keras_input()`, then the output tensor from calling `layer(input)` is returned.
- `NULL` or missing, then a `Layer` instance is returned.

Input Shape

A single string, a list of strings, or an `int32` or `int64` tensor of shape `(batch_size, ...,)`.

Output Shape

An `int32` tensor of shape `(batch_size, ...)`.

Reference

- SipHash with salt

See Also

- `https://keras.io/api/layers/preprocessing_layers/categorical/hashing#hashing-class`

Other categorical features preprocessing layers:
`layer_category_encoding()`
`layer_hashed_crossing()`
`layer_integer_lookup()`
`layer_string_lookup()`

Other preprocessing layers:
`layer_category_encoding()`
`layer_center_crop()`
`layer_discretization()`
`layer_feature_space()`
`layer_hashed_crossing()`
`layer_integer_lookup()`
`layer_mel_spectrogram()`
`layer_normalization()`
`layer_random_brightness()`
`layer_random_contrast()`
`layer_random_crop()`
`layer_random_flip()`
`layer_random_rotation()`
`layer_random_translation()`
`layer_random_zoom()`
`layer_rescaling()`
`layer_resizing()`
`layer_string_lookup()`
`layer_text_vectorization()`

Other layers:
`Layer()`
`layer_activation()`
`layer_activation_elu()`
`layer_activation_leaky_relu()`
`layer_activation_parametric_relu()`
`layer_activation_relu()`

layer_activation_softmax()
layer_activity_regularization()
layer_add()
layer_additive_attention()
layer_alpha_dropout()
layer_attention()
layer_average()
layer_average_pooling_1d()
layer_average_pooling_2d()
layer_average_pooling_3d()
layer_batch_normalization()
layer_bidirectional()
layer_category_encoding()
layer_center_crop()
layer_concatenate()
layer_conv_1d()
layer_conv_1d_transpose()
layer_conv_2d()
layer_conv_2d_transpose()
layer_conv_3d()
layer_conv_3d_transpose()
layer_conv_lstm_1d()
layer_conv_lstm_2d()
layer_conv_lstm_3d()
layer_cropping_1d()
layer_cropping_2d()
layer_cropping_3d()
layer_dense()
layer_depthwise_conv_1d()
layer_depthwise_conv_2d()
layer_discretization()
layer_dot()
layer_dropout()
layer_einsum_dense()
layer_embedding()
layer_feature_space()
layer_flatten()
layer_flax_module_wrapper()
layer_gaussian_dropout()
layer_gaussian_noise()
layer_global_average_pooling_1d()
layer_global_average_pooling_2d()
layer_global_average_pooling_3d()
layer_global_max_pooling_1d()
layer_global_max_pooling_2d()
layer_global_max_pooling_3d()
layer_group_normalization()
layer_group_query_attention()

layer_gru()
layer_hashed_crossing()
layer_identity()
layer_integer_lookup()
layer_jax_model_wrapper()
layer_lambda()
layer_layer_normalization()
layer_lstm()
layer_masking()
layer_max_pooling_1d()
layer_max_pooling_2d()
layer_max_pooling_3d()
layer_maximum()
layer_mel_spectrogram()
layer_minimum()
layer_multi_head_attention()
layer_multiply()
layer_normalization()
layer_permute()
layer_random_brightness()
layer_random_contrast()
layer_random_crop()
layer_random_flip()
layer_random_rotation()
layer_random_translation()
layer_random_zoom()
layer_repeat_vector()
layer_rescaling()
layer_reshape()
layer_resizing()
layer_rnn()
layer_separable_conv_1d()
layer_separable_conv_2d()
layer_simple_rnn()
layer_spatial_dropout_1d()
layer_spatial_dropout_2d()
layer_spatial_dropout_3d()
layer_spectral_normalization()
layer_string_lookup()
layer_subtract()
layer_text_vectorization()
layer_tfsm()
layer_time_distributed()
layer_torch_module_wrapper()
layer_unit_normalization()
layer_upsampling_1d()
layer_upsampling_2d()
layer_upsampling_3d()

layer_zero_padding_1d()
layer_zero_padding_2d()
layer_zero_padding_3d()
rnn_cell_gru()
rnn_cell_lstm()
rnn_cell_simple()
rnn_cells_stack()

---

layer_identity *Identity layer.*

Description

This layer should be used as a placeholder when no operation is to be performed. The layer just returns its `inputs` argument as output.

Usage

```
layer_identity(object, ...)
```

Arguments

`object`	Object to compose the layer with. A tensor, array, or sequential model.
`...`	For forward/backward compatability.

Value

The return value depends on the value provided for the first argument. If `object` is:

- a `keras_model_sequential()`, then the layer is added to the sequential model (which is modified in place). To enable piping, the sequential model is also returned, invisibly.
- a `keras_input()`, then the output tensor from calling `layer(input)` is returned.
- `NULL` or missing, then a `Layer` instance is returned.

See Also

Other core layers:
layer_dense()
layer_einsum_dense()
layer_embedding()
layer_lambda()
layer_masking()

Other layers:
Layer()
layer_activation()

layer_activation_elu()
layer_activation_leaky_relu()
layer_activation_parametric_relu()
layer_activation_relu()
layer_activation_softmax()
layer_activity_regularization()
layer_add()
layer_additive_attention()
layer_alpha_dropout()
layer_attention()
layer_average()
layer_average_pooling_1d()
layer_average_pooling_2d()
layer_average_pooling_3d()
layer_batch_normalization()
layer_bidirectional()
layer_category_encoding()
layer_center_crop()
layer_concatenate()
layer_conv_1d()
layer_conv_1d_transpose()
layer_conv_2d()
layer_conv_2d_transpose()
layer_conv_3d()
layer_conv_3d_transpose()
layer_conv_lstm_1d()
layer_conv_lstm_2d()
layer_conv_lstm_3d()
layer_cropping_1d()
layer_cropping_2d()
layer_cropping_3d()
layer_dense()
layer_depthwise_conv_1d()
layer_depthwise_conv_2d()
layer_discretization()
layer_dot()
layer_dropout()
layer_einsum_dense()
layer_embedding()
layer_feature_space()
layer_flatten()
layer_flax_module_wrapper()
layer_gaussian_dropout()
layer_gaussian_noise()
layer_global_average_pooling_1d()
layer_global_average_pooling_2d()
layer_global_average_pooling_3d()
layer_global_max_pooling_1d()

layer_global_max_pooling_2d()
layer_global_max_pooling_3d()
layer_group_normalization()
layer_group_query_attention()
layer_gru()
layer_hashed_crossing()
layer_hashing()
layer_integer_lookup()
layer_jax_model_wrapper()
layer_lambda()
layer_layer_normalization()
layer_lstm()
layer_masking()
layer_max_pooling_1d()
layer_max_pooling_2d()
layer_max_pooling_3d()
layer_maximum()
layer_mel_spectrogram()
layer_minimum()
layer_multi_head_attention()
layer_multiply()
layer_normalization()
layer_permute()
layer_random_brightness()
layer_random_contrast()
layer_random_crop()
layer_random_flip()
layer_random_rotation()
layer_random_translation()
layer_random_zoom()
layer_repeat_vector()
layer_rescaling()
layer_reshape()
layer_resizing()
layer_rnn()
layer_separable_conv_1d()
layer_separable_conv_2d()
layer_simple_rnn()
layer_spatial_dropout_1d()
layer_spatial_dropout_2d()
layer_spatial_dropout_3d()
layer_spectral_normalization()
layer_string_lookup()
layer_subtract()
layer_text_vectorization()
layer_tfsm()
layer_time_distributed()
layer_torch_module_wrapper()

layer_unit_normalization()
layer_upsampling_1d()
layer_upsampling_2d()
layer_upsampling_3d()
layer_zero_padding_1d()
layer_zero_padding_2d()
layer_zero_padding_3d()
rnn_cell_gru()
rnn_cell_lstm()
rnn_cell_simple()
rnn_cells_stack()

`layer_integer_lookup` *A preprocessing layer that maps integers to (possibly encoded) indices.*

Description

This layer maps a set of arbitrary integer input tokens into indexed integer output via a table-based vocabulary lookup. The layer's output indices will be contiguously arranged up to the maximum vocab size, even if the input tokens are non-continguous or unbounded. The layer supports multiple options for encoding the output via `output_mode`, and has optional support for out-of-vocabulary (OOV) tokens and masking.

The vocabulary for the layer must be either supplied on construction or learned via `adapt()`. During `adapt()`, the layer will analyze a data set, determine the frequency of individual integer tokens, and create a vocabulary from them. If the vocabulary is capped in size, the most frequent tokens will be used to create the vocabulary and all others will be treated as OOV.

There are two possible output modes for the layer. When `output_mode` is `"int"`, input integers are converted to their index in the vocabulary (an integer). When `output_mode` is `"multi_hot"`, `"count"`, or `"tf_idf"`, input integers are encoded into an array where each dimension corresponds to an element in the vocabulary.

The vocabulary can optionally contain a mask token as well as an OOV token (which can optionally occupy multiple indices in the vocabulary, as set by `num_oov_indices`). The position of these tokens in the vocabulary is fixed. When `output_mode` is `"int"`, the vocabulary will begin with the mask token at index 0, followed by OOV indices, followed by the rest of the vocabulary. When `output_mode` is `"multi_hot"`, `"count"`, or `"tf_idf"` the vocabulary will begin with OOV indices and instances of the mask token will be dropped.

Note: This layer uses TensorFlow internally. It cannot be used as part of the compiled computation graph of a model with any backend other than TensorFlow. It can however be used with any backend when running eagerly. It can also always be used as part of an input preprocessing pipeline with any backend (outside the model itself), which is how we recommend to use this layer.

Note: This layer is safe to use inside a `tf.data` pipeline (independently of which backend you're using).

Usage

```
layer_integer_lookup(
  object,
  max_tokens = NULL,
  num_oov_indices = 1L,
  mask_token = NULL,
  oov_token = -1L,
  vocabulary = NULL,
  vocabulary_dtype = "int64",
  idf_weights = NULL,
  invert = FALSE,
  output_mode = "int",
  sparse = FALSE,
  pad_to_max_tokens = FALSE,
  name = NULL,
  ...
)
```

Arguments

`object`	Object to compose the layer with. A tensor, array, or sequential model.
`max_tokens`	Maximum size of the vocabulary for this layer. This should only be specified when adapting the vocabulary or when setting `pad_to_max_tokens=TRUE`. If NULL, there is no cap on the size of the vocabulary. Note that this size includes the OOV and mask tokens. Defaults to `NULL`.
`num_oov_indices`	The number of out-of-vocabulary tokens to use. If this value is more than 1, OOV inputs are modulated to determine their OOV value. If this value is 0, OOV inputs will cause an error when calling the layer. Defaults to `1`.
`mask_token`	An integer token that represents masked inputs. When `output_mode` is `"int"`, the token is included in vocabulary and mapped to index 0. In other output modes, the token will not appear in the vocabulary and instances of the mask token in the input will be dropped. If set to NULL, no mask term will be added. Defaults to `NULL`.
`oov_token`	Only used when `invert` is `TRUE`. The token to return for OOV indices. Defaults to `-1`.
`vocabulary`	Optional. Either an array of integers or a string path to a text file. If passing an array, can pass a list, list, 1D NumPy array, or 1D tensor containing the integer vocbulary terms. If passing a file path, the file should contain one line per term in the vocabulary. If this argument is set, there is no need to `adapt()` the layer.
`vocabulary_dtype`	The dtype of the vocabulary terms, for example `"int64"` or `"int32"`. Defaults to `"int64"`.
`idf_weights`	Only valid when `output_mode` is `"tf_idf"`. A list, list, 1D NumPy array, or 1D tensor or the same length as the vocabulary, containing the floating point inverse document frequency weights, which will be multiplied by per sample

term counts for the final TF-IDF weight. If the `vocabulary` argument is set, and `output_mode` is `"tf_idf"`, this argument must be supplied.

`invert`	Only valid when `output_mode` is `"int"`. If `TRUE`, this layer will map indices to vocabulary items instead of mapping vocabulary items to indices. Defaults to `FALSE`.
`output_mode`	Specification for the output of the layer. Values can be `"int"`, `"one_hot"`, `"multi_hot"`, `"count"`, or `"tf_idf"` configuring the layer as follows: • `"int"`: Return the vocabulary indices of the input tokens. • `"one_hot"`: Encodes each individual element in the input into an array the same size as the vocabulary, containing a 1 at the element index. If the last dimension is size 1, will encode on that dimension. If the last dimension is not size 1, will append a new dimension for the encoded output. • `"multi_hot"`: Encodes each sample in the input into a single array the same size as the vocabulary, containing a 1 for each vocabulary term present in the sample. Treats the last dimension as the sample dimension, if input shape is `(..., sample_length)`, output shape will be `(..., num_tokens)`. • `"count"`: As `"multi_hot"`, but the int array contains a count of the number of times the token at that index appeared in the sample. • `"tf_idf"`: As `"multi_hot"`, but the TF-IDF algorithm is applied to find the value in each token slot. For `"int"` output, any shape of input and output is supported. For all other output modes, currently only output up to rank 2 is supported. Defaults to `"int"`.
`sparse`	Boolean. Only applicable to `"multi_hot"`, `"count"`, and `"tf_idf"` output modes. Only supported with TensorFlow backend. If `TRUE`, returns a `SparseTensor` instead of a dense `Tensor`. Defaults to `FALSE`.
`pad_to_max_tokens`	Only applicable when `output_mode` is `"multi_hot"`, `"count"`, or `"tf_idf"`. If `TRUE`, the output will have its feature axis padded to `max_tokens` even if the number of unique tokens in the vocabulary is less than `max_tokens`, resulting in a tensor of shape `(batch_size, max_tokens)` regardless of vocabulary size. Defaults to `FALSE`.
`name`	String, name for the object
`...`	For forward/backward compatability.

Value

The return value depends on the value provided for the first argument. If `object` is:

- a `keras_model_sequential()`, then the layer is added to the sequential model (which is modified in place). To enable piping, the sequential model is also returned, invisibly.
- a `keras_input()`, then the output tensor from calling `layer(input)` is returned.
- `NULL` or missing, then a `Layer` instance is returned.

Examples

Creating a lookup layer with a known vocabulary

This example creates a lookup layer with a pre-existing vocabulary.

```
vocab <- c(12, 36, 1138, 42) |> as.integer()
data <- op_array(rbind(c(12, 1138, 42),
                      c(42, 1000, 36)))  # Note OOV tokens
out <- data |> layer_integer_lookup(vocabulary = vocab)
out
```

```
## tf.Tensor(
## [[1 3 4]
##  [4 0 2]], shape=(2, 3), dtype=int64)
```

Creating a lookup layer with an adapted vocabulary

This example creates a lookup layer and generates the vocabulary by analyzing the dataset.

```
data <- op_array(rbind(c(12, 1138, 42),
                      c(42, 1000, 36)))  # Note OOV tokens
layer <- layer_integer_lookup()
layer |> adapt(data)
layer |> get_vocabulary() |> str()
```

```
## List of 6
##  $ : int -1
##  $ : num 42
##  $ : num 1138
##  $ : num 1000
##  $ : num 36
##  $ : num 12
```

Note that the OOV token -1 have been added to the vocabulary. The remaining tokens are sorted by frequency (42, which has 2 occurrences, is first) then by inverse sort order.

```
layer(data)
```

```
## tf.Tensor(
## [[5 2 1]
##  [1 3 4]], shape=(2, 3), dtype=int64)
```

Lookups with multiple OOV indices

This example demonstrates how to use a lookup layer with multiple OOV indices. When a layer is created with more than one OOV index, any OOV tokens are hashed into the number of OOV buckets, distributing OOV tokens in a deterministic fashion across the set.

```
vocab <- c(12, 36, 1138, 42) |> as.integer()
data <- op_array(rbind(c(12, 1138, 42),
                      c(37, 1000, 36)))  # Note OOV tokens
```

```
out <- data |>
  layer_integer_lookup(vocabulary = vocab,
                       num_oov_indices = 2)
out
```

```
## tf.Tensor(
## [[2 4 5]
##  [1 0 3]], shape=(2, 3), dtype=int64)
```

Note that the output for OOV token 37 is 1, while the output for OOV token 1000 is 0. The in-vocab terms have their output index increased by 1 from earlier examples (12 maps to 2, etc) in order to make space for the extra OOV token.

One-hot output

Configure the layer with `output_mode='one_hot'`. Note that the first `num_oov_indices` dimensions in the ont_hot encoding represent OOV values.

```
vocab <- c(12, 36, 1138, 42) |> as.integer()
data <- op_array(c(12, 36, 1138, 42, 7), 'int32')  # Note OOV tokens
layer <- layer_integer_lookup(vocabulary = vocab,
                              output_mode = 'one_hot')
layer(data)
```

```
## tf.Tensor(
## [[0 1 0 0 0]
##  [0 0 1 0 0]
##  [0 0 0 1 0]
##  [0 0 0 0 1]
##  [1 0 0 0 0]], shape=(5, 5), dtype=int64)
```

Multi-hot output

Configure the layer with `output_mode = 'multi_hot'`. Note that the first `num_oov_indices` dimensions in the multi_hot encoding represent OOV tokens

```
vocab <- c(12, 36, 1138, 42) |> as.integer()
data <- op_array(rbind(c(12, 1138, 42, 42),
                       c(42,    7, 36,  7)), "int64")  # Note OOV tokens
layer <- layer_integer_lookup(vocabulary = vocab,
                              output_mode = 'multi_hot')
layer(data)
```

```
## tf.Tensor(
## [[0 1 0 1 1]
##  [1 0 1 0 1]], shape=(2, 5), dtype=int64)
```

Token count output

Configure the layer with `output_mode='count'`. As with multi_hot output, the first `num_oov_indices` dimensions in the output represent OOV tokens.

```
vocab <- c(12, 36, 1138, 42) |> as.integer()
data <- rbind(c(12, 1138, 42, 42),
              c(42,    7, 36,  7)) |> op_array("int64")
layer <- layer_integer_lookup(vocabulary = vocab,
                              output_mode = 'count')
layer(data)
```

```
## tf.Tensor(
## [[0 1 0 1 2]
##  [2 0 1 0 1]], shape=(2, 5), dtype=int64)
```

TF-IDF output

Configure the layer with `output_mode='tf_idf'`. As with multi_hot output, the first `num_oov_indices` dimensions in the output represent OOV tokens.

Each token bin will output `token_count * idf_weight`, where the idf weights are the inverse document frequency weights per token. These should be provided along with the vocabulary. Note that the `idf_weight` for OOV tokens will default to the average of all idf weights passed in.

```
vocab <- c(12, 36, 1138, 42) |> as.integer()
idf_weights <- c(0.25, 0.75, 0.6, 0.4)
data <- rbind(c(12, 1138, 42, 42),
              c(42,    7, 36,  7)) |> op_array("int64")
layer <- layer_integer_lookup(output_mode = 'tf_idf',
                              vocabulary = vocab,
                              idf_weights = idf_weights)
layer(data)
```

```
## tf.Tensor(
## [[0.   0.25 0.   0.6  0.8 ]
##  [1.   0.   0.75 0.   0.4 ]], shape=(2, 5), dtype=float32)
```

To specify the idf weights for oov tokens, you will need to pass the entire vocabulary including the leading oov token.

```
vocab <- c(-1, 12, 36, 1138, 42) |> as.integer()
idf_weights <- c(0.9, 0.25, 0.75, 0.6, 0.4)
data <- rbind(c(12, 1138, 42, 42),
              c(42,    7, 36,  7)) |> op_array("int64")
layer <- layer_integer_lookup(output_mode = 'tf_idf',
                              vocabulary = vocab,
                              idf_weights = idf_weights)
layer(data)
```

```
## tf.Tensor(
## [[0.   0.25 0.   0.6  0.8 ]
##  [1.8  0.   0.75 0.   0.4 ]], shape=(2, 5), dtype=float32)
```

When adapting the layer in `"tf_idf"` mode, each input sample will be considered a document, and IDF weight per token will be calculated as: `log(1 + num_documents / (1 + token_document_count))`.

Inverse lookup

This example demonstrates how to map indices to tokens using this layer. (You can also use `adapt()` with `inverse = TRUE`, but for simplicity we'll pass the vocab in this example.)

```
vocab <- c(12, 36, 1138, 42) |> as.integer()
data <- op_array(c(1, 3, 4,
                  4, 0, 2)) |> op_reshape(c(2,-1)) |> op_cast("int32")
layer <- layer_integer_lookup(vocabulary = vocab, invert = TRUE)
layer(data)
```

```
## tf.Tensor(
## [[  12 1138   42]
##  [  42   -1   36]], shape=(2, 3), dtype=int64)
```

Note that the first index correspond to the oov token by default.

Forward and inverse lookup pairs

This example demonstrates how to use the vocabulary of a standard lookup layer to create an inverse lookup layer.

```
vocab <- c(12, 36, 1138, 42) |> as.integer()
data <- op_array(rbind(c(12, 1138, 42), c(42, 1000, 36)), "int32")
layer <- layer_integer_lookup(vocabulary = vocab)
i_layer <- layer_integer_lookup(vocabulary = get_vocabulary(layer),
                                invert = TRUE)
int_data <- layer(data)
i_layer(int_data)
```

```
## tf.Tensor(
## [[  12 1138   42]
##  [  42   -1   36]], shape=(2, 3), dtype=int64)
```

In this example, the input token 1000 resulted in an output of -1, since 1000 was not in the vocabulary - it got represented as an OOV, and all OOV tokens are returned as -1 in the inverse layer. Also, note that for the inverse to work, you must have already set the forward layer vocabulary either directly or via `adapt()` before calling `get_vocabulary()`.

See Also

- https://keras.io/api/layers/preprocessing_layers/categorical/integer_lookup#integerlookup-class

Other categorical features preprocessing layers:
layer_category_encoding()
layer_hashed_crossing()
layer_hashing()
layer_string_lookup()

Other preprocessing layers:
layer_category_encoding()
layer_center_crop()
layer_discretization()
layer_feature_space()
layer_hashed_crossing()
layer_hashing()
layer_mel_spectrogram()
layer_normalization()
layer_random_brightness()
layer_random_contrast()
layer_random_crop()
layer_random_flip()
layer_random_rotation()
layer_random_translation()
layer_random_zoom()
layer_rescaling()
layer_resizing()
layer_string_lookup()
layer_text_vectorization()

Other layers:
Layer()
layer_activation()
layer_activation_elu()
layer_activation_leaky_relu()
layer_activation_parametric_relu()
layer_activation_relu()
layer_activation_softmax()
layer_activity_regularization()
layer_add()
layer_additive_attention()
layer_alpha_dropout()
layer_attention()
layer_average()
layer_average_pooling_1d()
layer_average_pooling_2d()
layer_average_pooling_3d()

layer_batch_normalization()
layer_bidirectional()
layer_category_encoding()
layer_center_crop()
layer_concatenate()
layer_conv_1d()
layer_conv_1d_transpose()
layer_conv_2d()
layer_conv_2d_transpose()
layer_conv_3d()
layer_conv_3d_transpose()
layer_conv_lstm_1d()
layer_conv_lstm_2d()
layer_conv_lstm_3d()
layer_cropping_1d()
layer_cropping_2d()
layer_cropping_3d()
layer_dense()
layer_depthwise_conv_1d()
layer_depthwise_conv_2d()
layer_discretization()
layer_dot()
layer_dropout()
layer_einsum_dense()
layer_embedding()
layer_feature_space()
layer_flatten()
layer_flax_module_wrapper()
layer_gaussian_dropout()
layer_gaussian_noise()
layer_global_average_pooling_1d()
layer_global_average_pooling_2d()
layer_global_average_pooling_3d()
layer_global_max_pooling_1d()
layer_global_max_pooling_2d()
layer_global_max_pooling_3d()
layer_group_normalization()
layer_group_query_attention()
layer_gru()
layer_hashed_crossing()
layer_hashing()
layer_identity()
layer_jax_model_wrapper()
layer_lambda()
layer_layer_normalization()
layer_lstm()
layer_masking()
layer_max_pooling_1d()

layer_max_pooling_2d()
layer_max_pooling_3d()
layer_maximum()
layer_mel_spectrogram()
layer_minimum()
layer_multi_head_attention()
layer_multiply()
layer_normalization()
layer_permute()
layer_random_brightness()
layer_random_contrast()
layer_random_crop()
layer_random_flip()
layer_random_rotation()
layer_random_translation()
layer_random_zoom()
layer_repeat_vector()
layer_rescaling()
layer_reshape()
layer_resizing()
layer_rnn()
layer_separable_conv_1d()
layer_separable_conv_2d()
layer_simple_rnn()
layer_spatial_dropout_1d()
layer_spatial_dropout_2d()
layer_spatial_dropout_3d()
layer_spectral_normalization()
layer_string_lookup()
layer_subtract()
layer_text_vectorization()
layer_tfsm()
layer_time_distributed()
layer_torch_module_wrapper()
layer_unit_normalization()
layer_upsampling_1d()
layer_upsampling_2d()
layer_upsampling_3d()
layer_zero_padding_1d()
layer_zero_padding_2d()
layer_zero_padding_3d()
rnn_cell_gru()
rnn_cell_lstm()
rnn_cell_simple()
rnn_cells_stack()

layer_jax_model_wrapper

Keras Layer that wraps a JAX model.

Description

This layer enables the use of JAX components within Keras when using JAX as the backend for Keras.

Usage

```
layer_jax_model_wrapper(
  object,
  call_fn,
  init_fn = NULL,
  params = NULL,
  state = NULL,
  seed = NULL,
  ...
)
```

Arguments

`object`	Object to compose the layer with. A tensor, array, or sequential model.
`call_fn`	The function to call the model. See description above for the list of arguments it takes and the outputs it returns.
`init_fn`	the function to call to initialize the model. See description above for the list of arguments it takes and the ouputs it returns. If `NULL`, then `params` and/or `state` must be provided.
`params`	A `PyTree` containing all the model trainable parameters. This allows passing trained parameters or controlling the initialization. If both `params` and `state` are `NULL`, `init_fn()` is called at build time to initialize the trainable parameters of the model.
`state`	A `PyTree` containing all the model non-trainable state. This allows passing learned state or controlling the initialization. If both `params` and `state` are `NULL`, and `call_fn()` takes a `state` argument, then `init_fn()` is called at build time to initialize the non-trainable state of the model.
`seed`	Seed for random number generator. Optional.
`...`	For forward/backward compatability.

Value

The return value depends on the value provided for the first argument. If `object` is:

- a `keras_model_sequential()`, then the layer is added to the sequential model (which is modified in place). To enable piping, the sequential model is also returned, invisibly.

- a `keras_input()`, then the output tensor from calling `layer(input)` is returned.
- `NULL` or missing, then a `Layer` instance is returned.

Model function

This layer accepts JAX models in the form of a function, `call_fn()`, which must take the following arguments with these exact names:

- `params`: trainable parameters of the model.
- `state` (*optional*): non-trainable state of the model. Can be omitted if the model has no non-trainable state.
- `rng` (*optional*): a `jax.random.PRNGKey` instance. Can be omitted if the model does not need RNGs, neither during training nor during inference.
- `inputs`: inputs to the model, a JAX array or a `PyTree` of arrays.
- `training` (*optional*): an argument specifying if we're in training mode or inference mode, `TRUE` is passed in training mode. Can be omitted if the model behaves the same in training mode and inference mode.

The `inputs` argument is mandatory. Inputs to the model must be provided via a single argument. If the JAX model takes multiple inputs as separate arguments, they must be combined into a single structure, for instance in a `tuple()` or a `dict()`.

Model weights initialization:

The initialization of the `params` and `state` of the model can be handled by this layer, in which case the `init_fn()` argument must be provided. This allows the model to be initialized dynamically with the right shape. Alternatively, and if the shape is known, the `params` argument and optionally the `state` argument can be used to create an already initialized model.

The `init_fn()` function, if provided, must take the following arguments with these exact names:

- `rng`: a `jax.random.PRNGKey` instance.
- `inputs`: a JAX array or a `PyTree` of arrays with placeholder values to provide the shape of the inputs.
- `training` (*optional*): an argument specifying if we're in training mode or inference mode. `True` is always passed to `init_fn`. Can be omitted regardless of whether `call_fn` has a `training` argument.

Models with non-trainable state:

For JAX models that have non-trainable state:

- `call_fn()` must have a `state` argument
- `call_fn()` must return a `tuple()` containing the outputs of the model and the new non-trainable state of the model
- `init_fn()` must return a `tuple()` containing the initial trainable params of the model and the initial non-trainable state of the model.

This code shows a possible combination of `call_fn()` and `init_fn()` signatures for a model with non-trainable state. In this example, the model has a `training` argument and an `rng` argument in `call_fn()`.

```
stateful_call <- function(params, state, rng, inputs, training) {
  outputs <- ....
  new_state <- ....
  tuple(outputs, new_state)
}

stateful_init <- function(rng, inputs) {
  initial_params <- ....
  initial_state <- ....
  tuple(initial_params, initial_state)
}
```

Models without non-trainable state:

For JAX models with no non-trainable state:

- `call_fn()` must not have a `state` argument
- `call_fn()` must return only the outputs of the model
- `init_fn()` must return only the initial trainable params of the model.

This code shows a possible combination of `call_fn()` and `init_fn()` signatures for a model without non-trainable state. In this example, the model does not have a `training` argument and does not have an `rng` argument in `call_fn()`.

```
stateful_call <- function(pparams, inputs) {
  outputs <- ....
  outputs
}

stateful_init <- function(rng, inputs) {
  initial_params <- ....
  initial_params
}
```

Conforming to the required signature:

If a model has a different signature than the one required by `JaxLayer`, one can easily write a wrapper method to adapt the arguments. This example shows a model that has multiple inputs as separate arguments, expects multiple RNGs in a `dict`, and has a `deterministic` argument with the opposite meaning of `training`. To conform, the inputs are combined in a single structure using a `tuple`, the RNG is split and used the populate the expected `dict`, and the Boolean flag is negated:

```
jax <- import("jax")
my_model_fn <- function(params, rngs, input1, input2, deterministic) {
  ....
  if (!deterministic) {
    dropout_rng <- rngs$dropout
    keep <- jax$random$bernoulli(dropout_rng, dropout_rate, x$shape)
    x <- jax$numpy$where(keep, x / dropout_rate, 0)
    ....
  }
  ....
  return(outputs)
}

my_model_wrapper_fn <- function(params, rng, inputs, training) {
  c(input1, input2) %<-% inputs
  c(rng1, rng2) %<-% jax$random$split(rng)
  rngs <-  list(dropout = rng1, preprocessing = rng2)
  deterministic <-  !training
  my_model_fn(params, rngs, input1, input2, deterministic)
}

keras_layer <- layer_jax_model_wrapper(call_fn = my_model_wrapper_fn,
                                       params = initial_params)
```

Usage with Haiku modules:

`JaxLayer` enables the use of Haiku components in the form of `haiku.Module`. This is achieved by transforming the module per the Haiku pattern and then passing `module.apply` in the `call_fn` parameter and `module.init` in the `init_fn` parameter if needed.

If the model has non-trainable state, it should be transformed with `haiku.transform_with_state`. If the model has no non-trainable state, it should be transformed with `haiku.transform`. Additionally, and optionally, if the module does not use RNGs in "apply", it can be transformed with `haiku.without_apply_rng`.

The following example shows how to create a `JaxLayer` from a Haiku module that uses random number generators via `hk.next_rng_key()` and takes a training positional argument:

```
# reticulate::py_install("haiku", "r-keras")
hk <- import("haiku")
MyHaikuModule(hk$Module) \%py_class\% {

  `__call__` <- \(self, x, training) {
    x <- hk$Conv2D(32L, tuple(3L, 3L))(x)
    x <- jax$nn$relu(x)
    x <- hk$AvgPool(tuple(1L, 2L, 2L, 1L),
                    tuple(1L, 2L, 2L, 1L), "VALID")(x)
    x <- hk$Flatten()(x)
    x <- hk$Linear(200L)(x)
    if (training)
      x <- hk$dropout(rng = hk$next_rng_key(), rate = 0.3, x = x)
    x <- jax$nn$relu(x)
    x <- hk$Linear(10L)(x)
    x <- jax$nn$softmax(x)
    x
  }

}

my_haiku_module_fn <- function(inputs, training) {
  module <- MyHaikuModule()
  module(inputs, training)
}

transformed_module <- hk$transform(my_haiku_module_fn)

keras_layer <-
  layer_jax_model_wrapper(call_fn = transformed_module$apply,
                          init_fn = transformed_module$init)
```

See Also

Other wrapping layers:
`layer_flax_module_wrapper()`
`layer_torch_module_wrapper()`

Other layers:
`Layer()`
`layer_activation()`
`layer_activation_elu()`
`layer_activation_leaky_relu()`
`layer_activation_parametric_relu()`
`layer_activation_relu()`
`layer_activation_softmax()`
`layer_activity_regularization()`
`layer_add()`
`layer_additive_attention()`
`layer_alpha_dropout()`
`layer_attention()`
`layer_average()`
`layer_average_pooling_1d()`
`layer_average_pooling_2d()`
`layer_average_pooling_3d()`
`layer_batch_normalization()`
`layer_bidirectional()`
`layer_category_encoding()`
`layer_center_crop()`
`layer_concatenate()`
`layer_conv_1d()`
`layer_conv_1d_transpose()`
`layer_conv_2d()`
`layer_conv_2d_transpose()`
`layer_conv_3d()`
`layer_conv_3d_transpose()`
`layer_conv_lstm_1d()`
`layer_conv_lstm_2d()`
`layer_conv_lstm_3d()`
`layer_cropping_1d()`
`layer_cropping_2d()`

layer_cropping_3d()
layer_dense()
layer_depthwise_conv_1d()
layer_depthwise_conv_2d()
layer_discretization()
layer_dot()
layer_dropout()
layer_einsum_dense()
layer_embedding()
layer_feature_space()
layer_flatten()
layer_flax_module_wrapper()
layer_gaussian_dropout()
layer_gaussian_noise()
layer_global_average_pooling_1d()
layer_global_average_pooling_2d()
layer_global_average_pooling_3d()
layer_global_max_pooling_1d()
layer_global_max_pooling_2d()
layer_global_max_pooling_3d()
layer_group_normalization()
layer_group_query_attention()
layer_gru()
layer_hashed_crossing()
layer_hashing()
layer_identity()
layer_integer_lookup()
layer_lambda()
layer_layer_normalization()
layer_lstm()
layer_masking()
layer_max_pooling_1d()
layer_max_pooling_2d()
layer_max_pooling_3d()
layer_maximum()
layer_mel_spectrogram()
layer_minimum()
layer_multi_head_attention()
layer_multiply()
layer_normalization()
layer_permute()
layer_random_brightness()
layer_random_contrast()
layer_random_crop()
layer_random_flip()
layer_random_rotation()
layer_random_translation()
layer_random_zoom()

layer_repeat_vector()
layer_rescaling()
layer_reshape()
layer_resizing()
layer_rnn()
layer_separable_conv_1d()
layer_separable_conv_2d()
layer_simple_rnn()
layer_spatial_dropout_1d()
layer_spatial_dropout_2d()
layer_spatial_dropout_3d()
layer_spectral_normalization()
layer_string_lookup()
layer_subtract()
layer_text_vectorization()
layer_tfsm()
layer_time_distributed()
layer_torch_module_wrapper()
layer_unit_normalization()
layer_upsampling_1d()
layer_upsampling_2d()
layer_upsampling_3d()
layer_zero_padding_1d()
layer_zero_padding_2d()
layer_zero_padding_3d()
rnn_cell_gru()
rnn_cell_lstm()
rnn_cell_simple()
rnn_cells_stack()

---

layer_lambda *Wraps arbitrary expressions as a* Layer *object.*

Description

The layer_lambda() layer exists so that arbitrary expressions can be used as a Layer when constructing Sequential and Functional API models. Lambda layers are best suited for simple operations or quick experimentation. For more advanced use cases, prefer writing new subclasses of Layer using new_layer_class().

Usage

```
layer_lambda(
  object,
  f,
  output_shape = NULL,
  mask = NULL,
  arguments = NULL,
  ...
)
```

Arguments

`object`	Object to compose the layer with. A tensor, array, or sequential model.
`f`	The function to be evaluated. Takes input tensor as first argument.
`output_shape`	Expected output shape from function. This argument can usually be inferred if not explicitly provided. Can be a list or function. If a list, it only specifies the first dimension onward; sample dimension is assumed either the same as the input: `output_shape = c(input_shape[1], output_shape)` or, the input is `NULL` and the sample dimension is also `NULL`: `output_shape = c(NA, output_shape)`. If a function, it specifies the entire shape as a function of the input shape: `output_shape = f(input_shape)`.
`mask`	Either `NULL` (indicating no masking) or a callable with the same signature as the `compute_mask` layer method, or a tensor that will be returned as output mask regardless of what the input is.
`arguments`	Optional named list of arguments to be passed to the function.
`...`	For forward/backward compatability.

Value

The return value depends on the value provided for the first argument. If `object` is:

- a `keras_model_sequential()`, then the layer is added to the sequential model (which is modified in place). To enable piping, the sequential model is also returned, invisibly.
- a `keras_input()`, then the output tensor from calling `layer(input)` is returned.
- `NULL` or missing, then a `Layer` instance is returned.

Examples

```
# add a x -> x^2 layer
model <- keras_model_sequential()
model |> layer_lambda(\(x) x^2)
```

See Also

- https://keras.io/api/layers/core_layers/lambda#lambda-class

Other core layers:
`layer_dense()`
`layer_einsum_dense()`
`layer_embedding()`
`layer_identity()`
`layer_masking()`

Other layers:
Layer()
layer_activation()
layer_activation_elu()
layer_activation_leaky_relu()
layer_activation_parametric_relu()
layer_activation_relu()
layer_activation_softmax()
layer_activity_regularization()
layer_add()
layer_additive_attention()
layer_alpha_dropout()
layer_attention()
layer_average()
layer_average_pooling_1d()
layer_average_pooling_2d()
layer_average_pooling_3d()
layer_batch_normalization()
layer_bidirectional()
layer_category_encoding()
layer_center_crop()
layer_concatenate()
layer_conv_1d()
layer_conv_1d_transpose()
layer_conv_2d()
layer_conv_2d_transpose()
layer_conv_3d()
layer_conv_3d_transpose()
layer_conv_lstm_1d()
layer_conv_lstm_2d()
layer_conv_lstm_3d()
layer_cropping_1d()
layer_cropping_2d()
layer_cropping_3d()
layer_dense()
layer_depthwise_conv_1d()
layer_depthwise_conv_2d()
layer_discretization()
layer_dot()
layer_dropout()
layer_einsum_dense()
layer_embedding()
layer_feature_space()
layer_flatten()
layer_flax_module_wrapper()
layer_gaussian_dropout()
layer_gaussian_noise()
layer_global_average_pooling_1d()

layer_global_average_pooling_2d()
layer_global_average_pooling_3d()
layer_global_max_pooling_1d()
layer_global_max_pooling_2d()
layer_global_max_pooling_3d()
layer_group_normalization()
layer_group_query_attention()
layer_gru()
layer_hashed_crossing()
layer_hashing()
layer_identity()
layer_integer_lookup()
layer_jax_model_wrapper()
layer_layer_normalization()
layer_lstm()
layer_masking()
layer_max_pooling_1d()
layer_max_pooling_2d()
layer_max_pooling_3d()
layer_maximum()
layer_mel_spectrogram()
layer_minimum()
layer_multi_head_attention()
layer_multiply()
layer_normalization()
layer_permute()
layer_random_brightness()
layer_random_contrast()
layer_random_crop()
layer_random_flip()
layer_random_rotation()
layer_random_translation()
layer_random_zoom()
layer_repeat_vector()
layer_rescaling()
layer_reshape()
layer_resizing()
layer_rnn()
layer_separable_conv_1d()
layer_separable_conv_2d()
layer_simple_rnn()
layer_spatial_dropout_1d()
layer_spatial_dropout_2d()
layer_spatial_dropout_3d()
layer_spectral_normalization()
layer_string_lookup()
layer_subtract()
layer_text_vectorization()

layer_tfsm()
layer_time_distributed()
layer_torch_module_wrapper()
layer_unit_normalization()
layer_upsampling_1d()
layer_upsampling_2d()
layer_upsampling_3d()
layer_zero_padding_1d()
layer_zero_padding_2d()
layer_zero_padding_3d()
rnn_cell_gru()
rnn_cell_lstm()
rnn_cell_simple()
rnn_cells_stack()

---

layer_layer_normalization
Layer normalization layer (Ba et al., 2016).

Description

Normalize the activations of the previous layer for each given example in a batch independently, rather than across a batch like Batch Normalization. i.e. applies a transformation that maintains the mean activation within each example close to 0 and the activation standard deviation close to 1.

If scale or center are enabled, the layer will scale the normalized outputs by broadcasting them with a trainable variable gamma, and center the outputs by broadcasting with a trainable variable beta. gamma will default to a ones tensor and beta will default to a zeros tensor, so that centering and scaling are no-ops before training has begun.

So, with scaling and centering enabled the normalization equations are as follows:

Let the intermediate activations for a mini-batch to be the inputs.

For each sample x in a batch of inputs, we compute the mean and variance of the sample, normalize each value in the sample (including a small factor epsilon for numerical stability), and finally, transform the normalized output by gamma and beta, which are learned parameters:

```
outputs <- inputs |> apply(1, function(x) {
  x_normalized <- (x - mean(x)) /
                  sqrt(var(x) + epsilon)
  x_normalized * gamma + beta
})
```

gamma and beta will span the axes of inputs specified in axis, and this part of the inputs' shape must be fully defined.

For example:

```
layer <- layer_layer_normalization(axis = c(2, 3, 4))

layer(op_ones(c(5, 20, 30, 40))) |> invisible() # build()
shape(layer$beta)
```

```
## shape(20, 30, 40)
```

```
shape(layer$gamma)
```

```
## shape(20, 30, 40)
```

Note that other implementations of layer normalization may choose to define `gamma` and `beta` over a separate set of axes from the axes being normalized across. For example, Group Normalization (Wu et al. 2018) with group size of 1 corresponds to a `layer_layer_normalization()` that normalizes across height, width, and channel and has `gamma` and `beta` span only the channel dimension. So, this `layer_layer_normalization()` implementation will not match a `layer_group_normalization()` layer with group size set to 1.

Usage

```
layer_layer_normalization(
  object,
  axis = -1L,
  epsilon = 0.001,
  center = TRUE,
  scale = TRUE,
  rms_scaling = FALSE,
  beta_initializer = "zeros",
  gamma_initializer = "ones",
  beta_regularizer = NULL,
  gamma_regularizer = NULL,
  beta_constraint = NULL,
  gamma_constraint = NULL,
  ...
)
```

Arguments

`object`	Object to compose the layer with. A tensor, array, or sequential model.
`axis`	Integer or list. The axis or axes to normalize across. Typically, this is the features axis/axes. The left-out axes are typically the batch axis/axes. `-1` is the last dimension in the input. Defaults to `-1`.
`epsilon`	Small float added to variance to avoid dividing by zero. Defaults to 1e-3.
`center`	If `TRUE`, add offset of `beta` to normalized tensor. If `FALSE`, `beta` is ignored. Defaults to `TRUE`.

`scale`	If `TRUE`, multiply by `gamma`. If `FALSE`, `gamma` is not used. When the next layer is linear (also e.g. `layer_activation_relu()`), this can be disabled since the scaling will be done by the next layer. Defaults to `TRUE`.
`rms_scaling`	If `TRUE`, `center` and `scale` are ignored, and the inputs are scaled by `gamma` and the inverse square root of the square of all inputs. This is an approximate and faster approach that avoids ever computing the mean of the input.
`beta_initializer`	Initializer for the beta weight. Defaults to zeros.
`gamma_initializer`	Initializer for the gamma weight. Defaults to ones.
`beta_regularizer`	Optional regularizer for the beta weight. `NULL` by default.
`gamma_regularizer`	Optional regularizer for the gamma weight. `NULL` by default.
`beta_constraint`	Optional constraint for the beta weight. `NULL` by default.
`gamma_constraint`	Optional constraint for the gamma weight. `NULL` by default.
`...`	Base layer keyword arguments (e.g. `name` and `dtype`).

Value

The return value depends on the value provided for the first argument. If `object` is:

- a `keras_model_sequential()`, then the layer is added to the sequential model (which is modified in place). To enable piping, the sequential model is also returned, invisibly.
- a `keras_input()`, then the output tensor from calling `layer(input)` is returned.
- `NULL` or missing, then a `Layer` instance is returned.

Reference

- Lei Ba et al., 2016.

See Also

- `https://keras.io/api/layers/normalization_layers/layer_normalization#layernormalization-class`

Other normalization layers:
`layer_batch_normalization()`
`layer_group_normalization()`
`layer_spectral_normalization()`
`layer_unit_normalization()`

Other layers:
`Layer()`
`layer_activation()`
`layer_activation_elu()`

layer_activation_leaky_relu()
layer_activation_parametric_relu()
layer_activation_relu()
layer_activation_softmax()
layer_activity_regularization()
layer_add()
layer_additive_attention()
layer_alpha_dropout()
layer_attention()
layer_average()
layer_average_pooling_1d()
layer_average_pooling_2d()
layer_average_pooling_3d()
layer_batch_normalization()
layer_bidirectional()
layer_category_encoding()
layer_center_crop()
layer_concatenate()
layer_conv_1d()
layer_conv_1d_transpose()
layer_conv_2d()
layer_conv_2d_transpose()
layer_conv_3d()
layer_conv_3d_transpose()
layer_conv_lstm_1d()
layer_conv_lstm_2d()
layer_conv_lstm_3d()
layer_cropping_1d()
layer_cropping_2d()
layer_cropping_3d()
layer_dense()
layer_depthwise_conv_1d()
layer_depthwise_conv_2d()
layer_discretization()
layer_dot()
layer_dropout()
layer_einsum_dense()
layer_embedding()
layer_feature_space()
layer_flatten()
layer_flax_module_wrapper()
layer_gaussian_dropout()
layer_gaussian_noise()
layer_global_average_pooling_1d()
layer_global_average_pooling_2d()
layer_global_average_pooling_3d()
layer_global_max_pooling_1d()
layer_global_max_pooling_2d()

layer_global_max_pooling_3d()
layer_group_normalization()
layer_group_query_attention()
layer_gru()
layer_hashed_crossing()
layer_hashing()
layer_identity()
layer_integer_lookup()
layer_jax_model_wrapper()
layer_lambda()
layer_lstm()
layer_masking()
layer_max_pooling_1d()
layer_max_pooling_2d()
layer_max_pooling_3d()
layer_maximum()
layer_mel_spectrogram()
layer_minimum()
layer_multi_head_attention()
layer_multiply()
layer_normalization()
layer_permute()
layer_random_brightness()
layer_random_contrast()
layer_random_crop()
layer_random_flip()
layer_random_rotation()
layer_random_translation()
layer_random_zoom()
layer_repeat_vector()
layer_rescaling()
layer_reshape()
layer_resizing()
layer_rnn()
layer_separable_conv_1d()
layer_separable_conv_2d()
layer_simple_rnn()
layer_spatial_dropout_1d()
layer_spatial_dropout_2d()
layer_spatial_dropout_3d()
layer_spectral_normalization()
layer_string_lookup()
layer_subtract()
layer_text_vectorization()
layer_tfsm()
layer_time_distributed()
layer_torch_module_wrapper()
layer_unit_normalization()

layer_upsampling_1d()
layer_upsampling_2d()
layer_upsampling_3d()
layer_zero_padding_1d()
layer_zero_padding_2d()
layer_zero_padding_3d()
rnn_cell_gru()
rnn_cell_lstm()
rnn_cell_simple()
rnn_cells_stack()

layer_lstm *Long Short-Term Memory layer - Hochreiter 1997.*

Description

Based on available runtime hardware and constraints, this layer will choose different implementations (cuDNN-based or backend-native) to maximize the performance. If a GPU is available and all the arguments to the layer meet the requirement of the cuDNN kernel (see below for details), the layer will use a fast cuDNN implementation when using the TensorFlow backend. The requirements to use the cuDNN implementation are:

1. activation == tanh
2. recurrent_activation == sigmoid
3. dropout == 0 and recurrent_dropout == 0
4. unroll is FALSE
5. use_bias is TRUE
6. Inputs, if use masking, are strictly right-padded.
7. Eager execution is enabled in the outermost context.

For example:

```
input <- random_uniform(c(32, 10, 8))
output <- input |> layer_lstm(4)
shape(output)

## shape(32, 4)


lstm <- layer_lstm(units = 4, return_sequences = TRUE, return_state = TRUE)
c(whole_seq_output, final_memory_state, final_carry_state) %<-% lstm(input)
shape(whole_seq_output)

## shape(32, 10, 4)
```

```
shape(final_memory_state)

## shape(32, 4)


shape(final_carry_state)

## shape(32, 4)
```

Usage

```
layer_lstm(
  object,
  units,
  activation = "tanh",
  recurrent_activation = "sigmoid",
  use_bias = TRUE,
  kernel_initializer = "glorot_uniform",
  recurrent_initializer = "orthogonal",
  bias_initializer = "zeros",
  unit_forget_bias = TRUE,
  kernel_regularizer = NULL,
  recurrent_regularizer = NULL,
  bias_regularizer = NULL,
  activity_regularizer = NULL,
  kernel_constraint = NULL,
  recurrent_constraint = NULL,
  bias_constraint = NULL,
  dropout = 0,
  recurrent_dropout = 0,
  seed = NULL,
  return_sequences = FALSE,
  return_state = FALSE,
  go_backwards = FALSE,
  stateful = FALSE,
  unroll = FALSE,
  use_cudnn = "auto",
  ...
)
```

Arguments

`object`	Object to compose the layer with. A tensor, array, or sequential model.
`units`	Positive integer, dimensionality of the output space.
`activation`	Activation function to use. Default: hyperbolic tangent (`tanh`). If you pass `NULL`, no activation is applied (ie. "linear" activation: `a(x) = x`).

`recurrent_activation`	Activation function to use for the recurrent step. Default: sigmoid (`sigmoid`). If you pass `NULL`, no activation is applied (ie. "linear" activation: `a(x) = x`).
`use_bias`	Boolean, (default `TRUE`), whether the layer should use a bias vector.
`kernel_initializer`	Initializer for the `kernel` weights matrix, used for the linear transformation of the inputs. Default: `"glorot_uniform"`.
`recurrent_initializer`	Initializer for the `recurrent_kernel` weights matrix, used for the linear transformation of the recurrent state. Default: `"orthogonal"`.
`bias_initializer`	Initializer for the bias vector. Default: `"zeros"`.
`unit_forget_bias`	Boolean (default `TRUE`). If `TRUE`, add 1 to the bias of the forget gate at initialization. Setting it to `TRUE` will also force `bias_initializer="zeros"`. This is recommended in Jozefowicz et al.
`kernel_regularizer`	Regularizer function applied to the `kernel` weights matrix. Default: `NULL`.
`recurrent_regularizer`	Regularizer function applied to the `recurrent_kernel` weights matrix. Default: `NULL`.
`bias_regularizer`	Regularizer function applied to the bias vector. Default: `NULL`.
`activity_regularizer`	Regularizer function applied to the output of the layer (its "activation"). Default: `NULL`.
`kernel_constraint`	Constraint function applied to the `kernel` weights matrix. Default: `NULL`.
`recurrent_constraint`	Constraint function applied to the `recurrent_kernel` weights matrix. Default: `NULL`.
`bias_constraint`	Constraint function applied to the bias vector. Default: `NULL`.
`dropout`	Float between 0 and 1. Fraction of the units to drop for the linear transformation of the inputs. Default: 0.
`recurrent_dropout`	Float between 0 and 1. Fraction of the units to drop for the linear transformation of the recurrent state. Default: 0.
`seed`	Random seed for dropout.
`return_sequences`	Boolean. Whether to return the last output in the output sequence, or the full sequence. Default: `FALSE`.
`return_state`	Boolean. Whether to return the last state in addition to the output. Default: `FALSE`.
`go_backwards`	Boolean (default: `FALSE`). If `TRUE`, process the input sequence backwards and return the reversed sequence.

stateful	Boolean (default: FALSE). If TRUE, the last state for each sample at index i in a batch will be used as initial state for the sample of index i in the following batch.
unroll	Boolean (default FALSE). If TRUE, the network will be unrolled, else a symbolic loop will be used. Unrolling can speed-up a RNN, although it tends to be more memory-intensive. Unrolling is only suitable for short sequences.
use_cudnn	Whether to use a cuDNN-backed implementation. "auto" will attempt to use cuDNN when feasible, and will fallback to the default implementation if not.
...	For forward/backward compatability.

Value

The return value depends on the value provided for the first argument. If object is:

- a keras_model_sequential(), then the layer is added to the sequential model (which is modified in place). To enable piping, the sequential model is also returned, invisibly.
- a keras_input(), then the output tensor from calling layer(input) is returned.
- NULL or missing, then a Layer instance is returned.

Call Arguments

- inputs: A 3D tensor, with shape (batch, timesteps, feature).
- mask: Binary tensor of shape (samples, timesteps) indicating whether a given timestep should be masked (optional). An individual TRUE entry indicates that the corresponding timestep should be utilized, while a FALSE entry indicates that the corresponding timestep should be ignored. Defaults to NULL.
- training: Boolean indicating whether the layer should behave in training mode or in inference mode. This argument is passed to the cell when calling it. This is only relevant if dropout or recurrent_dropout is used (optional). Defaults to NULL.
- initial_state: List of initial state tensors to be passed to the first call of the cell (optional, NULL causes creation of zero-filled initial state tensors). Defaults to NULL.

See Also

- https://keras.io/api/layers/recurrent_layers/lstm#lstm-class

Other lstm rnn layers:
rnn_cell_lstm()

Other rnn layers:
layer_bidirectional()
layer_conv_lstm_1d()
layer_conv_lstm_2d()
layer_conv_lstm_3d()
layer_gru()
layer_rnn()
layer_simple_rnn()
layer_time_distributed()

rnn_cell_gru()
rnn_cell_lstm()
rnn_cell_simple()
rnn_cells_stack()

Other layers:
Layer()
layer_activation()
layer_activation_elu()
layer_activation_leaky_relu()
layer_activation_parametric_relu()
layer_activation_relu()
layer_activation_softmax()
layer_activity_regularization()
layer_add()
layer_additive_attention()
layer_alpha_dropout()
layer_attention()
layer_average()
layer_average_pooling_1d()
layer_average_pooling_2d()
layer_average_pooling_3d()
layer_batch_normalization()
layer_bidirectional()
layer_category_encoding()
layer_center_crop()
layer_concatenate()
layer_conv_1d()
layer_conv_1d_transpose()
layer_conv_2d()
layer_conv_2d_transpose()
layer_conv_3d()
layer_conv_3d_transpose()
layer_conv_lstm_1d()
layer_conv_lstm_2d()
layer_conv_lstm_3d()
layer_cropping_1d()
layer_cropping_2d()
layer_cropping_3d()
layer_dense()
layer_depthwise_conv_1d()
layer_depthwise_conv_2d()
layer_discretization()
layer_dot()
layer_dropout()
layer_einsum_dense()
layer_embedding()
layer_feature_space()

layer_flatten()
layer_flax_module_wrapper()
layer_gaussian_dropout()
layer_gaussian_noise()
layer_global_average_pooling_1d()
layer_global_average_pooling_2d()
layer_global_average_pooling_3d()
layer_global_max_pooling_1d()
layer_global_max_pooling_2d()
layer_global_max_pooling_3d()
layer_group_normalization()
layer_group_query_attention()
layer_gru()
layer_hashed_crossing()
layer_hashing()
layer_identity()
layer_integer_lookup()
layer_jax_model_wrapper()
layer_lambda()
layer_layer_normalization()
layer_masking()
layer_max_pooling_1d()
layer_max_pooling_2d()
layer_max_pooling_3d()
layer_maximum()
layer_mel_spectrogram()
layer_minimum()
layer_multi_head_attention()
layer_multiply()
layer_normalization()
layer_permute()
layer_random_brightness()
layer_random_contrast()
layer_random_crop()
layer_random_flip()
layer_random_rotation()
layer_random_translation()
layer_random_zoom()
layer_repeat_vector()
layer_rescaling()
layer_reshape()
layer_resizing()
layer_rnn()
layer_separable_conv_1d()
layer_separable_conv_2d()
layer_simple_rnn()
layer_spatial_dropout_1d()
layer_spatial_dropout_2d()

layer_spatial_dropout_3d()
layer_spectral_normalization()
layer_string_lookup()
layer_subtract()
layer_text_vectorization()
layer_tfsm()
layer_time_distributed()
layer_torch_module_wrapper()
layer_unit_normalization()
layer_upsampling_1d()
layer_upsampling_2d()
layer_upsampling_3d()
layer_zero_padding_1d()
layer_zero_padding_2d()
layer_zero_padding_3d()
rnn_cell_gru()
rnn_cell_lstm()
rnn_cell_simple()
rnn_cells_stack()

---

layer_masking *Masks a sequence by using a mask value to skip timesteps.*

Description

For each timestep in the input tensor (dimension #1 in the tensor), if all values in the input tensor at that timestep are equal to `mask_value`, then the timestep will be masked (skipped) in all downstream layers (as long as they support masking).

If any downstream layer does not support masking yet receives such an input mask, an exception will be raised.

Usage

```
layer_masking(object, mask_value = 0, ...)
```

Arguments

`object`	Object to compose the layer with. A tensor, array, or sequential model.
`mask_value`	see description
`...`	For forward/backward compatability.

Value

The return value depends on the value provided for the first argument. If `object` is:

- a `keras_model_sequential()`, then the layer is added to the sequential model (which is modified in place). To enable piping, the sequential model is also returned, invisibly.
- a `keras_input()`, then the output tensor from calling `layer(input)` is returned.
- `NULL` or missing, then a `Layer` instance is returned.

Examples

Consider an array `x` of shape `c(samples, timesteps, features)`, to be fed to an LSTM layer. You want to mask timestep #3 and #5 because you lack data for these timesteps. You can:

- Set `x[, 3, ] <- 0.` and `x[, 5, ] <- 0.`
- Insert a `layer_masking()` layer with `mask_value = 0.` before the LSTM layer:

```
c(samples, timesteps, features) %<-% c(32, 10, 8)
inputs <- c(samples, timesteps, features) %>% { array(runif(prod(.)), dim = .) }
inputs[, 3, ] <- 0
inputs[, 5, ] <- 0

model <- keras_model_sequential() %>%
  layer_masking(mask_value = 0) %>%
  layer_lstm(32)

output <- model(inputs)
# The time step 3 and 5 will be skipped from LSTM calculation.
```

Note

in the Keras masking convention, a masked timestep is denoted by a mask value of `FALSE`, while a non-masked (i.e. usable) timestep is denoted by a mask value of `TRUE`.

See Also

- https://keras.io/api/layers/core_layers/masking#masking-class

Other core layers:
`layer_dense()`
`layer_einsum_dense()`
`layer_embedding()`
`layer_identity()`
`layer_lambda()`

Other layers:
`Layer()`
`layer_activation()`
`layer_activation_elu()`
`layer_activation_leaky_relu()`

layer_activation_parametric_relu()
layer_activation_relu()
layer_activation_softmax()
layer_activity_regularization()
layer_add()
layer_additive_attention()
layer_alpha_dropout()
layer_attention()
layer_average()
layer_average_pooling_1d()
layer_average_pooling_2d()
layer_average_pooling_3d()
layer_batch_normalization()
layer_bidirectional()
layer_category_encoding()
layer_center_crop()
layer_concatenate()
layer_conv_1d()
layer_conv_1d_transpose()
layer_conv_2d()
layer_conv_2d_transpose()
layer_conv_3d()
layer_conv_3d_transpose()
layer_conv_lstm_1d()
layer_conv_lstm_2d()
layer_conv_lstm_3d()
layer_cropping_1d()
layer_cropping_2d()
layer_cropping_3d()
layer_dense()
layer_depthwise_conv_1d()
layer_depthwise_conv_2d()
layer_discretization()
layer_dot()
layer_dropout()
layer_einsum_dense()
layer_embedding()
layer_feature_space()
layer_flatten()
layer_flax_module_wrapper()
layer_gaussian_dropout()
layer_gaussian_noise()
layer_global_average_pooling_1d()
layer_global_average_pooling_2d()
layer_global_average_pooling_3d()
layer_global_max_pooling_1d()
layer_global_max_pooling_2d()
layer_global_max_pooling_3d()

layer_group_normalization()
layer_group_query_attention()
layer_gru()
layer_hashed_crossing()
layer_hashing()
layer_identity()
layer_integer_lookup()
layer_jax_model_wrapper()
layer_lambda()
layer_layer_normalization()
layer_lstm()
layer_max_pooling_1d()
layer_max_pooling_2d()
layer_max_pooling_3d()
layer_maximum()
layer_mel_spectrogram()
layer_minimum()
layer_multi_head_attention()
layer_multiply()
layer_normalization()
layer_permute()
layer_random_brightness()
layer_random_contrast()
layer_random_crop()
layer_random_flip()
layer_random_rotation()
layer_random_translation()
layer_random_zoom()
layer_repeat_vector()
layer_rescaling()
layer_reshape()
layer_resizing()
layer_rnn()
layer_separable_conv_1d()
layer_separable_conv_2d()
layer_simple_rnn()
layer_spatial_dropout_1d()
layer_spatial_dropout_2d()
layer_spatial_dropout_3d()
layer_spectral_normalization()
layer_string_lookup()
layer_subtract()
layer_text_vectorization()
layer_tfsm()
layer_time_distributed()
layer_torch_module_wrapper()
layer_unit_normalization()
layer_upsampling_1d()

layer_upsampling_2d()
layer_upsampling_3d()
layer_zero_padding_1d()
layer_zero_padding_2d()
layer_zero_padding_3d()
rnn_cell_gru()
rnn_cell_lstm()
rnn_cell_simple()
rnn_cells_stack()

---

layer_maximum *Computes element-wise maximum on a list of inputs.*

Description

It takes as input a list of tensors, all of the same shape, and returns a single tensor (also of the same shape).

Usage

```
layer_maximum(inputs, ...)
```

Arguments

inputs	layers to combine
...	For forward/backward compatability.

Value

The return value depends on the value provided for the first argument. If `object` is:

- a `keras_model_sequential()`, then the layer is added to the sequential model (which is modified in place). To enable piping, the sequential model is also returned, invisibly.
- a `keras_input()`, then the output tensor from calling `layer(input)` is returned.
- `NULL` or missing, then a `Layer` instance is returned.

Examples

```
input_shape <- c(2, 3, 4)
x1 <- random_uniform(input_shape)
x2 <- random_uniform(input_shape)
y <- layer_maximum(x1, x2)
```

Usage in a Keras model:

```
input1 <- layer_input(shape = c(16))
x1 <- input1 |> layer_dense(8, activation = 'relu')
input2 <- layer_input(shape = c(32))
x2 <- input2 |> layer_dense(8, activation = 'relu')
# equivalent to `y <- layer_maximum(x1, x2)`
y <- layer_maximum(x1, x2)
out <- y |> layer_dense(4)
model <- keras_model(inputs = c(input1, input2), outputs = out)
```

See Also

- https://keras.io/api/layers/merging_layers/maximum#maximum-class

Other merging layers:
layer_add()
layer_average()
layer_concatenate()
layer_dot()
layer_minimum()
layer_multiply()
layer_subtract()

Other layers:
Layer()
layer_activation()
layer_activation_elu()
layer_activation_leaky_relu()
layer_activation_parametric_relu()
layer_activation_relu()
layer_activation_softmax()
layer_activity_regularization()
layer_add()
layer_additive_attention()
layer_alpha_dropout()
layer_attention()
layer_average()
layer_average_pooling_1d()
layer_average_pooling_2d()
layer_average_pooling_3d()
layer_batch_normalization()
layer_bidirectional()
layer_category_encoding()
layer_center_crop()
layer_concatenate()
layer_conv_1d()
layer_conv_1d_transpose()
layer_conv_2d()
layer_conv_2d_transpose()
layer_conv_3d()

layer_conv_3d_transpose()
layer_conv_lstm_1d()
layer_conv_lstm_2d()
layer_conv_lstm_3d()
layer_cropping_1d()
layer_cropping_2d()
layer_cropping_3d()
layer_dense()
layer_depthwise_conv_1d()
layer_depthwise_conv_2d()
layer_discretization()
layer_dot()
layer_dropout()
layer_einsum_dense()
layer_embedding()
layer_feature_space()
layer_flatten()
layer_flax_module_wrapper()
layer_gaussian_dropout()
layer_gaussian_noise()
layer_global_average_pooling_1d()
layer_global_average_pooling_2d()
layer_global_average_pooling_3d()
layer_global_max_pooling_1d()
layer_global_max_pooling_2d()
layer_global_max_pooling_3d()
layer_group_normalization()
layer_group_query_attention()
layer_gru()
layer_hashed_crossing()
layer_hashing()
layer_identity()
layer_integer_lookup()
layer_jax_model_wrapper()
layer_lambda()
layer_layer_normalization()
layer_lstm()
layer_masking()
layer_max_pooling_1d()
layer_max_pooling_2d()
layer_max_pooling_3d()
layer_mel_spectrogram()
layer_minimum()
layer_multi_head_attention()
layer_multiply()
layer_normalization()
layer_permute()
layer_random_brightness()

layer_random_contrast()
layer_random_crop()
layer_random_flip()
layer_random_rotation()
layer_random_translation()
layer_random_zoom()
layer_repeat_vector()
layer_rescaling()
layer_reshape()
layer_resizing()
layer_rnn()
layer_separable_conv_1d()
layer_separable_conv_2d()
layer_simple_rnn()
layer_spatial_dropout_1d()
layer_spatial_dropout_2d()
layer_spatial_dropout_3d()
layer_spectral_normalization()
layer_string_lookup()
layer_subtract()
layer_text_vectorization()
layer_tfsm()
layer_time_distributed()
layer_torch_module_wrapper()
layer_unit_normalization()
layer_upsampling_1d()
layer_upsampling_2d()
layer_upsampling_3d()
layer_zero_padding_1d()
layer_zero_padding_2d()
layer_zero_padding_3d()
rnn_cell_gru()
rnn_cell_lstm()
rnn_cell_simple()
rnn_cells_stack()

---

layer_max_pooling_1d *Max pooling operation for 1D temporal data.*

Description

Downsamples the input representation by taking the maximum value over a spatial window of size pool_size. The window is shifted by strides.

The resulting output when using the "valid" padding option has a shape of: output_shape = (input_shape - pool_size

The resulting output shape when using the "same" padding option is: output_shape = input_shape / strides

Usage

```
layer_max_pooling_1d(
  object,
  pool_size = 2L,
  strides = NULL,
  padding = "valid",
  data_format = NULL,
  name = NULL,
  ...
)
```

Arguments

object	Object to compose the layer with. A tensor, array, or sequential model.
pool_size	int, size of the max pooling window.
strides	int or NULL. Specifies how much the pooling window moves for each pooling step. If NULL, it will default to pool_size.
padding	string, either "valid" or "same" (case-insensitive). "valid" means no padding. "same" results in padding evenly to the left/right or up/down of the input such that output has the same height/width dimension as the input.
data_format	string, either "channels_last" or "channels_first". The ordering of the dimensions in the inputs. "channels_last" corresponds to inputs with shape (batch, steps, features) while "channels_first" corresponds to inputs with shape (batch, features, steps). It defaults to the image_data_format value found in your Keras config file at ~/.keras/keras.json. If you never set it, then it will be "channels_last".
name	String, name for the object
...	For forward/backward compatability.

Value

The return value depends on the value provided for the first argument. If object is:

- a keras_model_sequential(), then the layer is added to the sequential model (which is modified in place). To enable piping, the sequential model is also returned, invisibly.
- a keras_input(), then the output tensor from calling layer(input) is returned.
- NULL or missing, then a Layer instance is returned.

Input Shape

- If data_format="channels_last": 3D tensor with shape (batch_size, steps, features).
- If data_format="channels_first": 3D tensor with shape (batch_size, features, steps).

Output Shape

- If data_format="channels_last": 3D tensor with shape (batch_size, downsampled_steps, features).
- If data_format="channels_first": 3D tensor with shape (batch_size, features, downsampled_steps).

Examples

strides=1 and padding="valid":

```
x <- op_reshape(c(1, 2, 3, 4, 5),
               c(1, 5, 1))
max_pool_1d <- layer_max_pooling_1d(pool_size = 2,
                                    strides = 1,
                                    padding = "valid")
max_pool_1d(x)
```

```
## tf.Tensor(
## [[[2.]
##   [3.]
##   [4.]
##   [5.]]], shape=(1, 4, 1), dtype=float32)
```

strides=2 and padding="valid":

```
x <- op_reshape(c(1, 2, 3, 4, 5),
               c(1, 5, 1))
max_pool_1d <- layer_max_pooling_1d(pool_size = 2,
                                    strides = 2,
                                    padding = "valid")
max_pool_1d(x)
```

```
## tf.Tensor(
## [[[2.]
##   [4.]]], shape=(1, 2, 1), dtype=float32)
```

strides=1 and padding="same":

```
x <- op_reshape(c(1, 2, 3, 4, 5),
               c(1, 5, 1))
max_pool_1d <- layer_max_pooling_1d(pool_size = 2,
                                    strides = 1,
                                    padding = "same")
max_pool_1d(x)
```

```
## tf.Tensor(
## [[[2.]
##   [3.]
##   [4.]
##   [5.]
##   [5.]]], shape=(1, 5, 1), dtype=float32)
```

See Also

- https://keras.io/api/layers/pooling_layers/max_pooling1d#maxpooling1d-class

Other pooling layers:
layer_average_pooling_1d()
layer_average_pooling_2d()
layer_average_pooling_3d()
layer_global_average_pooling_1d()
layer_global_average_pooling_2d()
layer_global_average_pooling_3d()
layer_global_max_pooling_1d()
layer_global_max_pooling_2d()
layer_global_max_pooling_3d()
layer_max_pooling_2d()
layer_max_pooling_3d()

Other layers:
Layer()
layer_activation()
layer_activation_elu()
layer_activation_leaky_relu()
layer_activation_parametric_relu()
layer_activation_relu()
layer_activation_softmax()
layer_activity_regularization()
layer_add()
layer_additive_attention()
layer_alpha_dropout()
layer_attention()
layer_average()
layer_average_pooling_1d()
layer_average_pooling_2d()
layer_average_pooling_3d()
layer_batch_normalization()
layer_bidirectional()
layer_category_encoding()
layer_center_crop()
layer_concatenate()
layer_conv_1d()
layer_conv_1d_transpose()
layer_conv_2d()
layer_conv_2d_transpose()
layer_conv_3d()
layer_conv_3d_transpose()
layer_conv_lstm_1d()
layer_conv_lstm_2d()
layer_conv_lstm_3d()
layer_cropping_1d()

layer_cropping_2d()
layer_cropping_3d()
layer_dense()
layer_depthwise_conv_1d()
layer_depthwise_conv_2d()
layer_discretization()
layer_dot()
layer_dropout()
layer_einsum_dense()
layer_embedding()
layer_feature_space()
layer_flatten()
layer_flax_module_wrapper()
layer_gaussian_dropout()
layer_gaussian_noise()
layer_global_average_pooling_1d()
layer_global_average_pooling_2d()
layer_global_average_pooling_3d()
layer_global_max_pooling_1d()
layer_global_max_pooling_2d()
layer_global_max_pooling_3d()
layer_group_normalization()
layer_group_query_attention()
layer_gru()
layer_hashed_crossing()
layer_hashing()
layer_identity()
layer_integer_lookup()
layer_jax_model_wrapper()
layer_lambda()
layer_layer_normalization()
layer_lstm()
layer_masking()
layer_max_pooling_2d()
layer_max_pooling_3d()
layer_maximum()
layer_mel_spectrogram()
layer_minimum()
layer_multi_head_attention()
layer_multiply()
layer_normalization()
layer_permute()
layer_random_brightness()
layer_random_contrast()
layer_random_crop()
layer_random_flip()
layer_random_rotation()
layer_random_translation()

layer_random_zoom()
layer_repeat_vector()
layer_rescaling()
layer_reshape()
layer_resizing()
layer_rnn()
layer_separable_conv_1d()
layer_separable_conv_2d()
layer_simple_rnn()
layer_spatial_dropout_1d()
layer_spatial_dropout_2d()
layer_spatial_dropout_3d()
layer_spectral_normalization()
layer_string_lookup()
layer_subtract()
layer_text_vectorization()
layer_tfsm()
layer_time_distributed()
layer_torch_module_wrapper()
layer_unit_normalization()
layer_upsampling_1d()
layer_upsampling_2d()
layer_upsampling_3d()
layer_zero_padding_1d()
layer_zero_padding_2d()
layer_zero_padding_3d()
rnn_cell_gru()
rnn_cell_lstm()
rnn_cell_simple()
rnn_cells_stack()

layer_max_pooling_2d *Max pooling operation for 2D spatial data.*

Description

Downsamples the input along its spatial dimensions (height and width) by taking the maximum value over an input window (of size defined by `pool_size`) for each channel of the input. The window is shifted by `strides` along each dimension.

The resulting output when using the `"valid"` padding option has a spatial shape (number of rows or columns) of: `output_shape = floor((input_shape - pool_size) / strides) + 1` (when `input_shape >= pool_size`)

The resulting output shape when using the `"same"` padding option is: `output_shape = floor((input_shape - 1) / strides) + 1`

Usage

```
layer_max_pooling_2d(
  object,
  pool_size = list(2L, 2L),
  strides = NULL,
  padding = "valid",
  data_format = NULL,
  name = NULL,
  ...
)
```

Arguments

`object`	Object to compose the layer with. A tensor, array, or sequential model.
`pool_size`	int or list of 2 integers, factors by which to downscale (dim1, dim2). If only one integer is specified, the same window length will be used for all dimensions.
`strides`	int or list of 2 integers, or `NULL`. Strides values. If `NULL`, it will default to `pool_size`. If only one int is specified, the same stride size will be used for all dimensions.
`padding`	string, either `"valid"` or `"same"` (case-insensitive). `"valid"` means no padding. `"same"` results in padding evenly to the left/right or up/down of the input such that output has the same height/width dimension as the input.
`data_format`	string, either `"channels_last"` or `"channels_first"`. The ordering of the dimensions in the inputs. `"channels_last"` corresponds to inputs with shape `(batch, height, width, channels)` while `"channels_first"` corresponds to inputs with shape `(batch, channels, height, width)`. It defaults to the `image_data_format` value found in your Keras config file at `~/.keras/keras.json`. If you never set it, then it will be `"channels_last"`.
`name`	String, name for the object
`...`	For forward/backward compatability.

Value

The return value depends on the value provided for the first argument. If `object` is:

- a `keras_model_sequential()`, then the layer is added to the sequential model (which is modified in place). To enable piping, the sequential model is also returned, invisibly.
- a `keras_input()`, then the output tensor from calling `layer(input)` is returned.
- `NULL` or missing, then a `Layer` instance is returned.

Input Shape

- If `data_format="channels_last"`: 4D tensor with shape `(batch_size, height, width, channels)`.
- If `data_format="channels_first"`: 4D tensor with shape `(batch_size, channels, height, width)`.

Output Shape

- If data_format="channels_last": 4D tensor with shape (batch_size, pooled_height, pooled_width, channel
- If data_format="channels_first": 4D tensor with shape (batch_size, channels, pooled_height, pooled_wid

Examples

strides = (1, 1) and padding = "valid":

```
x <- rbind(c(1., 2., 3.),
           c(4., 5., 6.),
           c(7., 8., 9.)) |> op_reshape(c(1, 3, 3, 1))
max_pool_2d <- layer_max_pooling_2d(pool_size = c(2, 2),
                                    strides = c(1, 1),
                                    padding = "valid")
max_pool_2d(x)

## tf.Tensor(
## [[[[5.]
##    [6.]]
##
##   [[8.]
##    [9.]]]], shape=(1, 2, 2, 1), dtype=float32)
```

strides = c(2, 2) and padding = "valid":

```
x <- rbind(c(1., 2., 3., 4.),
           c(5., 6., 7., 8.),
           c(9., 10., 11., 12.)) |> op_reshape(c(1, 3, 4, 1))
max_pool_2d <- layer_max_pooling_2d(pool_size = c(2, 2),
                                    strides = c(2, 2),
                                    padding = "valid")
max_pool_2d(x)

## tf.Tensor(
## [[[[6.]
##    [8.]]]], shape=(1, 1, 2, 1), dtype=float32)
```

stride = (1, 1) and padding = "same":

```
x <- rbind(c(1., 2., 3.),
           c(4., 5., 6.),
           c(7., 8., 9.)) |> op_reshape(c(1, 3, 3, 1))
max_pool_2d <- layer_max_pooling_2d(pool_size = c(2, 2),
                                    strides = c(1, 1),
                                    padding = "same")
max_pool_2d(x)
```

```
## tf.Tensor(
## [[[[5.]
##    [6.]
##    [6.]]
##
##   [[8.]
##    [9.]
##    [9.]]
##
##   [[8.]
##    [9.]
##    [9.]]]], shape=(1, 3, 3, 1), dtype=float32)
```

See Also

- https://keras.io/api/layers/pooling_layers/max_pooling2d#maxpooling2d-class

Other pooling layers:
layer_average_pooling_1d()
layer_average_pooling_2d()
layer_average_pooling_3d()
layer_global_average_pooling_1d()
layer_global_average_pooling_2d()
layer_global_average_pooling_3d()
layer_global_max_pooling_1d()
layer_global_max_pooling_2d()
layer_global_max_pooling_3d()
layer_max_pooling_1d()
layer_max_pooling_3d()

Other layers:
Layer()
layer_activation()
layer_activation_elu()
layer_activation_leaky_relu()
layer_activation_parametric_relu()
layer_activation_relu()
layer_activation_softmax()
layer_activity_regularization()
layer_add()
layer_additive_attention()
layer_alpha_dropout()
layer_attention()
layer_average()
layer_average_pooling_1d()
layer_average_pooling_2d()
layer_average_pooling_3d()
layer_batch_normalization()

layer_bidirectional()
layer_category_encoding()
layer_center_crop()
layer_concatenate()
layer_conv_1d()
layer_conv_1d_transpose()
layer_conv_2d()
layer_conv_2d_transpose()
layer_conv_3d()
layer_conv_3d_transpose()
layer_conv_lstm_1d()
layer_conv_lstm_2d()
layer_conv_lstm_3d()
layer_cropping_1d()
layer_cropping_2d()
layer_cropping_3d()
layer_dense()
layer_depthwise_conv_1d()
layer_depthwise_conv_2d()
layer_discretization()
layer_dot()
layer_dropout()
layer_einsum_dense()
layer_embedding()
layer_feature_space()
layer_flatten()
layer_flax_module_wrapper()
layer_gaussian_dropout()
layer_gaussian_noise()
layer_global_average_pooling_1d()
layer_global_average_pooling_2d()
layer_global_average_pooling_3d()
layer_global_max_pooling_1d()
layer_global_max_pooling_2d()
layer_global_max_pooling_3d()
layer_group_normalization()
layer_group_query_attention()
layer_gru()
layer_hashed_crossing()
layer_hashing()
layer_identity()
layer_integer_lookup()
layer_jax_model_wrapper()
layer_lambda()
layer_layer_normalization()
layer_lstm()
layer_masking()
layer_max_pooling_1d()

layer_max_pooling_3d()
layer_maximum()
layer_mel_spectrogram()
layer_minimum()
layer_multi_head_attention()
layer_multiply()
layer_normalization()
layer_permute()
layer_random_brightness()
layer_random_contrast()
layer_random_crop()
layer_random_flip()
layer_random_rotation()
layer_random_translation()
layer_random_zoom()
layer_repeat_vector()
layer_rescaling()
layer_reshape()
layer_resizing()
layer_rnn()
layer_separable_conv_1d()
layer_separable_conv_2d()
layer_simple_rnn()
layer_spatial_dropout_1d()
layer_spatial_dropout_2d()
layer_spatial_dropout_3d()
layer_spectral_normalization()
layer_string_lookup()
layer_subtract()
layer_text_vectorization()
layer_tfsm()
layer_time_distributed()
layer_torch_module_wrapper()
layer_unit_normalization()
layer_upsampling_1d()
layer_upsampling_2d()
layer_upsampling_3d()
layer_zero_padding_1d()
layer_zero_padding_2d()
layer_zero_padding_3d()
rnn_cell_gru()
rnn_cell_lstm()
rnn_cell_simple()
rnn_cells_stack()

layer_max_pooling_3d *Max pooling operation for 3D data (spatial or spatio-temporal).*

Description

Downsamples the input along its spatial dimensions (depth, height, and width) by taking the maximum value over an input window (of size defined by `pool_size`) for each channel of the input. The window is shifted by `strides` along each dimension.

Usage

```
layer_max_pooling_3d(
  object,
  pool_size = list(2L, 2L, 2L),
  strides = NULL,
  padding = "valid",
  data_format = NULL,
  name = NULL,
  ...
)
```

Arguments

`object`	Object to compose the layer with. A tensor, array, or sequential model.
`pool_size`	int or list of 3 integers, factors by which to downscale (dim1, dim2, dim3). If only one integer is specified, the same window length will be used for all dimensions.
`strides`	int or list of 3 integers, or `NULL`. Strides values. If `NULL`, it will default to `pool_size`. If only one int is specified, the same stride size will be used for all dimensions.
`padding`	string, either `"valid"` or `"same"` (case-insensitive). `"valid"` means no padding. `"same"` results in padding evenly to the left/right or up/down of the input such that output has the same height/width dimension as the input.
`data_format`	string, either `"channels_last"` or `"channels_first"`. The ordering of the dimensions in the inputs. `"channels_last"` corresponds to inputs with shape `(batch, spatial_dim1, spatial_dim2, spatial_dim3, channels)` while `"channels_first"` corresponds to inputs with shape `(batch, channels, spatial_dim1, spatial_di` It defaults to the `image_data_format` value found in your Keras config file at `~/.keras/keras.json`. If you never set it, then it will be `"channels_last"`.
`name`	String, name for the object
`...`	For forward/backward compatability.

Value

The return value depends on the value provided for the first argument. If `object` is:

- a `keras_model_sequential()`, then the layer is added to the sequential model (which is modified in place). To enable piping, the sequential model is also returned, invisibly.
- a `keras_input()`, then the output tensor from calling `layer(input)` is returned.
- `NULL` or missing, then a `Layer` instance is returned.

Input Shape

- If data_format="channels_last": 5D tensor with shape: (batch_size, spatial_dim1, spatial_dim2, spatial_
- If data_format="channels_first": 5D tensor with shape: (batch_size, channels, spatial_dim1, spatial_di

Output Shape

- If data_format="channels_last": 5D tensor with shape: (batch_size, pooled_dim1, pooled_dim2, pooled_di
- If data_format="channels_first": 5D tensor with shape: (batch_size, channels, pooled_dim1, pooled_dim2

Examples

```
depth <- 30
height <- 30
width <- 30
channels <- 3

inputs <- layer_input(shape=c(depth, height, width, channels))
layer <- layer_max_pooling_3d(pool_size=3)
outputs <- inputs |> layer()
outputs

## <KerasTensor shape=(None, 10, 10, 10, 3), dtype=float32, sparse=False, name=keras_tensor_1>
```

See Also

- https://keras.io/api/layers/pooling_layers/max_pooling3d#maxpooling3d-class

Other pooling layers:
layer_average_pooling_1d()
layer_average_pooling_2d()
layer_average_pooling_3d()
layer_global_average_pooling_1d()
layer_global_average_pooling_2d()
layer_global_average_pooling_3d()
layer_global_max_pooling_1d()
layer_global_max_pooling_2d()
layer_global_max_pooling_3d()
layer_max_pooling_1d()
layer_max_pooling_2d()

Other layers:
Layer()
layer_activation()
layer_activation_elu()
layer_activation_leaky_relu()
layer_activation_parametric_relu()
layer_activation_relu()

layer_activation_softmax()
layer_activity_regularization()
layer_add()
layer_additive_attention()
layer_alpha_dropout()
layer_attention()
layer_average()
layer_average_pooling_1d()
layer_average_pooling_2d()
layer_average_pooling_3d()
layer_batch_normalization()
layer_bidirectional()
layer_category_encoding()
layer_center_crop()
layer_concatenate()
layer_conv_1d()
layer_conv_1d_transpose()
layer_conv_2d()
layer_conv_2d_transpose()
layer_conv_3d()
layer_conv_3d_transpose()
layer_conv_lstm_1d()
layer_conv_lstm_2d()
layer_conv_lstm_3d()
layer_cropping_1d()
layer_cropping_2d()
layer_cropping_3d()
layer_dense()
layer_depthwise_conv_1d()
layer_depthwise_conv_2d()
layer_discretization()
layer_dot()
layer_dropout()
layer_einsum_dense()
layer_embedding()
layer_feature_space()
layer_flatten()
layer_flax_module_wrapper()
layer_gaussian_dropout()
layer_gaussian_noise()
layer_global_average_pooling_1d()
layer_global_average_pooling_2d()
layer_global_average_pooling_3d()
layer_global_max_pooling_1d()
layer_global_max_pooling_2d()
layer_global_max_pooling_3d()
layer_group_normalization()
layer_group_query_attention()

layer_gru()
layer_hashed_crossing()
layer_hashing()
layer_identity()
layer_integer_lookup()
layer_jax_model_wrapper()
layer_lambda()
layer_layer_normalization()
layer_lstm()
layer_masking()
layer_max_pooling_1d()
layer_max_pooling_2d()
layer_maximum()
layer_mel_spectrogram()
layer_minimum()
layer_multi_head_attention()
layer_multiply()
layer_normalization()
layer_permute()
layer_random_brightness()
layer_random_contrast()
layer_random_crop()
layer_random_flip()
layer_random_rotation()
layer_random_translation()
layer_random_zoom()
layer_repeat_vector()
layer_rescaling()
layer_reshape()
layer_resizing()
layer_rnn()
layer_separable_conv_1d()
layer_separable_conv_2d()
layer_simple_rnn()
layer_spatial_dropout_1d()
layer_spatial_dropout_2d()
layer_spatial_dropout_3d()
layer_spectral_normalization()
layer_string_lookup()
layer_subtract()
layer_text_vectorization()
layer_tfsm()
layer_time_distributed()
layer_torch_module_wrapper()
layer_unit_normalization()
layer_upsampling_1d()
layer_upsampling_2d()
layer_upsampling_3d()

layer_zero_padding_1d()
layer_zero_padding_2d()
layer_zero_padding_3d()
rnn_cell_gru()
rnn_cell_lstm()
rnn_cell_simple()
rnn_cells_stack()

---

layer_mel_spectrogram *A preprocessing layer to convert raw audio signals to Mel spectrograms.*

Description

This layer takes float32/float64 single or batched audio signal as inputs and computes the Mel spectrogram using Short-Time Fourier Transform and Mel scaling. The input should be a 1D (unbatched) or 2D (batched) tensor representing audio signals. The output will be a 2D or 3D tensor representing Mel spectrograms.

A spectrogram is an image-like representation that shows the frequency spectrum of a signal over time. It uses x-axis to represent time, y-axis to represent frequency, and each pixel to represent intensity. Mel spectrograms are a special type of spectrogram that use the mel scale, which approximates how humans perceive sound. They are commonly used in speech and music processing tasks like speech recognition, speaker identification, and music genre classification.

Usage

```
layer_mel_spectrogram(
  object,
  fft_length = 2048L,
  sequence_stride = 512L,
  sequence_length = NULL,
  window = "hann",
  sampling_rate = 16000L,
  num_mel_bins = 128L,
  min_freq = 20,
  max_freq = NULL,
  power_to_db = TRUE,
  top_db = 80,
  mag_exp = 2,
  min_power = 1e-10,
  ref_power = 1,
  ...
)
```

Arguments

`object`	Object to compose the layer with. A tensor, array, or sequential model.
`fft_length`	Integer, size of the FFT window.
`sequence_stride`	Integer, number of samples between successive STFT columns.
`sequence_length`	Integer, size of the window used for applying `window` to each audio frame. If `NULL`, defaults to `fft_length`.
`window`	String, name of the window function to use. Available values are `"hann"` and `"hamming"`. If `window` is a tensor, it will be used directly as the window and its length must be `sequence_length`. If `window` is `NULL`, no windowing is used. Defaults to `"hann"`.
`sampling_rate`	Integer, sample rate of the input signal.
`num_mel_bins`	Integer, number of mel bins to generate.
`min_freq`	Float, minimum frequency of the mel bins.
`max_freq`	Float, maximum frequency of the mel bins. If `NULL`, defaults to `sampling_rate / 2`.
`power_to_db`	If TRUE, convert the power spectrogram to decibels.
`top_db`	Float, minimum negative cut-off `max(10 * log10(S)) - top_db`.
`mag_exp`	Float, exponent for the magnitude spectrogram. 1 for magnitude, 2 for power, etc. Default is 2.
`min_power`	Float, minimum value for power and `ref_power`.
`ref_power`	Float, the power is scaled relative to it `10 * log10(S / ref_power)`.
`...`	For forward/backward compatability.

Value

The return value depends on the value provided for the first argument. If `object` is:

- a `keras_model_sequential()`, then the layer is added to the sequential model (which is modified in place). To enable piping, the sequential model is also returned, invisibly.
- a `keras_input()`, then the output tensor from calling `layer(input)` is returned.
- `NULL` or missing, then a `Layer` instance is returned.

References

- Spectrogram,
- Mel scale.

Examples

Unbatched audio signal

```
layer <- layer_mel_spectrogram(
  num_mel_bins = 64,
  sampling_rate = 8000,
  sequence_stride = 256,
  fft_length = 2048
)
layer(random_uniform(shape = c(16000))) |> shape()
```

Batched audio signal

```
layer <- layer_mel_spectrogram(
  num_mel_bins = 80,
  sampling_rate = 8000,
  sequence_stride = 128,
  fft_length = 2048
)
layer(random_uniform(shape = c(2, 16000))) |> shape()
```

Input Shape

1D (unbatched) or 2D (batched) tensor with shape:`(..., samples)`.

Output Shape

2D (unbatched) or 3D (batched) tensor with shape:`(..., num_mel_bins, time)`.

See Also

Other preprocessing layers:
`layer_category_encoding()`
`layer_center_crop()`
`layer_discretization()`
`layer_feature_space()`
`layer_hashed_crossing()`
`layer_hashing()`
`layer_integer_lookup()`
`layer_normalization()`
`layer_random_brightness()`
`layer_random_contrast()`
`layer_random_crop()`
`layer_random_flip()`
`layer_random_rotation()`
`layer_random_translation()`
`layer_random_zoom()`
`layer_rescaling()`

layer_resizing()
layer_string_lookup()
layer_text_vectorization()

Other layers:
Layer()
layer_activation()
layer_activation_elu()
layer_activation_leaky_relu()
layer_activation_parametric_relu()
layer_activation_relu()
layer_activation_softmax()
layer_activity_regularization()
layer_add()
layer_additive_attention()
layer_alpha_dropout()
layer_attention()
layer_average()
layer_average_pooling_1d()
layer_average_pooling_2d()
layer_average_pooling_3d()
layer_batch_normalization()
layer_bidirectional()
layer_category_encoding()
layer_center_crop()
layer_concatenate()
layer_conv_1d()
layer_conv_1d_transpose()
layer_conv_2d()
layer_conv_2d_transpose()
layer_conv_3d()
layer_conv_3d_transpose()
layer_conv_lstm_1d()
layer_conv_lstm_2d()
layer_conv_lstm_3d()
layer_cropping_1d()
layer_cropping_2d()
layer_cropping_3d()
layer_dense()
layer_depthwise_conv_1d()
layer_depthwise_conv_2d()
layer_discretization()
layer_dot()
layer_dropout()
layer_einsum_dense()
layer_embedding()
layer_feature_space()
layer_flatten()

layer_flax_module_wrapper()
layer_gaussian_dropout()
layer_gaussian_noise()
layer_global_average_pooling_1d()
layer_global_average_pooling_2d()
layer_global_average_pooling_3d()
layer_global_max_pooling_1d()
layer_global_max_pooling_2d()
layer_global_max_pooling_3d()
layer_group_normalization()
layer_group_query_attention()
layer_gru()
layer_hashed_crossing()
layer_hashing()
layer_identity()
layer_integer_lookup()
layer_jax_model_wrapper()
layer_lambda()
layer_layer_normalization()
layer_lstm()
layer_masking()
layer_max_pooling_1d()
layer_max_pooling_2d()
layer_max_pooling_3d()
layer_maximum()
layer_minimum()
layer_multi_head_attention()
layer_multiply()
layer_normalization()
layer_permute()
layer_random_brightness()
layer_random_contrast()
layer_random_crop()
layer_random_flip()
layer_random_rotation()
layer_random_translation()
layer_random_zoom()
layer_repeat_vector()
layer_rescaling()
layer_reshape()
layer_resizing()
layer_rnn()
layer_separable_conv_1d()
layer_separable_conv_2d()
layer_simple_rnn()
layer_spatial_dropout_1d()
layer_spatial_dropout_2d()
layer_spatial_dropout_3d()

layer_spectral_normalization()
layer_string_lookup()
layer_subtract()
layer_text_vectorization()
layer_tfsm()
layer_time_distributed()
layer_torch_module_wrapper()
layer_unit_normalization()
layer_upsampling_1d()
layer_upsampling_2d()
layer_upsampling_3d()
layer_zero_padding_1d()
layer_zero_padding_2d()
layer_zero_padding_3d()
rnn_cell_gru()
rnn_cell_lstm()
rnn_cell_simple()
rnn_cells_stack()

layer_minimum *Computes elementwise minimum on a list of inputs.*

Description

It takes as input a list of tensors, all of the same shape, and returns a single tensor (also of the same shape).

Usage

```
layer_minimum(inputs, ...)
```

Arguments

inputs	layers to combine
...	For forward/backward compatability.

Value

The return value depends on the value provided for the first argument. If `object` is:

- a `keras_model_sequential()`, then the layer is added to the sequential model (which is modified in place). To enable piping, the sequential model is also returned, invisibly.
- a `keras_input()`, then the output tensor from calling `layer(input)` is returned.
- `NULL` or missing, then a `Layer` instance is returned.

Examples

```
input_shape <- c(2, 3, 4)
x1 <- random_uniform(input_shape)
x2 <- random_uniform(input_shape)
y <- layer_minimum(x1, x2)
```

Usage in a Keras model:

```
input1 <- layer_input(shape = c(16))
x1 <- input1 |> layer_dense(8, activation = 'relu')
input2 <- layer_input(shape = c(32))
x2 <- input2 |> layer_dense(8, activation = 'relu')
# equivalent to `y <- layer_minimum(x1, x2)`
y <- layer_minimum(x1, x2)
out <- y |> layer_dense(4)
model <- keras_model(inputs = c(input1, input2), outputs = out)
```

See Also

- https://keras.io/api/layers/merging_layers/minimum#minimum-class

Other merging layers:
layer_add()
layer_average()
layer_concatenate()
layer_dot()
layer_maximum()
layer_multiply()
layer_subtract()

Other layers:
Layer()
layer_activation()
layer_activation_elu()
layer_activation_leaky_relu()
layer_activation_parametric_relu()
layer_activation_relu()
layer_activation_softmax()
layer_activity_regularization()
layer_add()
layer_additive_attention()
layer_alpha_dropout()
layer_attention()
layer_average()
layer_average_pooling_1d()
layer_average_pooling_2d()
layer_average_pooling_3d()
layer_batch_normalization()
layer_bidirectional()

layer_category_encoding()
layer_center_crop()
layer_concatenate()
layer_conv_1d()
layer_conv_1d_transpose()
layer_conv_2d()
layer_conv_2d_transpose()
layer_conv_3d()
layer_conv_3d_transpose()
layer_conv_lstm_1d()
layer_conv_lstm_2d()
layer_conv_lstm_3d()
layer_cropping_1d()
layer_cropping_2d()
layer_cropping_3d()
layer_dense()
layer_depthwise_conv_1d()
layer_depthwise_conv_2d()
layer_discretization()
layer_dot()
layer_dropout()
layer_einsum_dense()
layer_embedding()
layer_feature_space()
layer_flatten()
layer_flax_module_wrapper()
layer_gaussian_dropout()
layer_gaussian_noise()
layer_global_average_pooling_1d()
layer_global_average_pooling_2d()
layer_global_average_pooling_3d()
layer_global_max_pooling_1d()
layer_global_max_pooling_2d()
layer_global_max_pooling_3d()
layer_group_normalization()
layer_group_query_attention()
layer_gru()
layer_hashed_crossing()
layer_hashing()
layer_identity()
layer_integer_lookup()
layer_jax_model_wrapper()
layer_lambda()
layer_layer_normalization()
layer_lstm()
layer_masking()
layer_max_pooling_1d()
layer_max_pooling_2d()

layer_max_pooling_3d()
layer_maximum()
layer_mel_spectrogram()
layer_multi_head_attention()
layer_multiply()
layer_normalization()
layer_permute()
layer_random_brightness()
layer_random_contrast()
layer_random_crop()
layer_random_flip()
layer_random_rotation()
layer_random_translation()
layer_random_zoom()
layer_repeat_vector()
layer_rescaling()
layer_reshape()
layer_resizing()
layer_rnn()
layer_separable_conv_1d()
layer_separable_conv_2d()
layer_simple_rnn()
layer_spatial_dropout_1d()
layer_spatial_dropout_2d()
layer_spatial_dropout_3d()
layer_spectral_normalization()
layer_string_lookup()
layer_subtract()
layer_text_vectorization()
layer_tfsm()
layer_time_distributed()
layer_torch_module_wrapper()
layer_unit_normalization()
layer_upsampling_1d()
layer_upsampling_2d()
layer_upsampling_3d()
layer_zero_padding_1d()
layer_zero_padding_2d()
layer_zero_padding_3d()
rnn_cell_gru()
rnn_cell_lstm()
rnn_cell_simple()
rnn_cells_stack()

layer_multiply *Performs elementwise multiplication.*

Description

It takes as input a list of tensors, all of the same shape, and returns a single tensor (also of the same shape).

Usage

```
layer_multiply(inputs, ...)
```

Arguments

`inputs`	layers to combine
`...`	For forward/backward compatability.

Value

The return value depends on the value provided for the first argument. If `object` is:

- a `keras_model_sequential()`, then the layer is added to the sequential model (which is modified in place). To enable piping, the sequential model is also returned, invisibly.
- a `keras_input()`, then the output tensor from calling `layer(input)` is returned.
- `NULL` or missing, then a `Layer` instance is returned.

Examples

```
input_shape <- c(2, 3, 4)
x1 <- random_uniform(input_shape)
x2 <- random_uniform(input_shape)
y <- layer_multiply(x1, x2)
```

Usage in a Keras model:

```
input1 <- layer_input(shape = c(16))
x1 <- input1 |> layer_dense(8, activation = 'relu')
input2 <- layer_input(shape = c(32))
x2 <- input2 |> layer_dense(8, activation = 'relu')
# equivalent to `y <- layer_multiply(x1, x2)`
y <- layer_multiply(x1, x2)
out <- y |> layer_dense(4)
model <- keras_model(inputs = c(input1, input2), outputs = out)
```

See Also

- https://keras.io/api/layers/merging_layers/multiply#multiply-class

Other merging layers:
`layer_add()`
`layer_average()`
`layer_concatenate()`
`layer_dot()`
`layer_maximum()`

layer_minimum()
layer_subtract()

Other layers:
Layer()
layer_activation()
layer_activation_elu()
layer_activation_leaky_relu()
layer_activation_parametric_relu()
layer_activation_relu()
layer_activation_softmax()
layer_activity_regularization()
layer_add()
layer_additive_attention()
layer_alpha_dropout()
layer_attention()
layer_average()
layer_average_pooling_1d()
layer_average_pooling_2d()
layer_average_pooling_3d()
layer_batch_normalization()
layer_bidirectional()
layer_category_encoding()
layer_center_crop()
layer_concatenate()
layer_conv_1d()
layer_conv_1d_transpose()
layer_conv_2d()
layer_conv_2d_transpose()
layer_conv_3d()
layer_conv_3d_transpose()
layer_conv_lstm_1d()
layer_conv_lstm_2d()
layer_conv_lstm_3d()
layer_cropping_1d()
layer_cropping_2d()
layer_cropping_3d()
layer_dense()
layer_depthwise_conv_1d()
layer_depthwise_conv_2d()
layer_discretization()
layer_dot()
layer_dropout()
layer_einsum_dense()
layer_embedding()
layer_feature_space()
layer_flatten()
layer_flax_module_wrapper()

layer_gaussian_dropout()
layer_gaussian_noise()
layer_global_average_pooling_1d()
layer_global_average_pooling_2d()
layer_global_average_pooling_3d()
layer_global_max_pooling_1d()
layer_global_max_pooling_2d()
layer_global_max_pooling_3d()
layer_group_normalization()
layer_group_query_attention()
layer_gru()
layer_hashed_crossing()
layer_hashing()
layer_identity()
layer_integer_lookup()
layer_jax_model_wrapper()
layer_lambda()
layer_layer_normalization()
layer_lstm()
layer_masking()
layer_max_pooling_1d()
layer_max_pooling_2d()
layer_max_pooling_3d()
layer_maximum()
layer_mel_spectrogram()
layer_minimum()
layer_multi_head_attention()
layer_normalization()
layer_permute()
layer_random_brightness()
layer_random_contrast()
layer_random_crop()
layer_random_flip()
layer_random_rotation()
layer_random_translation()
layer_random_zoom()
layer_repeat_vector()
layer_rescaling()
layer_reshape()
layer_resizing()
layer_rnn()
layer_separable_conv_1d()
layer_separable_conv_2d()
layer_simple_rnn()
layer_spatial_dropout_1d()
layer_spatial_dropout_2d()
layer_spatial_dropout_3d()
layer_spectral_normalization()

layer_string_lookup()
layer_subtract()
layer_text_vectorization()
layer_tfsm()
layer_time_distributed()
layer_torch_module_wrapper()
layer_unit_normalization()
layer_upsampling_1d()
layer_upsampling_2d()
layer_upsampling_3d()
layer_zero_padding_1d()
layer_zero_padding_2d()
layer_zero_padding_3d()
rnn_cell_gru()
rnn_cell_lstm()
rnn_cell_simple()
rnn_cells_stack()

---

layer_multi_head_attention   *Multi Head Attention layer.*

Description

This is an implementation of multi-headed attention as described in the paper "Attention is all you Need" Vaswani et al., 2017. If query, key, value are the same, then this is self-attention. Each timestep in query attends to the corresponding sequence in key, and returns a fixed-width vector.

This layer first projects query, key and value. These are (effectively) a list of tensors of length num_attention_heads, where the corresponding shapes are (batch_size, <query dimensions>, key_dim), (batch_size, <key/value dimensions>, key_dim), (batch_size, <key/value dimensions>, value_dim).

Then, the query and key tensors are dot-producted and scaled. These are softmaxed to obtain attention probabilities. The value tensors are then interpolated by these probabilities, then concatenated back to a single tensor.

Finally, the result tensor with the last dimension as value_dim can take a linear projection and return.

Usage

```
layer_multi_head_attention(
  inputs,
  num_heads,
  key_dim,
  value_dim = NULL,
  dropout = 0,
  use_bias = TRUE,
  output_shape = NULL,
  attention_axes = NULL,
  kernel_initializer = "glorot_uniform",
  bias_initializer = "zeros",
  kernel_regularizer = NULL,
  bias_regularizer = NULL,
  activity_regularizer = NULL,
  kernel_constraint = NULL,
  bias_constraint = NULL,
  ...
)
```

Arguments

`inputs`	see description
`num_heads`	Number of attention heads.
`key_dim`	Size of each attention head for query and key.
`value_dim`	Size of each attention head for value.
`dropout`	Dropout probability.
`use_bias`	Boolean, whether the dense layers use bias vectors/matrices.
`output_shape`	The expected shape of an output tensor, besides the batch and sequence dims. If not specified, projects back to the query feature dim (the query input's last dimension).
`attention_axes`	axes over which the attention is applied. `NULL` means attention over all axes, but batch, heads, and features.
`kernel_initializer`	Initializer for dense layer kernels.
`bias_initializer`	Initializer for dense layer biases.
`kernel_regularizer`	Regularizer for dense layer kernels.
`bias_regularizer`	Regularizer for dense layer biases.
`activity_regularizer`	Regularizer for dense layer activity.
`kernel_constraint`	Constraint for dense layer kernels.
`bias_constraint`	Constraint for dense layer kernels.
`...`	For forward/backward compatability.

Value

The return value depends on the value provided for the first argument. If `object` is:

- a `keras_model_sequential()`, then the layer is added to the sequential model (which is modified in place). To enable piping, the sequential model is also returned, invisibly.
- a `keras_input()`, then the output tensor from calling `layer(input)` is returned.
- `NULL` or missing, then a `Layer` instance is returned.

Call Arguments

- `query`: Query tensor of shape `(B, T, dim)`, where `B` is the batch size, `T` is the target sequence length, and dim is the feature dimension.
- `value`: Value tensor of shape `(B, S, dim)`, where `B` is the batch size, `S` is the source sequence length, and dim is the feature dimension.
- `key`: Optional key tensor of shape `(B, S, dim)`. If not given, will use `value` for both `key` and `value`, which is the most common case.
- `attention_mask`: a boolean mask of shape `(B, T, S)`, that prevents attention to certain positions. The boolean mask specifies which query elements can attend to which key elements, 1 indicates attention and 0 indicates no attention. Broadcasting can happen for the missing batch dimensions and the head dimension.
- `return_attention_scores`: A boolean to indicate whether the output should be `(attention_output, attention_sc` if `TRUE`, or `attention_output` if `FALSE`. Defaults to `FALSE`.
- `training`: Python boolean indicating whether the layer should behave in training mode (adding dropout) or in inference mode (no dropout). Will go with either using the training mode of the parent layer/model, or `FALSE` (inference) if there is no parent layer.
- `use_causal_mask`: A boolean to indicate whether to apply a causal mask to prevent tokens from attending to future tokens (e.g., used in a decoder Transformer).

Call return

- attention_output: The result of the computation, of shape `(B, T, E)`, where `T` is for target sequence shapes and `E` is the query input last dimension if `output_shape` is `NULL`. Otherwise, the multi-head outputs are projected to the shape specified by `output_shape`.
- attention_scores: (Optional) multi-head attention coefficients over attention axes.

Properties

A `MultiHeadAttention` `Layer` instance has the following additional read-only properties:

- `attention_axes`
- `dropout`
- `key_dense`
- `key_dim`
- `num_heads`
- `output_dense`
- `output_shape`
- `query_dense`
- `use_bias`
- `value_dense`
- `value_dim`

See Also

- https://keras.io/api/layers/attention_layers/multi_head_attention#multiheadattention-class

Other attention layers:
`layer_additive_attention()`
`layer_attention()`
`layer_group_query_attention()`

Other layers:
`Layer()`
`layer_activation()`
`layer_activation_elu()`
`layer_activation_leaky_relu()`
`layer_activation_parametric_relu()`
`layer_activation_relu()`
`layer_activation_softmax()`
`layer_activity_regularization()`
`layer_add()`
`layer_additive_attention()`
`layer_alpha_dropout()`
`layer_attention()`
`layer_average()`
`layer_average_pooling_1d()`
`layer_average_pooling_2d()`
`layer_average_pooling_3d()`
`layer_batch_normalization()`
`layer_bidirectional()`
`layer_category_encoding()`
`layer_center_crop()`
`layer_concatenate()`
`layer_conv_1d()`
`layer_conv_1d_transpose()`
`layer_conv_2d()`
`layer_conv_2d_transpose()`
`layer_conv_3d()`
`layer_conv_3d_transpose()`
`layer_conv_lstm_1d()`
`layer_conv_lstm_2d()`
`layer_conv_lstm_3d()`
`layer_cropping_1d()`
`layer_cropping_2d()`
`layer_cropping_3d()`
`layer_dense()`
`layer_depthwise_conv_1d()`
`layer_depthwise_conv_2d()`
`layer_discretization()`
`layer_dot()`
`layer_dropout()`

layer_einsum_dense()
layer_embedding()
layer_feature_space()
layer_flatten()
layer_flax_module_wrapper()
layer_gaussian_dropout()
layer_gaussian_noise()
layer_global_average_pooling_1d()
layer_global_average_pooling_2d()
layer_global_average_pooling_3d()
layer_global_max_pooling_1d()
layer_global_max_pooling_2d()
layer_global_max_pooling_3d()
layer_group_normalization()
layer_group_query_attention()
layer_gru()
layer_hashed_crossing()
layer_hashing()
layer_identity()
layer_integer_lookup()
layer_jax_model_wrapper()
layer_lambda()
layer_layer_normalization()
layer_lstm()
layer_masking()
layer_max_pooling_1d()
layer_max_pooling_2d()
layer_max_pooling_3d()
layer_maximum()
layer_mel_spectrogram()
layer_minimum()
layer_multiply()
layer_normalization()
layer_permute()
layer_random_brightness()
layer_random_contrast()
layer_random_crop()
layer_random_flip()
layer_random_rotation()
layer_random_translation()
layer_random_zoom()
layer_repeat_vector()
layer_rescaling()
layer_reshape()
layer_resizing()
layer_rnn()
layer_separable_conv_1d()
layer_separable_conv_2d()

layer_simple_rnn()
layer_spatial_dropout_1d()
layer_spatial_dropout_2d()
layer_spatial_dropout_3d()
layer_spectral_normalization()
layer_string_lookup()
layer_subtract()
layer_text_vectorization()
layer_tfsm()
layer_time_distributed()
layer_torch_module_wrapper()
layer_unit_normalization()
layer_upsampling_1d()
layer_upsampling_2d()
layer_upsampling_3d()
layer_zero_padding_1d()
layer_zero_padding_2d()
layer_zero_padding_3d()
rnn_cell_gru()
rnn_cell_lstm()
rnn_cell_simple()
rnn_cells_stack()

layer_normalization — *A preprocessing layer that normalizes continuous features.*

Description

This layer will shift and scale inputs into a distribution centered around 0 with standard deviation 1. It accomplishes this by precomputing the mean and variance of the data, and calling `(input - mean) / sqrt(var)` at runtime.

The mean and variance values for the layer must be either supplied on construction or learned via `adapt()`. `adapt()` will compute the mean and variance of the data and store them as the layer's weights. `adapt()` should be called before `fit()`, `evaluate()`, or `predict()`.

Usage

```
layer_normalization(
  object,
  axis = -1L,
  mean = NULL,
  variance = NULL,
  invert = FALSE,
  ...
)
```

Arguments

`object`	Object to compose the layer with. A tensor, array, or sequential model.
`axis`	Integer, list of integers, or NULL. The axis or axes that should have a separate mean and variance for each index in the shape. For example, if shape is `(NULL, 5)` and `axis=1`, the layer will track 5 separate mean and variance values for the last axis. If `axis` is set to `NULL`, the layer will normalize all elements in the input by a scalar mean and variance. When `-1`, the last axis of the input is assumed to be a feature dimension and is normalized per index. Note that in the specific case of batched scalar inputs where the only axis is the batch axis, the default will normalize each index in the batch separately. In this case, consider passing `axis=NULL`. Defaults to `-1`.
`mean`	The mean value(s) to use during normalization. The passed value(s) will be broadcast to the shape of the kept axes above; if the value(s) cannot be broadcast, an error will be raised when this layer's `build()` method is called.
`variance`	The variance value(s) to use during normalization. The passed value(s) will be broadcast to the shape of the kept axes above; if the value(s) cannot be broadcast, an error will be raised when this layer's `build()` method is called.
`invert`	If `TRUE`, this layer will apply the inverse transformation to its inputs: it would turn a normalized input back into its original form.
`...`	For forward/backward compatability.

Value

The return value depends on the value provided for the first argument. If `object` is:

- a `keras_model_sequential()`, then the layer is added to the sequential model (which is modified in place). To enable piping, the sequential model is also returned, invisibly.
- a `keras_input()`, then the output tensor from calling `layer(input)` is returned.
- `NULL` or missing, then a `Layer` instance is returned.

Examples

Calculate a global mean and variance by analyzing the dataset in `adapt()`.

```
adapt_data <- op_array(c(1., 2., 3., 4., 5.), dtype='float32')
input_data <- op_array(c(1., 2., 3.), dtype='float32')
layer <- layer_normalization(axis = NULL)
layer %>% adapt(adapt_data)
layer(input_data)
```

```
## tf.Tensor([-1.4142135  -0.70710677  0.        ], shape=(3), dtype=float32)
```

Calculate a mean and variance for each index on the last axis.

```
adapt_data <- op_array(rbind(c(0., 7., 4.),
                       c(2., 9., 6.),
                       c(0., 7., 4.),
                       c(2., 9., 6.)), dtype='float32')
input_data <- op_array(matrix(c(0., 7., 4.), nrow = 1), dtype='float32')
layer <- layer_normalization(axis=-1)
layer %>% adapt(adapt_data)
layer(input_data)

## tf.Tensor([[-1. -1. -1.]], shape=(1, 3), dtype=float32)
```

Pass the mean and variance directly.

```
input_data <- op_array(rbind(1, 2, 3), dtype='float32')
layer <- layer_normalization(mean=3., variance=2.)
layer(input_data)

## tf.Tensor(
## [[-1.4142135 ]
##  [-0.70710677]
##  [ 0.        ]], shape=(3, 1), dtype=float32)
```

Use the layer to de-normalize inputs (after adapting the layer).

```
adapt_data <- op_array(rbind(c(0., 7., 4.),
                       c(2., 9., 6.),
                       c(0., 7., 4.),
                       c(2., 9., 6.)), dtype='float32')
input_data <- op_array(c(1., 2., 3.), dtype='float32')
layer <- layer_normalization(axis=-1, invert=TRUE)
layer %>% adapt(adapt_data)
layer(input_data)

## tf.Tensor([[ 2. 10.  8.]], shape=(1, 3), dtype=float32)
```

See Also

- https://keras.io/api/layers/preprocessing_layers/numerical/normalization#normalization-class

Other numerical features preprocessing layers:
`layer_discretization()`

Other preprocessing layers:
`layer_category_encoding()`
`layer_center_crop()`

layer_discretization()
layer_feature_space()
layer_hashed_crossing()
layer_hashing()
layer_integer_lookup()
layer_mel_spectrogram()
layer_random_brightness()
layer_random_contrast()
layer_random_crop()
layer_random_flip()
layer_random_rotation()
layer_random_translation()
layer_random_zoom()
layer_rescaling()
layer_resizing()
layer_string_lookup()
layer_text_vectorization()

Other layers:
Layer()
layer_activation()
layer_activation_elu()
layer_activation_leaky_relu()
layer_activation_parametric_relu()
layer_activation_relu()
layer_activation_softmax()
layer_activity_regularization()
layer_add()
layer_additive_attention()
layer_alpha_dropout()
layer_attention()
layer_average()
layer_average_pooling_1d()
layer_average_pooling_2d()
layer_average_pooling_3d()
layer_batch_normalization()
layer_bidirectional()
layer_category_encoding()
layer_center_crop()
layer_concatenate()
layer_conv_1d()
layer_conv_1d_transpose()
layer_conv_2d()
layer_conv_2d_transpose()
layer_conv_3d()
layer_conv_3d_transpose()
layer_conv_lstm_1d()
layer_conv_lstm_2d()

layer_conv_lstm_3d()
layer_cropping_1d()
layer_cropping_2d()
layer_cropping_3d()
layer_dense()
layer_depthwise_conv_1d()
layer_depthwise_conv_2d()
layer_discretization()
layer_dot()
layer_dropout()
layer_einsum_dense()
layer_embedding()
layer_feature_space()
layer_flatten()
layer_flax_module_wrapper()
layer_gaussian_dropout()
layer_gaussian_noise()
layer_global_average_pooling_1d()
layer_global_average_pooling_2d()
layer_global_average_pooling_3d()
layer_global_max_pooling_1d()
layer_global_max_pooling_2d()
layer_global_max_pooling_3d()
layer_group_normalization()
layer_group_query_attention()
layer_gru()
layer_hashed_crossing()
layer_hashing()
layer_identity()
layer_integer_lookup()
layer_jax_model_wrapper()
layer_lambda()
layer_layer_normalization()
layer_lstm()
layer_masking()
layer_max_pooling_1d()
layer_max_pooling_2d()
layer_max_pooling_3d()
layer_maximum()
layer_mel_spectrogram()
layer_minimum()
layer_multi_head_attention()
layer_multiply()
layer_permute()
layer_random_brightness()
layer_random_contrast()
layer_random_crop()
layer_random_flip()

layer_random_rotation()
layer_random_translation()
layer_random_zoom()
layer_repeat_vector()
layer_rescaling()
layer_reshape()
layer_resizing()
layer_rnn()
layer_separable_conv_1d()
layer_separable_conv_2d()
layer_simple_rnn()
layer_spatial_dropout_1d()
layer_spatial_dropout_2d()
layer_spatial_dropout_3d()
layer_spectral_normalization()
layer_string_lookup()
layer_subtract()
layer_text_vectorization()
layer_tfsm()
layer_time_distributed()
layer_torch_module_wrapper()
layer_unit_normalization()
layer_upsampling_1d()
layer_upsampling_2d()
layer_upsampling_3d()
layer_zero_padding_1d()
layer_zero_padding_2d()
layer_zero_padding_3d()
rnn_cell_gru()
rnn_cell_lstm()
rnn_cell_simple()
rnn_cells_stack()

`layer_permute` *Permutes the dimensions of the input according to a given pattern.*

Description

Useful e.g. connecting RNNs and convnets.

Usage

```
layer_permute(object, dims, ...)
```

Arguments

`object`	Object to compose the layer with. A tensor, array, or sequential model.
`dims`	List of integers. Permutation pattern does not include the batch dimension. Indexing starts at 1. For instance, `c(2, 1)` permutes the first and second dimensions of the input.
`...`	For forward/backward compatability.

Value

The return value depends on the value provided for the first argument. If `object` is:

- a `keras_model_sequential()`, then the layer is added to the sequential model (which is modified in place). To enable piping, the sequential model is also returned, invisibly.
- a `keras_input()`, then the output tensor from calling `layer(input)` is returned.
- `NULL` or missing, then a `Layer` instance is returned.

Input Shape

Arbitrary.

Output Shape

Same as the input shape, but with the dimensions re-ordered according to the specified pattern.

Example

```
x <- layer_input(shape=c(10, 64))
y <- layer_permute(x, c(2, 1))
shape(y)

## shape(NA, 64, 10)
```

See Also

- `https://keras.io/api/layers/reshaping_layers/permute#permute-class`

Other reshaping layers:
`layer_cropping_1d()`
`layer_cropping_2d()`
`layer_cropping_3d()`
`layer_flatten()`
`layer_repeat_vector()`
`layer_reshape()`
`layer_upsampling_1d()`
`layer_upsampling_2d()`
`layer_upsampling_3d()`
`layer_zero_padding_1d()`
`layer_zero_padding_2d()`

layer_zero_padding_3d()

Other layers:
Layer()
layer_activation()
layer_activation_elu()
layer_activation_leaky_relu()
layer_activation_parametric_relu()
layer_activation_relu()
layer_activation_softmax()
layer_activity_regularization()
layer_add()
layer_additive_attention()
layer_alpha_dropout()
layer_attention()
layer_average()
layer_average_pooling_1d()
layer_average_pooling_2d()
layer_average_pooling_3d()
layer_batch_normalization()
layer_bidirectional()
layer_category_encoding()
layer_center_crop()
layer_concatenate()
layer_conv_1d()
layer_conv_1d_transpose()
layer_conv_2d()
layer_conv_2d_transpose()
layer_conv_3d()
layer_conv_3d_transpose()
layer_conv_lstm_1d()
layer_conv_lstm_2d()
layer_conv_lstm_3d()
layer_cropping_1d()
layer_cropping_2d()
layer_cropping_3d()
layer_dense()
layer_depthwise_conv_1d()
layer_depthwise_conv_2d()
layer_discretization()
layer_dot()
layer_dropout()
layer_einsum_dense()
layer_embedding()
layer_feature_space()
layer_flatten()
layer_flax_module_wrapper()
layer_gaussian_dropout()

layer_gaussian_noise()
layer_global_average_pooling_1d()
layer_global_average_pooling_2d()
layer_global_average_pooling_3d()
layer_global_max_pooling_1d()
layer_global_max_pooling_2d()
layer_global_max_pooling_3d()
layer_group_normalization()
layer_group_query_attention()
layer_gru()
layer_hashed_crossing()
layer_hashing()
layer_identity()
layer_integer_lookup()
layer_jax_model_wrapper()
layer_lambda()
layer_layer_normalization()
layer_lstm()
layer_masking()
layer_max_pooling_1d()
layer_max_pooling_2d()
layer_max_pooling_3d()
layer_maximum()
layer_mel_spectrogram()
layer_minimum()
layer_multi_head_attention()
layer_multiply()
layer_normalization()
layer_random_brightness()
layer_random_contrast()
layer_random_crop()
layer_random_flip()
layer_random_rotation()
layer_random_translation()
layer_random_zoom()
layer_repeat_vector()
layer_rescaling()
layer_reshape()
layer_resizing()
layer_rnn()
layer_separable_conv_1d()
layer_separable_conv_2d()
layer_simple_rnn()
layer_spatial_dropout_1d()
layer_spatial_dropout_2d()
layer_spatial_dropout_3d()
layer_spectral_normalization()
layer_string_lookup()

layer_subtract()
layer_text_vectorization()
layer_tfsm()
layer_time_distributed()
layer_torch_module_wrapper()
layer_unit_normalization()
layer_upsampling_1d()
layer_upsampling_2d()
layer_upsampling_3d()
layer_zero_padding_1d()
layer_zero_padding_2d()
layer_zero_padding_3d()
rnn_cell_gru()
rnn_cell_lstm()
rnn_cell_simple()
rnn_cells_stack()

layer_random_brightness

A preprocessing layer which randomly adjusts brightness during training.

Description

This layer will randomly increase/reduce the brightness for the input RGB images. At inference time, the output will be identical to the input. Call the layer with `training=TRUE` to adjust the brightness of the input.

Note: This layer is safe to use inside a `tf.data` pipeline (independently of which backend you're using).

Usage

```
layer_random_brightness(
  object,
  factor,
  value_range = list(0L, 255L),
  seed = NULL,
  ...
)
```

Arguments

object	Object to compose the layer with. A tensor, array, or sequential model.

`factor`	Float or a list of 2 floats between -1.0 and 1.0. The factor is used to determine the lower bound and upper bound of the brightness adjustment. A float value will be chosen randomly between the limits. When -1.0 is chosen, the output image will be black, and when 1.0 is chosen, the image will be fully white. When only one float is provided, eg, 0.2, then -0.2 will be used for lower bound and 0.2 will be used for upper bound.
`value_range`	Optional list of 2 floats for the lower and upper limit of the values of the input data. To make no change, use `c(0.0, 1.0)`, e.g., if the image input has been scaled before this layer. Defaults to `c(0.0, 255.0)`. The brightness adjustment will be scaled to this range, and the output values will be clipped to this range.
`seed`	optional integer, for fixed RNG behavior.
`...`	For forward/backward compatability.

Value

The return value depends on the value provided for the first argument. If `object` is:

- a `keras_model_sequential()`, then the layer is added to the sequential model (which is modified in place). To enable piping, the sequential model is also returned, invisibly.
- a `keras_input()`, then the output tensor from calling `layer(input)` is returned.
- `NULL` or missing, then a `Layer` instance is returned.

Inputs

3D (HWC) or 4D (NHWC) tensor, with float or int dtype. Input pixel values can be of any range (e.g. `[0., 1.)` or `[0, 255]`)

Output

3D (HWC) or 4D (NHWC) tensor with brightness adjusted based on the `factor`. By default, the layer will output floats. The output value will be clipped to the range `[0, 255]`, the valid range of RGB colors, and rescaled based on the `value_range` if needed.

Example

```
random_bright <- layer_random_brightness(factor=0.2, seed = 1)

# An image with shape [2, 2, 3]
image <- array(1:12, dim=c(2, 2, 3))

# Assume we randomly select the factor to be 0.1, then it will apply
# 0.1 * 255 to all the channel
output <- random_bright(image, training=TRUE)
output

## tf.Tensor(
## [[[39.605797 43.605797 47.605797]
##   [41.605797 45.605797 49.605797]]
##
## [[40.605797 44.605797 48.605797]
##   [42.605797 46.605797 50.605797]]], shape=(2, 2, 3), dtype=float32)
```

See Also

- https://keras.io/api/layers/preprocessing_layers/image_augmentation/random_brightness#randombrightness-class

Other image augmentation layers:
layer_random_contrast()
layer_random_crop()
layer_random_flip()
layer_random_rotation()
layer_random_translation()
layer_random_zoom()

Other preprocessing layers:
layer_category_encoding()
layer_center_crop()
layer_discretization()
layer_feature_space()
layer_hashed_crossing()
layer_hashing()
layer_integer_lookup()
layer_mel_spectrogram()
layer_normalization()
layer_random_contrast()
layer_random_crop()
layer_random_flip()
layer_random_rotation()
layer_random_translation()
layer_random_zoom()
layer_rescaling()
layer_resizing()
layer_string_lookup()
layer_text_vectorization()

Other layers:
Layer()
layer_activation()
layer_activation_elu()
layer_activation_leaky_relu()
layer_activation_parametric_relu()
layer_activation_relu()
layer_activation_softmax()
layer_activity_regularization()
layer_add()

layer_additive_attention()
layer_alpha_dropout()
layer_attention()
layer_average()
layer_average_pooling_1d()
layer_average_pooling_2d()
layer_average_pooling_3d()
layer_batch_normalization()
layer_bidirectional()
layer_category_encoding()
layer_center_crop()
layer_concatenate()
layer_conv_1d()
layer_conv_1d_transpose()
layer_conv_2d()
layer_conv_2d_transpose()
layer_conv_3d()
layer_conv_3d_transpose()
layer_conv_lstm_1d()
layer_conv_lstm_2d()
layer_conv_lstm_3d()
layer_cropping_1d()
layer_cropping_2d()
layer_cropping_3d()
layer_dense()
layer_depthwise_conv_1d()
layer_depthwise_conv_2d()
layer_discretization()
layer_dot()
layer_dropout()
layer_einsum_dense()
layer_embedding()
layer_feature_space()
layer_flatten()
layer_flax_module_wrapper()
layer_gaussian_dropout()
layer_gaussian_noise()
layer_global_average_pooling_1d()
layer_global_average_pooling_2d()
layer_global_average_pooling_3d()
layer_global_max_pooling_1d()
layer_global_max_pooling_2d()
layer_global_max_pooling_3d()
layer_group_normalization()
layer_group_query_attention()
layer_gru()
layer_hashed_crossing()
layer_hashing()

layer_identity()
layer_integer_lookup()
layer_jax_model_wrapper()
layer_lambda()
layer_layer_normalization()
layer_lstm()
layer_masking()
layer_max_pooling_1d()
layer_max_pooling_2d()
layer_max_pooling_3d()
layer_maximum()
layer_mel_spectrogram()
layer_minimum()
layer_multi_head_attention()
layer_multiply()
layer_normalization()
layer_permute()
layer_random_contrast()
layer_random_crop()
layer_random_flip()
layer_random_rotation()
layer_random_translation()
layer_random_zoom()
layer_repeat_vector()
layer_rescaling()
layer_reshape()
layer_resizing()
layer_rnn()
layer_separable_conv_1d()
layer_separable_conv_2d()
layer_simple_rnn()
layer_spatial_dropout_1d()
layer_spatial_dropout_2d()
layer_spatial_dropout_3d()
layer_spectral_normalization()
layer_string_lookup()
layer_subtract()
layer_text_vectorization()
layer_tfsm()
layer_time_distributed()
layer_torch_module_wrapper()
layer_unit_normalization()
layer_upsampling_1d()
layer_upsampling_2d()
layer_upsampling_3d()
layer_zero_padding_1d()
layer_zero_padding_2d()
layer_zero_padding_3d()

rnn_cell_gru()
rnn_cell_lstm()
rnn_cell_simple()
rnn_cells_stack()

`layer_random_contrast` *A preprocessing layer which randomly adjusts contrast during training.*

Description

This layer will randomly adjust the contrast of an image or images by a random factor. Contrast is adjusted independently for each channel of each image during training.

For each channel, this layer computes the mean of the image pixels in the channel and then adjusts each component `x` of each pixel to `(x - mean) * contrast_factor + mean`.

Input pixel values can be of any range (e.g. `[0., 1.)` or `[0, 255]`) and in integer or floating point dtype. By default, the layer will output floats.

Note: This layer is safe to use inside a `tf.data` pipeline (independently of which backend you're using).

Usage

```
layer_random_contrast(object, factor, seed = NULL, ...)
```

Arguments

`object`	Object to compose the layer with. A tensor, array, or sequential model.
`factor`	a positive float represented as fraction of value, or a tuple of size 2 representing lower and upper bound. When represented as a single float, lower = upper. The contrast factor will be randomly picked between `[1.0 - lower, 1.0 + upper]`. For any pixel x in the channel, the output will be `(x - mean) * factor + mean` where `mean` is the mean value of the channel.
`seed`	Integer. Used to create a random seed.
`...`	For forward/backward compatability.

Value

The return value depends on the value provided for the first argument. If `object` is:

- a `keras_model_sequential()`, then the layer is added to the sequential model (which is modified in place). To enable piping, the sequential model is also returned, invisibly.
- a `keras_input()`, then the output tensor from calling `layer(input)` is returned.
- `NULL` or missing, then a `Layer` instance is returned.

Input Shape

3D (unbatched) or 4D (batched) tensor with shape: `(..., height, width, channels)`, in `"channels_last"` format.

Output Shape

3D (unbatched) or 4D (batched) tensor with shape: `(..., height, width, channels)`, in `"channels_last"` format.

See Also

- `https://keras.io/api/layers/preprocessing_layers/image_augmentation/random_contrast#randomcontrast-class`

Other image augmentation layers:
`layer_random_brightness()`
`layer_random_crop()`
`layer_random_flip()`
`layer_random_rotation()`
`layer_random_translation()`
`layer_random_zoom()`

Other preprocessing layers:
`layer_category_encoding()`
`layer_center_crop()`
`layer_discretization()`
`layer_feature_space()`
`layer_hashed_crossing()`
`layer_hashing()`
`layer_integer_lookup()`
`layer_mel_spectrogram()`
`layer_normalization()`
`layer_random_brightness()`
`layer_random_crop()`
`layer_random_flip()`
`layer_random_rotation()`
`layer_random_translation()`
`layer_random_zoom()`
`layer_rescaling()`
`layer_resizing()`
`layer_string_lookup()`
`layer_text_vectorization()`

Other layers:
`Layer()`
`layer_activation()`
`layer_activation_elu()`
`layer_activation_leaky_relu()`

layer_activation_parametric_relu()
layer_activation_relu()
layer_activation_softmax()
layer_activity_regularization()
layer_add()
layer_additive_attention()
layer_alpha_dropout()
layer_attention()
layer_average()
layer_average_pooling_1d()
layer_average_pooling_2d()
layer_average_pooling_3d()
layer_batch_normalization()
layer_bidirectional()
layer_category_encoding()
layer_center_crop()
layer_concatenate()
layer_conv_1d()
layer_conv_1d_transpose()
layer_conv_2d()
layer_conv_2d_transpose()
layer_conv_3d()
layer_conv_3d_transpose()
layer_conv_lstm_1d()
layer_conv_lstm_2d()
layer_conv_lstm_3d()
layer_cropping_1d()
layer_cropping_2d()
layer_cropping_3d()
layer_dense()
layer_depthwise_conv_1d()
layer_depthwise_conv_2d()
layer_discretization()
layer_dot()
layer_dropout()
layer_einsum_dense()
layer_embedding()
layer_feature_space()
layer_flatten()
layer_flax_module_wrapper()
layer_gaussian_dropout()
layer_gaussian_noise()
layer_global_average_pooling_1d()
layer_global_average_pooling_2d()
layer_global_average_pooling_3d()
layer_global_max_pooling_1d()
layer_global_max_pooling_2d()
layer_global_max_pooling_3d()

layer_group_normalization()
layer_group_query_attention()
layer_gru()
layer_hashed_crossing()
layer_hashing()
layer_identity()
layer_integer_lookup()
layer_jax_model_wrapper()
layer_lambda()
layer_layer_normalization()
layer_lstm()
layer_masking()
layer_max_pooling_1d()
layer_max_pooling_2d()
layer_max_pooling_3d()
layer_maximum()
layer_mel_spectrogram()
layer_minimum()
layer_multi_head_attention()
layer_multiply()
layer_normalization()
layer_permute()
layer_random_brightness()
layer_random_crop()
layer_random_flip()
layer_random_rotation()
layer_random_translation()
layer_random_zoom()
layer_repeat_vector()
layer_rescaling()
layer_reshape()
layer_resizing()
layer_rnn()
layer_separable_conv_1d()
layer_separable_conv_2d()
layer_simple_rnn()
layer_spatial_dropout_1d()
layer_spatial_dropout_2d()
layer_spatial_dropout_3d()
layer_spectral_normalization()
layer_string_lookup()
layer_subtract()
layer_text_vectorization()
layer_tfsm()
layer_time_distributed()
layer_torch_module_wrapper()
layer_unit_normalization()
layer_upsampling_1d()

layer_upsampling_2d()
layer_upsampling_3d()
layer_zero_padding_1d()
layer_zero_padding_2d()
layer_zero_padding_3d()
rnn_cell_gru()
rnn_cell_lstm()
rnn_cell_simple()
rnn_cells_stack()

---

layer_random_crop *A preprocessing layer which randomly crops images during training.*

Description

During training, this layer will randomly choose a location to crop images down to a target size. The layer will crop all the images in the same batch to the same cropping location.

At inference time, and during training if an input image is smaller than the target size, the input will be resized and cropped so as to return the largest possible window in the image that matches the target aspect ratio. If you need to apply random cropping at inference time, set `training` to TRUE when calling the layer.

Input pixel values can be of any range (e.g. `[0., 1.)` or `[0, 255]`) and of integer or floating point dtype. By default, the layer will output floats.

Note: This layer is safe to use inside a `tf.data` pipeline (independently of which backend you're using).

Usage

```
layer_random_crop(
  object,
  height,
  width,
  seed = NULL,
  data_format = NULL,
  name = NULL,
  ...
)
```

Arguments

`object`	Object to compose the layer with. A tensor, array, or sequential model.
`height`	Integer, the height of the output shape.
`width`	Integer, the width of the output shape.
`seed`	Integer. Used to create a random seed.

`data_format`	see description
`name`	String, name for the object
`...`	Base layer keyword arguments, such as `name` and `dtype`.

Value

The return value depends on the value provided for the first argument. If `object` is:

- a `keras_model_sequential()`, then the layer is added to the sequential model (which is modified in place). To enable piping, the sequential model is also returned, invisibly.
- a `keras_input()`, then the output tensor from calling `layer(input)` is returned.
- `NULL` or missing, then a `Layer` instance is returned.

Input Shape

3D (unbatched) or 4D (batched) tensor with shape: `(..., height, width, channels)`, in `"channels_last"` format.

Output Shape

3D (unbatched) or 4D (batched) tensor with shape: `(..., target_height, target_width, channels)`.

See Also

- `https://keras.io/api/layers/preprocessing_layers/image_augmentation/random_crop#randomcrop-class`

Other image augmentation layers:
`layer_random_brightness()`
`layer_random_contrast()`
`layer_random_flip()`
`layer_random_rotation()`
`layer_random_translation()`
`layer_random_zoom()`

Other preprocessing layers:
`layer_category_encoding()`
`layer_center_crop()`
`layer_discretization()`
`layer_feature_space()`
`layer_hashed_crossing()`
`layer_hashing()`
`layer_integer_lookup()`
`layer_mel_spectrogram()`
`layer_normalization()`
`layer_random_brightness()`
`layer_random_contrast()`
`layer_random_flip()`
`layer_random_rotation()`

layer_random_translation()
layer_random_zoom()
layer_rescaling()
layer_resizing()
layer_string_lookup()
layer_text_vectorization()

Other layers:
Layer()
layer_activation()
layer_activation_elu()
layer_activation_leaky_relu()
layer_activation_parametric_relu()
layer_activation_relu()
layer_activation_softmax()
layer_activity_regularization()
layer_add()
layer_additive_attention()
layer_alpha_dropout()
layer_attention()
layer_average()
layer_average_pooling_1d()
layer_average_pooling_2d()
layer_average_pooling_3d()
layer_batch_normalization()
layer_bidirectional()
layer_category_encoding()
layer_center_crop()
layer_concatenate()
layer_conv_1d()
layer_conv_1d_transpose()
layer_conv_2d()
layer_conv_2d_transpose()
layer_conv_3d()
layer_conv_3d_transpose()
layer_conv_lstm_1d()
layer_conv_lstm_2d()
layer_conv_lstm_3d()
layer_cropping_1d()
layer_cropping_2d()
layer_cropping_3d()
layer_dense()
layer_depthwise_conv_1d()
layer_depthwise_conv_2d()
layer_discretization()
layer_dot()
layer_dropout()
layer_einsum_dense()

layer_embedding()
layer_feature_space()
layer_flatten()
layer_flax_module_wrapper()
layer_gaussian_dropout()
layer_gaussian_noise()
layer_global_average_pooling_1d()
layer_global_average_pooling_2d()
layer_global_average_pooling_3d()
layer_global_max_pooling_1d()
layer_global_max_pooling_2d()
layer_global_max_pooling_3d()
layer_group_normalization()
layer_group_query_attention()
layer_gru()
layer_hashed_crossing()
layer_hashing()
layer_identity()
layer_integer_lookup()
layer_jax_model_wrapper()
layer_lambda()
layer_layer_normalization()
layer_lstm()
layer_masking()
layer_max_pooling_1d()
layer_max_pooling_2d()
layer_max_pooling_3d()
layer_maximum()
layer_mel_spectrogram()
layer_minimum()
layer_multi_head_attention()
layer_multiply()
layer_normalization()
layer_permute()
layer_random_brightness()
layer_random_contrast()
layer_random_flip()
layer_random_rotation()
layer_random_translation()
layer_random_zoom()
layer_repeat_vector()
layer_rescaling()
layer_reshape()
layer_resizing()
layer_rnn()
layer_separable_conv_1d()
layer_separable_conv_2d()
layer_simple_rnn()

layer_spatial_dropout_1d()
layer_spatial_dropout_2d()
layer_spatial_dropout_3d()
layer_spectral_normalization()
layer_string_lookup()
layer_subtract()
layer_text_vectorization()
layer_tfsm()
layer_time_distributed()
layer_torch_module_wrapper()
layer_unit_normalization()
layer_upsampling_1d()
layer_upsampling_2d()
layer_upsampling_3d()
layer_zero_padding_1d()
layer_zero_padding_2d()
layer_zero_padding_3d()
rnn_cell_gru()
rnn_cell_lstm()
rnn_cell_simple()
rnn_cells_stack()

---

layer_random_flip *A preprocessing layer which randomly flips images during training.*

Description

This layer will flip the images horizontally and or vertically based on the mode attribute. During inference time, the output will be identical to input. Call the layer with training=TRUE to flip the input. Input pixel values can be of any range (e.g. [0., 1.) or [0, 255]) and of integer or floating point dtype. By default, the layer will output floats.

Note: This layer is safe to use inside a tf.data pipeline (independently of which backend you're using).

Usage

```
layer_random_flip(object, mode = "horizontal_and_vertical", seed = NULL, ...)
```

Arguments

object	Object to compose the layer with. A tensor, array, or sequential model.
mode	String indicating which flip mode to use. Can be "horizontal", "vertical", or "horizontal_and_vertical". "horizontal" is a left-right flip and "vertical" is a top-bottom flip. Defaults to "horizontal_and_vertical"
seed	Integer. Used to create a random seed.
...	Base layer keyword arguments, such as name and dtype.

Value

The return value depends on the value provided for the first argument. If `object` is:

- a `keras_model_sequential()`, then the layer is added to the sequential model (which is modified in place). To enable piping, the sequential model is also returned, invisibly.
- a `keras_input()`, then the output tensor from calling `layer(input)` is returned.
- `NULL` or missing, then a `Layer` instance is returned.

Input Shape

3D (unbatched) or 4D (batched) tensor with shape: `(..., height, width, channels)`, in `"channels_last"` format.

Output Shape

3D (unbatched) or 4D (batched) tensor with shape: `(..., height, width, channels)`, in `"channels_last"` format.

See Also

- `https://keras.io/api/layers/preprocessing_layers/image_augmentation/random_flip#randomflip-class`

Other image augmentation layers:
`layer_random_brightness()`
`layer_random_contrast()`
`layer_random_crop()`
`layer_random_rotation()`
`layer_random_translation()`
`layer_random_zoom()`

Other preprocessing layers:
`layer_category_encoding()`
`layer_center_crop()`
`layer_discretization()`
`layer_feature_space()`
`layer_hashed_crossing()`
`layer_hashing()`
`layer_integer_lookup()`
`layer_mel_spectrogram()`
`layer_normalization()`
`layer_random_brightness()`
`layer_random_contrast()`
`layer_random_crop()`
`layer_random_rotation()`
`layer_random_translation()`
`layer_random_zoom()`
`layer_rescaling()`
`layer_resizing()`

layer_string_lookup()
layer_text_vectorization()

Other layers:
Layer()
layer_activation()
layer_activation_elu()
layer_activation_leaky_relu()
layer_activation_parametric_relu()
layer_activation_relu()
layer_activation_softmax()
layer_activity_regularization()
layer_add()
layer_additive_attention()
layer_alpha_dropout()
layer_attention()
layer_average()
layer_average_pooling_1d()
layer_average_pooling_2d()
layer_average_pooling_3d()
layer_batch_normalization()
layer_bidirectional()
layer_category_encoding()
layer_center_crop()
layer_concatenate()
layer_conv_1d()
layer_conv_1d_transpose()
layer_conv_2d()
layer_conv_2d_transpose()
layer_conv_3d()
layer_conv_3d_transpose()
layer_conv_lstm_1d()
layer_conv_lstm_2d()
layer_conv_lstm_3d()
layer_cropping_1d()
layer_cropping_2d()
layer_cropping_3d()
layer_dense()
layer_depthwise_conv_1d()
layer_depthwise_conv_2d()
layer_discretization()
layer_dot()
layer_dropout()
layer_einsum_dense()
layer_embedding()
layer_feature_space()
layer_flatten()
layer_flax_module_wrapper()

layer_gaussian_dropout()
layer_gaussian_noise()
layer_global_average_pooling_1d()
layer_global_average_pooling_2d()
layer_global_average_pooling_3d()
layer_global_max_pooling_1d()
layer_global_max_pooling_2d()
layer_global_max_pooling_3d()
layer_group_normalization()
layer_group_query_attention()
layer_gru()
layer_hashed_crossing()
layer_hashing()
layer_identity()
layer_integer_lookup()
layer_jax_model_wrapper()
layer_lambda()
layer_layer_normalization()
layer_lstm()
layer_masking()
layer_max_pooling_1d()
layer_max_pooling_2d()
layer_max_pooling_3d()
layer_maximum()
layer_mel_spectrogram()
layer_minimum()
layer_multi_head_attention()
layer_multiply()
layer_normalization()
layer_permute()
layer_random_brightness()
layer_random_contrast()
layer_random_crop()
layer_random_rotation()
layer_random_translation()
layer_random_zoom()
layer_repeat_vector()
layer_rescaling()
layer_reshape()
layer_resizing()
layer_rnn()
layer_separable_conv_1d()
layer_separable_conv_2d()
layer_simple_rnn()
layer_spatial_dropout_1d()
layer_spatial_dropout_2d()
layer_spatial_dropout_3d()
layer_spectral_normalization()

layer_string_lookup()
layer_subtract()
layer_text_vectorization()
layer_tfsm()
layer_time_distributed()
layer_torch_module_wrapper()
layer_unit_normalization()
layer_upsampling_1d()
layer_upsampling_2d()
layer_upsampling_3d()
layer_zero_padding_1d()
layer_zero_padding_2d()
layer_zero_padding_3d()
rnn_cell_gru()
rnn_cell_lstm()
rnn_cell_simple()
rnn_cells_stack()

layer_random_rotation *A preprocessing layer which randomly rotates images during training.*

Description

This layer will apply random rotations to each image, filling empty space according to `fill_mode`.

By default, random rotations are only applied during training. At inference time, the layer does nothing. If you need to apply random rotations at inference time, pass `training = TRUE` when calling the layer.

Input pixel values can be of any range (e.g. `[0., 1.)` or `[0, 255]`) and of integer or floating point dtype. By default, the layer will output floats.

Note: This layer is safe to use inside a `tf.data` pipeline (independently of which backend you're using).

Usage

```
layer_random_rotation(
  object,
  factor,
  fill_mode = "reflect",
  interpolation = "bilinear",
  seed = NULL,
  fill_value = 0,
  value_range = list(0L, 255L),
  data_format = NULL,
  ...
)
```

Arguments

`object`	Object to compose the layer with. A tensor, array, or sequential model.
`factor`	a float represented as fraction of 2 Pi, or a tuple of size 2 representing lower and upper bound for rotating clockwise and counter-clockwise. A positive values means rotating counter clock-wise, while a negative value means clock-wise. When represented as a single float, this value is used for both the upper and lower bound. For instance, `factor=(-0.2, 0.3)` results in an output rotation by a random amount in the range `[-20% * 2pi, 30% * 2pi]`. `factor=0.2` results in an output rotating by a random amount in the range `[-20% * 2pi, 20% * 2pi]`.
`fill_mode`	Points outside the boundaries of the input are filled according to the given mode (one of `{"constant", "reflect", "wrap", "nearest"}`). • *reflect*: `(d c b a | a b c d | d c b a)` The input is extended by reflecting about the edge of the last pixel. • *constant*: `(k k k k | a b c d | k k k k)` The input is extended by filling all values beyond the edge with the same constant value k = 0. • *wrap*: `(a b c d | a b c d | a b c d)` The input is extended by wrapping around to the opposite edge. • *nearest*: `(a a a a | a b c d | d d d d)` The input is extended by the nearest pixel.
`interpolation`	Interpolation mode. Supported values: `"nearest"`, `"bilinear"`.
`seed`	Integer. Used to create a random seed.
`fill_value`	a float represents the value to be filled outside the boundaries when `fill_mode="constant"`.
`value_range`	see description
`data_format`	see description
`...`	For forward/backward compatability.

Value

The return value depends on the value provided for the first argument. If `object` is:

- a `keras_model_sequential()`, then the layer is added to the sequential model (which is modified in place). To enable piping, the sequential model is also returned, invisibly.
- a `keras_input()`, then the output tensor from calling `layer(input)` is returned.
- `NULL` or missing, then a `Layer` instance is returned.

Input Shape

3D (unbatched) or 4D (batched) tensor with shape: `(..., height, width, channels)`, in `"channels_last"` format

Output Shape

3D (unbatched) or 4D (batched) tensor with shape: `(..., height, width, channels)`, in `"channels_last"` format

See Also

- https://keras.io/api/layers/preprocessing_layers/image_augmentation/random_rotation#randomrotation-class

Other image augmentation layers:
`layer_random_brightness()`
`layer_random_contrast()`
`layer_random_crop()`
`layer_random_flip()`
`layer_random_translation()`
`layer_random_zoom()`

Other preprocessing layers:
`layer_category_encoding()`
`layer_center_crop()`
`layer_discretization()`
`layer_feature_space()`
`layer_hashed_crossing()`
`layer_hashing()`
`layer_integer_lookup()`
`layer_mel_spectrogram()`
`layer_normalization()`
`layer_random_brightness()`
`layer_random_contrast()`
`layer_random_crop()`
`layer_random_flip()`
`layer_random_translation()`
`layer_random_zoom()`
`layer_rescaling()`
`layer_resizing()`
`layer_string_lookup()`
`layer_text_vectorization()`

Other layers:
`Layer()`
`layer_activation()`
`layer_activation_elu()`
`layer_activation_leaky_relu()`
`layer_activation_parametric_relu()`
`layer_activation_relu()`
`layer_activation_softmax()`
`layer_activity_regularization()`
`layer_add()`
`layer_additive_attention()`
`layer_alpha_dropout()`
`layer_attention()`
`layer_average()`
`layer_average_pooling_1d()`

layer_average_pooling_2d()
layer_average_pooling_3d()
layer_batch_normalization()
layer_bidirectional()
layer_category_encoding()
layer_center_crop()
layer_concatenate()
layer_conv_1d()
layer_conv_1d_transpose()
layer_conv_2d()
layer_conv_2d_transpose()
layer_conv_3d()
layer_conv_3d_transpose()
layer_conv_lstm_1d()
layer_conv_lstm_2d()
layer_conv_lstm_3d()
layer_cropping_1d()
layer_cropping_2d()
layer_cropping_3d()
layer_dense()
layer_depthwise_conv_1d()
layer_depthwise_conv_2d()
layer_discretization()
layer_dot()
layer_dropout()
layer_einsum_dense()
layer_embedding()
layer_feature_space()
layer_flatten()
layer_flax_module_wrapper()
layer_gaussian_dropout()
layer_gaussian_noise()
layer_global_average_pooling_1d()
layer_global_average_pooling_2d()
layer_global_average_pooling_3d()
layer_global_max_pooling_1d()
layer_global_max_pooling_2d()
layer_global_max_pooling_3d()
layer_group_normalization()
layer_group_query_attention()
layer_gru()
layer_hashed_crossing()
layer_hashing()
layer_identity()
layer_integer_lookup()
layer_jax_model_wrapper()
layer_lambda()
layer_layer_normalization()

layer_lstm()
layer_masking()
layer_max_pooling_1d()
layer_max_pooling_2d()
layer_max_pooling_3d()
layer_maximum()
layer_mel_spectrogram()
layer_minimum()
layer_multi_head_attention()
layer_multiply()
layer_normalization()
layer_permute()
layer_random_brightness()
layer_random_contrast()
layer_random_crop()
layer_random_flip()
layer_random_translation()
layer_random_zoom()
layer_repeat_vector()
layer_rescaling()
layer_reshape()
layer_resizing()
layer_rnn()
layer_separable_conv_1d()
layer_separable_conv_2d()
layer_simple_rnn()
layer_spatial_dropout_1d()
layer_spatial_dropout_2d()
layer_spatial_dropout_3d()
layer_spectral_normalization()
layer_string_lookup()
layer_subtract()
layer_text_vectorization()
layer_tfsm()
layer_time_distributed()
layer_torch_module_wrapper()
layer_unit_normalization()
layer_upsampling_1d()
layer_upsampling_2d()
layer_upsampling_3d()
layer_zero_padding_1d()
layer_zero_padding_2d()
layer_zero_padding_3d()
rnn_cell_gru()
rnn_cell_lstm()
rnn_cell_simple()
rnn_cells_stack()

`layer_random_translation` *A preprocessing layer which randomly translates images during training.*

Description

This layer will apply random translations to each image during training, filling empty space according to `fill_mode`.

Input pixel values can be of any range (e.g. `[0., 1.)` or `[0, 255]`) and of integer or floating point dtype. By default, the layer will output floats.

Usage

```
layer_random_translation(
  object,
  height_factor,
  width_factor,
  fill_mode = "reflect",
  interpolation = "bilinear",
  seed = NULL,
  fill_value = 0,
  data_format = NULL,
  ...
)
```

Arguments

`object`	Object to compose the layer with. A tensor, array, or sequential model.
`height_factor`	a float represented as fraction of value, or a tuple of size 2 representing lower and upper bound for shifting vertically. A negative value means shifting image up, while a positive value means shifting image down. When represented as a single positive float, this value is used for both the upper and lower bound. For instance, `height_factor=(-0.2, 0.3)` results in an output shifted by a random amount in the range `[-20%, +30%]`. `height_factor=0.2` results in an output height shifted by a random amount in the range `[-20%, +20%]`.
`width_factor`	a float represented as fraction of value, or a tuple of size 2 representing lower and upper bound for shifting horizontally. A negative value means shifting image left, while a positive value means shifting image right. When represented as a single positive float, this value is used for both the upper and lower bound. For instance, `width_factor=(-0.2, 0.3)` results in an output shifted left by 20%, and shifted right by 30%. `width_factor=0.2` results in an output height shifted left or right by 20%.
`fill_mode`	Points outside the boundaries of the input are filled according to the given mode. Available methods are `"constant"`, `"nearest"`, `"wrap"` and `"reflect"`. Defaults to `"constant"`.

- `"reflect"`: `(d c b a | a b c d | d c b a)` The input is extended by reflecting about the edge of the last pixel.
- `"constant"`: `(k k k k | a b c d | k k k k)` The input is extended by filling all values beyond the edge with the same constant value k specified by `fill_value`.
- `"wrap"`: `(a b c d | a b c d | a b c d)` The input is extended by wrapping around to the opposite edge.
- `"nearest"`: `(a a a a | a b c d | d d d d)` The input is extended by the nearest pixel. Note that when using torch backend, `"reflect"` is redirected to `"mirror"` `(c d c b | a b c d | c b a b)` because torch does not support `"reflect"`. Note that torch backend does not support `"wrap"`.

`interpolation`	Interpolation mode. Supported values: `"nearest"`, `"bilinear"`.
`seed`	Integer. Used to create a random seed.
`fill_value`	a float represents the value to be filled outside the boundaries when `fill_mode="constant"`.
`data_format`	string, either `"channels_last"` or `"channels_first"`. The ordering of the dimensions in the inputs. `"channels_last"` corresponds to inputs with shape `(batch, height, width, channels)` while `"channels_first"` corresponds to inputs with shape `(batch, channels, height, width)`. It defaults to the `image_data_format` value found in your Keras config file at `~/.keras/keras.json`. If you never set it, then it will be `"channels_last"`.
`...`	Base layer keyword arguments, such as `name` and `dtype`.

Value

The return value depends on the value provided for the first argument. If `object` is:

- a `keras_model_sequential()`, then the layer is added to the sequential model (which is modified in place). To enable piping, the sequential model is also returned, invisibly.
- a `keras_input()`, then the output tensor from calling `layer(input)` is returned.
- `NULL` or missing, then a `Layer` instance is returned.

Input Shape

3D (unbatched) or 4D (batched) tensor with shape: `(..., height, width, channels)`, in `"channels_last"` format, or `(..., channels, height, width)`, in `"channels_first"` format.

Output Shape

3D (unbatched) or 4D (batched) tensor with shape: `(..., target_height, target_width, channels)`, or `(..., channels, target_height, target_width)`, in `"channels_first"` format.

Note: This layer is safe to use inside a `tf.data` pipeline (independently of which backend you're using).

See Also

- https://keras.io/api/layers/preprocessing_layers/image_augmentation/random_translation#randomtranslation-class

Other image augmentation layers:
layer_random_brightness()
layer_random_contrast()
layer_random_crop()
layer_random_flip()
layer_random_rotation()
layer_random_zoom()

Other preprocessing layers:
layer_category_encoding()
layer_center_crop()
layer_discretization()
layer_feature_space()
layer_hashed_crossing()
layer_hashing()
layer_integer_lookup()
layer_mel_spectrogram()
layer_normalization()
layer_random_brightness()
layer_random_contrast()
layer_random_crop()
layer_random_flip()
layer_random_rotation()
layer_random_zoom()
layer_rescaling()
layer_resizing()
layer_string_lookup()
layer_text_vectorization()

Other layers:
Layer()
layer_activation()
layer_activation_elu()
layer_activation_leaky_relu()
layer_activation_parametric_relu()
layer_activation_relu()
layer_activation_softmax()
layer_activity_regularization()
layer_add()
layer_additive_attention()
layer_alpha_dropout()
layer_attention()
layer_average()
layer_average_pooling_1d()

layer_average_pooling_2d()
layer_average_pooling_3d()
layer_batch_normalization()
layer_bidirectional()
layer_category_encoding()
layer_center_crop()
layer_concatenate()
layer_conv_1d()
layer_conv_1d_transpose()
layer_conv_2d()
layer_conv_2d_transpose()
layer_conv_3d()
layer_conv_3d_transpose()
layer_conv_lstm_1d()
layer_conv_lstm_2d()
layer_conv_lstm_3d()
layer_cropping_1d()
layer_cropping_2d()
layer_cropping_3d()
layer_dense()
layer_depthwise_conv_1d()
layer_depthwise_conv_2d()
layer_discretization()
layer_dot()
layer_dropout()
layer_einsum_dense()
layer_embedding()
layer_feature_space()
layer_flatten()
layer_flax_module_wrapper()
layer_gaussian_dropout()
layer_gaussian_noise()
layer_global_average_pooling_1d()
layer_global_average_pooling_2d()
layer_global_average_pooling_3d()
layer_global_max_pooling_1d()
layer_global_max_pooling_2d()
layer_global_max_pooling_3d()
layer_group_normalization()
layer_group_query_attention()
layer_gru()
layer_hashed_crossing()
layer_hashing()
layer_identity()
layer_integer_lookup()
layer_jax_model_wrapper()
layer_lambda()
layer_layer_normalization()

layer_lstm()
layer_masking()
layer_max_pooling_1d()
layer_max_pooling_2d()
layer_max_pooling_3d()
layer_maximum()
layer_mel_spectrogram()
layer_minimum()
layer_multi_head_attention()
layer_multiply()
layer_normalization()
layer_permute()
layer_random_brightness()
layer_random_contrast()
layer_random_crop()
layer_random_flip()
layer_random_rotation()
layer_random_zoom()
layer_repeat_vector()
layer_rescaling()
layer_reshape()
layer_resizing()
layer_rnn()
layer_separable_conv_1d()
layer_separable_conv_2d()
layer_simple_rnn()
layer_spatial_dropout_1d()
layer_spatial_dropout_2d()
layer_spatial_dropout_3d()
layer_spectral_normalization()
layer_string_lookup()
layer_subtract()
layer_text_vectorization()
layer_tfsm()
layer_time_distributed()
layer_torch_module_wrapper()
layer_unit_normalization()
layer_upsampling_1d()
layer_upsampling_2d()
layer_upsampling_3d()
layer_zero_padding_1d()
layer_zero_padding_2d()
layer_zero_padding_3d()
rnn_cell_gru()
rnn_cell_lstm()
rnn_cell_simple()
rnn_cells_stack()

layer_random_zoom — *A preprocessing layer which randomly zooms images during training.*

Description

This layer will randomly zoom in or out on each axis of an image independently, filling empty space according to `fill_mode`.

Input pixel values can be of any range (e.g. `[0., 1.)` or `[0, 255]`) and of integer or floating point dtype. By default, the layer will output floats.

Usage

```
layer_random_zoom(
  object,
  height_factor,
  width_factor = NULL,
  fill_mode = "reflect",
  interpolation = "bilinear",
  seed = NULL,
  fill_value = 0,
  data_format = NULL,
  ...
)
```

Arguments

`object`	Object to compose the layer with. A tensor, array, or sequential model.
`height_factor`	a float represented as fraction of value, or a list of size 2 representing lower and upper bound for zooming vertically. When represented as a single float, this value is used for both the upper and lower bound. A positive value means zooming out, while a negative value means zooming in. For instance, `height_factor=c(0.2, 0.3)` result in an output zoomed out by a random amount in the range `[+20%, +30%]`. `height_factor=c(-0.3, -0.2)` result in an output zoomed in by a random amount in the range `[+20%, +30%]`.
`width_factor`	a float represented as fraction of value, or a list of size 2 representing lower and upper bound for zooming horizontally. When represented as a single float, this value is used for both the upper and lower bound. For instance, `width_factor=c(0.2, 0.3)` result in an output zooming out between 20% to 30%. `width_factor=c(-0.3, -0.2)` result in an output zooming in between 20% to 30%. `NULL` means i.e., zooming vertical and horizontal directions by preserving the aspect ratio. Defaults to `NULL`.
`fill_mode`	Points outside the boundaries of the input are filled according to the given mode. Available methods are `"constant"`, `"nearest"`, `"wrap"` and `"reflect"`. Defaults to `"constant"`.

- `"reflect"`: `(d c b a | a b c d | d c b a)` The input is extended by reflecting about the edge of the last pixel.

	• "constant": (k k k k | a b c d | k k k k) The input is extended by filling all values beyond the edge with the same constant value k specified by fill_value. • "wrap": (a b c d | a b c d | a b c d) The input is extended by wrapping around to the opposite edge. • "nearest": (a a a a | a b c d | d d d d) The input is extended by the nearest pixel. Note that when using torch backend, "reflect" is redirected to "mirror" (c d c b | a b c d | c b a b) because torch does not support "reflect". Note that torch backend does not support "wrap".
interpolation	Interpolation mode. Supported values: "nearest", "bilinear".
seed	Integer. Used to create a random seed.
fill_value	a float represents the value to be filled outside the boundaries when fill_mode="constant".
data_format	string, either "channels_last" or "channels_first". The ordering of the dimensions in the inputs. "channels_last" corresponds to inputs with shape (batch, height, width, channels) while "channels_first" corresponds to inputs with shape (batch, channels, height, width). It defaults to the image_data_format value found in your Keras config file at ~/.keras/keras.json. If you never set it, then it will be "channels_last".
...	Base layer keyword arguments, such as name and dtype.

Value

The return value depends on the value provided for the first argument. If object is:

- a keras_model_sequential(), then the layer is added to the sequential model (which is modified in place). To enable piping, the sequential model is also returned, invisibly.
- a keras_input(), then the output tensor from calling layer(input) is returned.
- NULL or missing, then a Layer instance is returned.

Input Shape

3D (unbatched) or 4D (batched) tensor with shape: (..., height, width, channels), in "channels_last" format, or (..., channels, height, width), in "channels_first" format.

Output Shape

3D (unbatched) or 4D (batched) tensor with shape: (..., target_height, target_width, channels), or (..., channels, target_height, target_width), in "channels_first" format.

Note: This layer is safe to use inside a tf.data pipeline (independently of which backend you're using).

Examples

```
input_img <- random_uniform(c(32, 224, 224, 3))
layer <- layer_random_zoom(height_factor = .5, width_factor = .2)
out_img <- layer(input_img)
```

See Also

- https://keras.io/api/layers/preprocessing_layers/image_augmentation/random_zoom#randomzoom-class

Other image augmentation layers:
layer_random_brightness()
layer_random_contrast()
layer_random_crop()
layer_random_flip()
layer_random_rotation()
layer_random_translation()

Other preprocessing layers:
layer_category_encoding()
layer_center_crop()
layer_discretization()
layer_feature_space()
layer_hashed_crossing()
layer_hashing()
layer_integer_lookup()
layer_mel_spectrogram()
layer_normalization()
layer_random_brightness()
layer_random_contrast()
layer_random_crop()
layer_random_flip()
layer_random_rotation()
layer_random_translation()
layer_rescaling()
layer_resizing()
layer_string_lookup()
layer_text_vectorization()

Other layers:
Layer()
layer_activation()
layer_activation_elu()
layer_activation_leaky_relu()
layer_activation_parametric_relu()
layer_activation_relu()
layer_activation_softmax()
layer_activity_regularization()
layer_add()
layer_additive_attention()
layer_alpha_dropout()
layer_attention()
layer_average()
layer_average_pooling_1d()

layer_average_pooling_2d()
layer_average_pooling_3d()
layer_batch_normalization()
layer_bidirectional()
layer_category_encoding()
layer_center_crop()
layer_concatenate()
layer_conv_1d()
layer_conv_1d_transpose()
layer_conv_2d()
layer_conv_2d_transpose()
layer_conv_3d()
layer_conv_3d_transpose()
layer_conv_lstm_1d()
layer_conv_lstm_2d()
layer_conv_lstm_3d()
layer_cropping_1d()
layer_cropping_2d()
layer_cropping_3d()
layer_dense()
layer_depthwise_conv_1d()
layer_depthwise_conv_2d()
layer_discretization()
layer_dot()
layer_dropout()
layer_einsum_dense()
layer_embedding()
layer_feature_space()
layer_flatten()
layer_flax_module_wrapper()
layer_gaussian_dropout()
layer_gaussian_noise()
layer_global_average_pooling_1d()
layer_global_average_pooling_2d()
layer_global_average_pooling_3d()
layer_global_max_pooling_1d()
layer_global_max_pooling_2d()
layer_global_max_pooling_3d()
layer_group_normalization()
layer_group_query_attention()
layer_gru()
layer_hashed_crossing()
layer_hashing()
layer_identity()
layer_integer_lookup()
layer_jax_model_wrapper()
layer_lambda()
layer_layer_normalization()

layer_lstm()
layer_masking()
layer_max_pooling_1d()
layer_max_pooling_2d()
layer_max_pooling_3d()
layer_maximum()
layer_mel_spectrogram()
layer_minimum()
layer_multi_head_attention()
layer_multiply()
layer_normalization()
layer_permute()
layer_random_brightness()
layer_random_contrast()
layer_random_crop()
layer_random_flip()
layer_random_rotation()
layer_random_translation()
layer_repeat_vector()
layer_rescaling()
layer_reshape()
layer_resizing()
layer_rnn()
layer_separable_conv_1d()
layer_separable_conv_2d()
layer_simple_rnn()
layer_spatial_dropout_1d()
layer_spatial_dropout_2d()
layer_spatial_dropout_3d()
layer_spectral_normalization()
layer_string_lookup()
layer_subtract()
layer_text_vectorization()
layer_tfsm()
layer_time_distributed()
layer_torch_module_wrapper()
layer_unit_normalization()
layer_upsampling_1d()
layer_upsampling_2d()
layer_upsampling_3d()
layer_zero_padding_1d()
layer_zero_padding_2d()
layer_zero_padding_3d()
rnn_cell_gru()
rnn_cell_lstm()
rnn_cell_simple()
rnn_cells_stack()

`layer_repeat_vector` *Repeats the input n times.*

Description

Repeats the input n times.

Usage

```
layer_repeat_vector(object, n, ...)
```

Arguments

`object`	Object to compose the layer with. A tensor, array, or sequential model.
`n`	Integer, repetition factor.
`...`	For forward/backward compatability.

Value

The return value depends on the value provided for the first argument. If `object` is:

- a `keras_model_sequential()`, then the layer is added to the sequential model (which is modified in place). To enable piping, the sequential model is also returned, invisibly.
- a `keras_input()`, then the output tensor from calling `layer(input)` is returned.
- `NULL` or missing, then a `Layer` instance is returned.

Example

```
x <- layer_input(shape = 32)
y <- layer_repeat_vector(x, n = 3)
shape(y)
```

```
## shape(NA, 3, 32)
```

Input Shape

2D tensor with shape `(batch_size, features)`.

Output Shape

3D tensor with shape `(batch_size, n, features)`.

See Also

- https://keras.io/api/layers/reshaping_layers/repeat_vector#repeatvector-class

Other reshaping layers:
layer_cropping_1d()
layer_cropping_2d()
layer_cropping_3d()
layer_flatten()
layer_permute()
layer_reshape()
layer_upsampling_1d()
layer_upsampling_2d()
layer_upsampling_3d()
layer_zero_padding_1d()
layer_zero_padding_2d()
layer_zero_padding_3d()

Other layers:
Layer()
layer_activation()
layer_activation_elu()
layer_activation_leaky_relu()
layer_activation_parametric_relu()
layer_activation_relu()
layer_activation_softmax()
layer_activity_regularization()
layer_add()
layer_additive_attention()
layer_alpha_dropout()
layer_attention()
layer_average()
layer_average_pooling_1d()
layer_average_pooling_2d()
layer_average_pooling_3d()
layer_batch_normalization()
layer_bidirectional()
layer_category_encoding()
layer_center_crop()
layer_concatenate()
layer_conv_1d()
layer_conv_1d_transpose()
layer_conv_2d()
layer_conv_2d_transpose()
layer_conv_3d()
layer_conv_3d_transpose()
layer_conv_lstm_1d()
layer_conv_lstm_2d()
layer_conv_lstm_3d()

layer_cropping_1d()
layer_cropping_2d()
layer_cropping_3d()
layer_dense()
layer_depthwise_conv_1d()
layer_depthwise_conv_2d()
layer_discretization()
layer_dot()
layer_dropout()
layer_einsum_dense()
layer_embedding()
layer_feature_space()
layer_flatten()
layer_flax_module_wrapper()
layer_gaussian_dropout()
layer_gaussian_noise()
layer_global_average_pooling_1d()
layer_global_average_pooling_2d()
layer_global_average_pooling_3d()
layer_global_max_pooling_1d()
layer_global_max_pooling_2d()
layer_global_max_pooling_3d()
layer_group_normalization()
layer_group_query_attention()
layer_gru()
layer_hashed_crossing()
layer_hashing()
layer_identity()
layer_integer_lookup()
layer_jax_model_wrapper()
layer_lambda()
layer_layer_normalization()
layer_lstm()
layer_masking()
layer_max_pooling_1d()
layer_max_pooling_2d()
layer_max_pooling_3d()
layer_maximum()
layer_mel_spectrogram()
layer_minimum()
layer_multi_head_attention()
layer_multiply()
layer_normalization()
layer_permute()
layer_random_brightness()
layer_random_contrast()
layer_random_crop()
layer_random_flip()

layer_random_rotation()
layer_random_translation()
layer_random_zoom()
layer_rescaling()
layer_reshape()
layer_resizing()
layer_rnn()
layer_separable_conv_1d()
layer_separable_conv_2d()
layer_simple_rnn()
layer_spatial_dropout_1d()
layer_spatial_dropout_2d()
layer_spatial_dropout_3d()
layer_spectral_normalization()
layer_string_lookup()
layer_subtract()
layer_text_vectorization()
layer_tfsm()
layer_time_distributed()
layer_torch_module_wrapper()
layer_unit_normalization()
layer_upsampling_1d()
layer_upsampling_2d()
layer_upsampling_3d()
layer_zero_padding_1d()
layer_zero_padding_2d()
layer_zero_padding_3d()
rnn_cell_gru()
rnn_cell_lstm()
rnn_cell_simple()
rnn_cells_stack()

layer_rescaling — *A preprocessing layer which rescales input values to a new range.*

Description

This layer rescales every value of an input (often an image) by multiplying by `scale` and adding `offset`.

For instance:

1. To rescale an input in the `[0, 255]` range to be in the `[0, 1]` range, you would pass `scale=1./255`.
2. To rescale an input in the `[0, 255]` range to be in the `[-1, 1]` range, you would pass `scale=1./127.5, offset=-1`.

The rescaling is applied both during training and inference. Inputs can be of integer or floating point dtype, and by default the layer will output floats.

Note: This layer is safe to use inside a `tf.data` pipeline (independently of which backend you're using).

Usage

```
layer_rescaling(object, scale, offset = 0, ...)
```

Arguments

`object`	Object to compose the layer with. A tensor, array, or sequential model.
`scale`	Float, the scale to apply to the inputs.
`offset`	Float, the offset to apply to the inputs.
`...`	Base layer keyword arguments, such as `name` and `dtype`.

Value

The return value depends on the value provided for the first argument. If `object` is:

- a `keras_model_sequential()`, then the layer is added to the sequential model (which is modified in place). To enable piping, the sequential model is also returned, invisibly.
- a `keras_input()`, then the output tensor from calling `layer(input)` is returned.
- `NULL` or missing, then a `Layer` instance is returned.

See Also

- https://keras.io/api/layers/preprocessing_layers/image_preprocessing/rescaling#rescaling-class

Other image preprocessing layers:
`layer_center_crop()`
`layer_resizing()`

Other preprocessing layers:
`layer_category_encoding()`
`layer_center_crop()`
`layer_discretization()`
`layer_feature_space()`
`layer_hashed_crossing()`
`layer_hashing()`
`layer_integer_lookup()`
`layer_mel_spectrogram()`
`layer_normalization()`
`layer_random_brightness()`
`layer_random_contrast()`
`layer_random_crop()`
`layer_random_flip()`

layer_random_rotation()
layer_random_translation()
layer_random_zoom()
layer_resizing()
layer_string_lookup()
layer_text_vectorization()

Other layers:
Layer()
layer_activation()
layer_activation_elu()
layer_activation_leaky_relu()
layer_activation_parametric_relu()
layer_activation_relu()
layer_activation_softmax()
layer_activity_regularization()
layer_add()
layer_additive_attention()
layer_alpha_dropout()
layer_attention()
layer_average()
layer_average_pooling_1d()
layer_average_pooling_2d()
layer_average_pooling_3d()
layer_batch_normalization()
layer_bidirectional()
layer_category_encoding()
layer_center_crop()
layer_concatenate()
layer_conv_1d()
layer_conv_1d_transpose()
layer_conv_2d()
layer_conv_2d_transpose()
layer_conv_3d()
layer_conv_3d_transpose()
layer_conv_lstm_1d()
layer_conv_lstm_2d()
layer_conv_lstm_3d()
layer_cropping_1d()
layer_cropping_2d()
layer_cropping_3d()
layer_dense()
layer_depthwise_conv_1d()
layer_depthwise_conv_2d()
layer_discretization()
layer_dot()
layer_dropout()
layer_einsum_dense()

layer_embedding()
layer_feature_space()
layer_flatten()
layer_flax_module_wrapper()
layer_gaussian_dropout()
layer_gaussian_noise()
layer_global_average_pooling_1d()
layer_global_average_pooling_2d()
layer_global_average_pooling_3d()
layer_global_max_pooling_1d()
layer_global_max_pooling_2d()
layer_global_max_pooling_3d()
layer_group_normalization()
layer_group_query_attention()
layer_gru()
layer_hashed_crossing()
layer_hashing()
layer_identity()
layer_integer_lookup()
layer_jax_model_wrapper()
layer_lambda()
layer_layer_normalization()
layer_lstm()
layer_masking()
layer_max_pooling_1d()
layer_max_pooling_2d()
layer_max_pooling_3d()
layer_maximum()
layer_mel_spectrogram()
layer_minimum()
layer_multi_head_attention()
layer_multiply()
layer_normalization()
layer_permute()
layer_random_brightness()
layer_random_contrast()
layer_random_crop()
layer_random_flip()
layer_random_rotation()
layer_random_translation()
layer_random_zoom()
layer_repeat_vector()
layer_reshape()
layer_resizing()
layer_rnn()
layer_separable_conv_1d()
layer_separable_conv_2d()
layer_simple_rnn()

`layer_spatial_dropout_1d()`
`layer_spatial_dropout_2d()`
`layer_spatial_dropout_3d()`
`layer_spectral_normalization()`
`layer_string_lookup()`
`layer_subtract()`
`layer_text_vectorization()`
`layer_tfsm()`
`layer_time_distributed()`
`layer_torch_module_wrapper()`
`layer_unit_normalization()`
`layer_upsampling_1d()`
`layer_upsampling_2d()`
`layer_upsampling_3d()`
`layer_zero_padding_1d()`
`layer_zero_padding_2d()`
`layer_zero_padding_3d()`
`rnn_cell_gru()`
`rnn_cell_lstm()`
`rnn_cell_simple()`
`rnn_cells_stack()`

`layer_reshape` *Layer that reshapes inputs into the given shape.*

Description

Layer that reshapes inputs into the given shape.

Usage

```
layer_reshape(object, target_shape, ...)
```

Arguments

`object`	Object to compose the layer with. A tensor, array, or sequential model.
`target_shape`	Target shape. List of integers, does not include the samples dimension (batch size).
`...`	For forward/backward compatability.

Value

The return value depends on the value provided for the first argument. If `object` is:

- a `keras_model_sequential()`, then the layer is added to the sequential model (which is modified in place). To enable piping, the sequential model is also returned, invisibly.
- a `keras_input()`, then the output tensor from calling `layer(input)` is returned.
- `NULL` or missing, then a `Layer` instance is returned.

Input Shape

Arbitrary, although all dimensions in the input shape must be known/fixed. Use the keyword argument input_shape (list of integers, does not include the samples/batch size axis) when using this layer as the first layer in a model.

Output Shape

```
(batch_size, *target_shape)
```

Examples

```
x <- layer_input(shape = 12)
y <- layer_reshape(x, c(3, 4))
shape(y)

## shape(NA, 3, 4)


# also supports shape inference using `-1` as dimension
y <- layer_reshape(x, c(-1, 2, 2))
shape(y)

## shape(NA, 3, 2, 2)
```

See Also

- https://keras.io/api/layers/reshaping_layers/reshape#reshape-class

Other reshaping layers:
layer_cropping_1d()
layer_cropping_2d()
layer_cropping_3d()
layer_flatten()
layer_permute()
layer_repeat_vector()
layer_upsampling_1d()
layer_upsampling_2d()
layer_upsampling_3d()
layer_zero_padding_1d()
layer_zero_padding_2d()
layer_zero_padding_3d()

Other layers:
Layer()
layer_activation()
layer_activation_elu()
layer_activation_leaky_relu()
layer_activation_parametric_relu()

layer_activation_relu()
layer_activation_softmax()
layer_activity_regularization()
layer_add()
layer_additive_attention()
layer_alpha_dropout()
layer_attention()
layer_average()
layer_average_pooling_1d()
layer_average_pooling_2d()
layer_average_pooling_3d()
layer_batch_normalization()
layer_bidirectional()
layer_category_encoding()
layer_center_crop()
layer_concatenate()
layer_conv_1d()
layer_conv_1d_transpose()
layer_conv_2d()
layer_conv_2d_transpose()
layer_conv_3d()
layer_conv_3d_transpose()
layer_conv_lstm_1d()
layer_conv_lstm_2d()
layer_conv_lstm_3d()
layer_cropping_1d()
layer_cropping_2d()
layer_cropping_3d()
layer_dense()
layer_depthwise_conv_1d()
layer_depthwise_conv_2d()
layer_discretization()
layer_dot()
layer_dropout()
layer_einsum_dense()
layer_embedding()
layer_feature_space()
layer_flatten()
layer_flax_module_wrapper()
layer_gaussian_dropout()
layer_gaussian_noise()
layer_global_average_pooling_1d()
layer_global_average_pooling_2d()
layer_global_average_pooling_3d()
layer_global_max_pooling_1d()
layer_global_max_pooling_2d()
layer_global_max_pooling_3d()
layer_group_normalization()

layer_group_query_attention()
layer_gru()
layer_hashed_crossing()
layer_hashing()
layer_identity()
layer_integer_lookup()
layer_jax_model_wrapper()
layer_lambda()
layer_layer_normalization()
layer_lstm()
layer_masking()
layer_max_pooling_1d()
layer_max_pooling_2d()
layer_max_pooling_3d()
layer_maximum()
layer_mel_spectrogram()
layer_minimum()
layer_multi_head_attention()
layer_multiply()
layer_normalization()
layer_permute()
layer_random_brightness()
layer_random_contrast()
layer_random_crop()
layer_random_flip()
layer_random_rotation()
layer_random_translation()
layer_random_zoom()
layer_repeat_vector()
layer_rescaling()
layer_resizing()
layer_rnn()
layer_separable_conv_1d()
layer_separable_conv_2d()
layer_simple_rnn()
layer_spatial_dropout_1d()
layer_spatial_dropout_2d()
layer_spatial_dropout_3d()
layer_spectral_normalization()
layer_string_lookup()
layer_subtract()
layer_text_vectorization()
layer_tfsm()
layer_time_distributed()
layer_torch_module_wrapper()
layer_unit_normalization()
layer_upsampling_1d()
layer_upsampling_2d()

layer_upsampling_3d()
layer_zero_padding_1d()
layer_zero_padding_2d()
layer_zero_padding_3d()
rnn_cell_gru()
rnn_cell_lstm()
rnn_cell_simple()
rnn_cells_stack()

---

layer_resizing — *A preprocessing layer which resizes images.*

Description

This layer resizes an image input to a target height and width. The input should be a 4D (batched) or 3D (unbatched) tensor in `"channels_last"` format. Input pixel values can be of any range (e.g. `[0., 1.)` or `[0, 255]`).

Usage

```
layer_resizing(
  object,
  height,
  width,
  interpolation = "bilinear",
  crop_to_aspect_ratio = FALSE,
  data_format = NULL,
  ...
)
```

Arguments

`object`	Object to compose the layer with. A tensor, array, or sequential model.
`height`	Integer, the height of the output shape.
`width`	Integer, the width of the output shape.
`interpolation`	String, the interpolation method. Supports `"bilinear"`, `"nearest"`, `"bicubic"`, `"lanczos3"`, `"lanczos5"`. Defaults to `"bilinear"`.
`crop_to_aspect_ratio`	If `TRUE`, resize the images without aspect ratio distortion. When the original aspect ratio differs from the target aspect ratio, the output image will be cropped so as to return the largest possible window in the image (of size `(height, width)`) that matches the target aspect ratio. By default (`crop_to_aspect_ratio=FALSE`), aspect ratio may not be preserved.

`data_format`	string, either `"channels_last"` or `"channels_first"`. The ordering of the dimensions in the inputs. `"channels_last"` corresponds to inputs with shape `(batch, height, width, channels)` while `"channels_first"` corresponds to inputs with shape `(batch, channels, height, width)`. It defaults to the `image_data_format` value found in your Keras config file at `~/.keras/keras.json`. If you never set it, then it will be `"channels_last"`.
`...`	Base layer keyword arguments, such as `name` and `dtype`.

Value

The return value depends on the value provided for the first argument. If `object` is:

- a `keras_model_sequential()`, then the layer is added to the sequential model (which is modified in place). To enable piping, the sequential model is also returned, invisibly.
- a `keras_input()`, then the output tensor from calling `layer(input)` is returned.
- `NULL` or missing, then a `Layer` instance is returned.

Input Shape

3D (unbatched) or 4D (batched) tensor with shape: `(..., height, width, channels)`, in `"channels_last"` format, or `(..., channels, height, width)`, in `"channels_first"` format.

Output Shape

3D (unbatched) or 4D (batched) tensor with shape: `(..., target_height, target_width, channels)`, or `(..., channels, target_height, target_width)`, in `"channels_first"` format.

Note: This layer is safe to use inside a `tf.data` pipeline (independently of which backend you're using).

See Also

- https://keras.io/api/layers/preprocessing_layers/image_preprocessing/resizing#resizing-class

Other image preprocessing layers:
`layer_center_crop()`
`layer_rescaling()`

Other preprocessing layers:
`layer_category_encoding()`
`layer_center_crop()`
`layer_discretization()`
`layer_feature_space()`
`layer_hashed_crossing()`
`layer_hashing()`
`layer_integer_lookup()`
`layer_mel_spectrogram()`
`layer_normalization()`

layer_random_brightness()
layer_random_contrast()
layer_random_crop()
layer_random_flip()
layer_random_rotation()
layer_random_translation()
layer_random_zoom()
layer_rescaling()
layer_string_lookup()
layer_text_vectorization()

Other layers:
Layer()
layer_activation()
layer_activation_elu()
layer_activation_leaky_relu()
layer_activation_parametric_relu()
layer_activation_relu()
layer_activation_softmax()
layer_activity_regularization()
layer_add()
layer_additive_attention()
layer_alpha_dropout()
layer_attention()
layer_average()
layer_average_pooling_1d()
layer_average_pooling_2d()
layer_average_pooling_3d()
layer_batch_normalization()
layer_bidirectional()
layer_category_encoding()
layer_center_crop()
layer_concatenate()
layer_conv_1d()
layer_conv_1d_transpose()
layer_conv_2d()
layer_conv_2d_transpose()
layer_conv_3d()
layer_conv_3d_transpose()
layer_conv_lstm_1d()
layer_conv_lstm_2d()
layer_conv_lstm_3d()
layer_cropping_1d()
layer_cropping_2d()
layer_cropping_3d()
layer_dense()
layer_depthwise_conv_1d()
layer_depthwise_conv_2d()

layer_discretization()
layer_dot()
layer_dropout()
layer_einsum_dense()
layer_embedding()
layer_feature_space()
layer_flatten()
layer_flax_module_wrapper()
layer_gaussian_dropout()
layer_gaussian_noise()
layer_global_average_pooling_1d()
layer_global_average_pooling_2d()
layer_global_average_pooling_3d()
layer_global_max_pooling_1d()
layer_global_max_pooling_2d()
layer_global_max_pooling_3d()
layer_group_normalization()
layer_group_query_attention()
layer_gru()
layer_hashed_crossing()
layer_hashing()
layer_identity()
layer_integer_lookup()
layer_jax_model_wrapper()
layer_lambda()
layer_layer_normalization()
layer_lstm()
layer_masking()
layer_max_pooling_1d()
layer_max_pooling_2d()
layer_max_pooling_3d()
layer_maximum()
layer_mel_spectrogram()
layer_minimum()
layer_multi_head_attention()
layer_multiply()
layer_normalization()
layer_permute()
layer_random_brightness()
layer_random_contrast()
layer_random_crop()
layer_random_flip()
layer_random_rotation()
layer_random_translation()
layer_random_zoom()
layer_repeat_vector()
layer_rescaling()
layer_reshape()

layer_rnn()
layer_separable_conv_1d()
layer_separable_conv_2d()
layer_simple_rnn()
layer_spatial_dropout_1d()
layer_spatial_dropout_2d()
layer_spatial_dropout_3d()
layer_spectral_normalization()
layer_string_lookup()
layer_subtract()
layer_text_vectorization()
layer_tfsm()
layer_time_distributed()
layer_torch_module_wrapper()
layer_unit_normalization()
layer_upsampling_1d()
layer_upsampling_2d()
layer_upsampling_3d()
layer_zero_padding_1d()
layer_zero_padding_2d()
layer_zero_padding_3d()
rnn_cell_gru()
rnn_cell_lstm()
rnn_cell_simple()
rnn_cells_stack()

layer_rnn *Base class for recurrent layers*

Description

Base class for recurrent layers

Usage

```
layer_rnn(
  object,
  cell,
  return_sequences = FALSE,
  return_state = FALSE,
  go_backwards = FALSE,
  stateful = FALSE,
  unroll = FALSE,
  zero_output_for_mask = FALSE,
  ...
)
```

Arguments

`object`	Object to compose the layer with. A tensor, array, or sequential model.
`cell`	A RNN cell instance or a list of RNN cell instances. A RNN cell is a class that has: • A `call(input_at_t, states_at_t)` method, returning `(output_at_t, states_at_t_plus_1)`. The call method of the cell can also take the optional argument `constants`, see section "Note on passing external constants" below. • A `state_size` attribute. This can be a single integer (single state) in which case it is the size of the recurrent state. This can also be a list of integers (one size per state). • A `output_size` attribute, a single integer. • A `get_initial_state(batch_size=NULL)` method that creates a tensor meant to be fed to `call()` as the initial state, if the user didn't specify any initial state via other means. The returned initial state should have shape `(batch_size, cell.state_size)`. The cell might choose to create a tensor full of zeros, or other values based on the cell's implementation. `inputs` is the input tensor to the RNN layer, with shape `(batch_size, timesteps, features)`. If this method is not implemented by the cell, the RNN layer will create a zero filled tensor with shape `(batch_size, cell$state_size)`. In the case that `cell` is a list of RNN cell instances, the cells will be stacked on top of each other in the RNN, resulting in an efficient stacked RNN.
`return_sequences`	Boolean (default `FALSE`). Whether to return the last output in the output sequence, or the full sequence.
`return_state`	Boolean (default `FALSE`). Whether to return the last state in addition to the output.
`go_backwards`	Boolean (default `FALSE`). If `TRUE`, process the input sequence backwards and return the reversed sequence.
`stateful`	Boolean (default `FALSE`). If TRUE, the last state for each sample at index `i` in a batch will be used as initial state for the sample of index `i` in the following batch.
`unroll`	Boolean (default `FALSE`). If TRUE, the network will be unrolled, else a symbolic loop will be used. Unrolling can speed-up a RNN, although it tends to be more memory-intensive. Unrolling is only suitable for short sequences.
`zero_output_for_mask`	Boolean (default `FALSE`). Whether the output should use zeros for the masked timesteps. Note that this field is only used when `return_sequences` is `TRUE` and `mask` is provided. It can useful if you want to reuse the raw output sequence of the RNN without interference from the masked timesteps, e.g., merging bidirectional RNNs.
`...`	For forward/backward compatability.

Value

The return value depends on the value provided for the first argument. If `object` is:

- a `keras_model_sequential()`, then the layer is added to the sequential model (which is modified in place). To enable piping, the sequential model is also returned, invisibly.
- a `keras_input()`, then the output tensor from calling `layer(input)` is returned.
- `NULL` or missing, then a `Layer` instance is returned.

Call Arguments

- `inputs`: Input tensor.
- `initial_state`: List of initial state tensors to be passed to the first call of the cell.
- `mask`: Binary tensor of shape `[batch_size, timesteps]` indicating whether a given timestep should be masked. An individual `TRUE` entry indicates that the corresponding timestep should be utilized, while a `FALSE` entry indicates that the corresponding timestep should be ignored.
- `training`: Python boolean indicating whether the layer should behave in training mode or in inference mode. This argument is passed to the cell when calling it. This is for use with cells that use dropout.

Input Shape

3-D tensor with shape `(batch_size, timesteps, features)`.

Output Shape

- If `return_state`: a list of tensors. The first tensor is the output. The remaining tensors are the last states, each with shape `(batch_size, state_size)`, where `state_size` could be a high dimension tensor shape.
- If `return_sequences`: 3D tensor with shape `(batch_size, timesteps, output_size)`.

Masking:

This layer supports masking for input data with a variable number of timesteps. To introduce masks to your data, use a `layer_embedding()` layer with the `mask_zero` parameter set to `TRUE`.

Note on using statefulness in RNNs:

You can set RNN layers to be 'stateful', which means that the states computed for the samples in one batch will be reused as initial states for the samples in the next batch. This assumes a one-to-one mapping between samples in different successive batches.

To enable statefulness:

- Specify `stateful = TRUE` in the layer constructor.
- Specify a fixed batch size for your model, by passing
  - If sequential model: `input_batch_shape = c(...)` to the `keras_model_sequential()` call.
  - Else for functional model with 1 or more input layers: `batch_shape = c(...)` to the `layer_input()` call(s).

  This is the expected shape of your inputs *including the batch size*. It should be a list of integers, e.g. `c(32, 10, 100)`.
- Specify `shuffle = FALSE` when calling `fit()`.

To reset the states of your model, call `reset_state()` on either a specific layer, or on your entire model.

Note on specifying the initial state of RNNs:

You can specify the initial state of RNN layers symbolically by calling them with the keyword argument `initial_state`. The value of `initial_state` should be a tensor or list of tensors representing the initial state of the RNN layer.

Examples

First, let's define a RNN Cell, as a layer subclass.

```
rnn_cell_minimal <- Layer(
  "MinimalRNNCell",

  initialize = function(units, ...) {
    super$initialize(...)
    self$units <- as.integer(units)
    self$state_size <- as.integer(units)
  },

  build = function(input_shape) {
    self$kernel <- self$add_weight(
      shape = shape(tail(input_shape, 1), self$units),
      initializer = 'uniform',
      name = 'kernel'
    )
    self$recurrent_kernel <- self$add_weight(
      shape = shape(self$units, self$units),
      initializer = 'uniform',
      name = 'recurrent_kernel'
    )
    self$built <- TRUE
  },

  call = function(inputs, states) {
    prev_output <- states[[1]]
    h <- op_matmul(inputs, self$kernel)
    output <- h + op_matmul(prev_output, self$recurrent_kernel)
    list(output, list(output))
  }
)
```

Let's use this cell in a RNN layer:

```
cell <- rnn_cell_minimal(units = 32)
x <- layer_input(shape = shape(NULL, 5))
layer <- layer_rnn(cell = cell)
y <- layer(x)
```

```
cells <- list(rnn_cell_minimal(units = 32), rnn_cell_minimal(units = 4))
x <- layer_input(shape = shape(NULL, 5))
layer <- layer_rnn(cell = cells)
y <- layer(x)
```

See Also

- https://keras.io/api/layers/recurrent_layers/rnn#rnn-class

Other rnn cells:
rnn_cell_gru()
rnn_cell_lstm()
rnn_cell_simple()

Other rnn layers:
layer_bidirectional()
layer_conv_lstm_1d()
layer_conv_lstm_2d()
layer_conv_lstm_3d()
layer_gru()
layer_lstm()
layer_simple_rnn()
layer_time_distributed()
rnn_cell_gru()
rnn_cell_lstm()
rnn_cell_simple()
rnn_cells_stack()

Other layers:
Layer()
layer_activation()
layer_activation_elu()
layer_activation_leaky_relu()
layer_activation_parametric_relu()
layer_activation_relu()
layer_activation_softmax()
layer_activity_regularization()
layer_add()
layer_additive_attention()
layer_alpha_dropout()
layer_attention()
layer_average()
layer_average_pooling_1d()
layer_average_pooling_2d()
layer_average_pooling_3d()
layer_batch_normalization()
layer_bidirectional()
layer_category_encoding()
layer_center_crop()

layer_concatenate()
layer_conv_1d()
layer_conv_1d_transpose()
layer_conv_2d()
layer_conv_2d_transpose()
layer_conv_3d()
layer_conv_3d_transpose()
layer_conv_lstm_1d()
layer_conv_lstm_2d()
layer_conv_lstm_3d()
layer_cropping_1d()
layer_cropping_2d()
layer_cropping_3d()
layer_dense()
layer_depthwise_conv_1d()
layer_depthwise_conv_2d()
layer_discretization()
layer_dot()
layer_dropout()
layer_einsum_dense()
layer_embedding()
layer_feature_space()
layer_flatten()
layer_flax_module_wrapper()
layer_gaussian_dropout()
layer_gaussian_noise()
layer_global_average_pooling_1d()
layer_global_average_pooling_2d()
layer_global_average_pooling_3d()
layer_global_max_pooling_1d()
layer_global_max_pooling_2d()
layer_global_max_pooling_3d()
layer_group_normalization()
layer_group_query_attention()
layer_gru()
layer_hashed_crossing()
layer_hashing()
layer_identity()
layer_integer_lookup()
layer_jax_model_wrapper()
layer_lambda()
layer_layer_normalization()
layer_lstm()
layer_masking()
layer_max_pooling_1d()
layer_max_pooling_2d()
layer_max_pooling_3d()
layer_maximum()

layer_mel_spectrogram()
layer_minimum()
layer_multi_head_attention()
layer_multiply()
layer_normalization()
layer_permute()
layer_random_brightness()
layer_random_contrast()
layer_random_crop()
layer_random_flip()
layer_random_rotation()
layer_random_translation()
layer_random_zoom()
layer_repeat_vector()
layer_rescaling()
layer_reshape()
layer_resizing()
layer_separable_conv_1d()
layer_separable_conv_2d()
layer_simple_rnn()
layer_spatial_dropout_1d()
layer_spatial_dropout_2d()
layer_spatial_dropout_3d()
layer_spectral_normalization()
layer_string_lookup()
layer_subtract()
layer_text_vectorization()
layer_tfsm()
layer_time_distributed()
layer_torch_module_wrapper()
layer_unit_normalization()
layer_upsampling_1d()
layer_upsampling_2d()
layer_upsampling_3d()
layer_zero_padding_1d()
layer_zero_padding_2d()
layer_zero_padding_3d()
rnn_cell_gru()
rnn_cell_lstm()
rnn_cell_simple()
rnn_cells_stack()

layer_separable_conv_1d *1D separable convolution layer.*

Description

This layer performs a depthwise convolution that acts separately on channels, followed by a pointwise convolution that mixes channels. If `use_bias` is TRUE and a bias initializer is provided, it adds a bias vector to the output. It then optionally applies an activation function to produce the final output.

Usage

```
layer_separable_conv_1d(
  object,
  filters,
  kernel_size,
  strides = 1L,
  padding = "valid",
  data_format = NULL,
  dilation_rate = 1L,
  depth_multiplier = 1L,
  activation = NULL,
  use_bias = TRUE,
  depthwise_initializer = "glorot_uniform",
  pointwise_initializer = "glorot_uniform",
  bias_initializer = "zeros",
  depthwise_regularizer = NULL,
  pointwise_regularizer = NULL,
  bias_regularizer = NULL,
  activity_regularizer = NULL,
  depthwise_constraint = NULL,
  pointwise_constraint = NULL,
  bias_constraint = NULL,
  ...
)
```

Arguments

`object`	Object to compose the layer with. A tensor, array, or sequential model.
`filters`	int, the dimensionality of the output space (i.e. the number of filters in the pointwise convolution).
`kernel_size`	int or list of 1 integers, specifying the size of the depthwise convolution window.
`strides`	int or list of 1 integers, specifying the stride length of the depthwise convolution. If only one int is specified, the same stride size will be used for all dimensions. `strides` > 1 is incompatible with `dilation_rate` > 1.
`padding`	string, either `"valid"` or `"same"` (case-insensitive). `"valid"` means no padding. `"same"` results in padding evenly to the left/right or up/down of the input. When `padding="same"` and `strides=1`, the output has the same size as the input.
`data_format`	string, either `"channels_last"` or `"channels_first"`. The ordering of the dimensions in the inputs. `"channels_last"` corresponds to inputs with shape `(batch, steps, features)` while `"channels_first"` corresponds to inputs

with shape `(batch, features, steps)`. It defaults to the `image_data_format` value found in your Keras config file at `~/.keras/keras.json`. If you never set it, then it will be `"channels_last"`.

`dilation_rate`	int or list of 1 integers, specifying the dilation rate to use for dilated convolution. If only one int is specified, the same dilation rate will be used for all dimensions.
`depth_multiplier`	The number of depthwise convolution output channels for each input channel. The total number of depthwise convolution output channels will be equal to `input_channel * depth_multiplier`.
`activation`	Activation function. If `NULL`, no activation is applied.
`use_bias`	bool, if `TRUE`, bias will be added to the output.
`depthwise_initializer`	An initializer for the depthwise convolution kernel. If NULL, then the default initializer (`"glorot_uniform"`) will be used.
`pointwise_initializer`	An initializer for the pointwise convolution kernel. If NULL, then the default initializer (`"glorot_uniform"`) will be used.
`bias_initializer`	An initializer for the bias vector. If NULL, the default initializer ('"zeros"') will be used.
`depthwise_regularizer`	Optional regularizer for the depthwise convolution kernel.
`pointwise_regularizer`	Optional regularizer for the pointwise convolution kernel.
`bias_regularizer`	Optional regularizer for the bias vector.
`activity_regularizer`	Optional regularizer function for the output.
`depthwise_constraint`	Optional projection function to be applied to the depthwise kernel after being updated by an `Optimizer` (e.g. used for norm constraints or value constraints for layer weights). The function must take as input the unprojected variable and must return the projected variable (which must have the same shape).
`pointwise_constraint`	Optional projection function to be applied to the pointwise kernel after being updated by an `Optimizer`.
`bias_constraint`	Optional projection function to be applied to the bias after being updated by an `Optimizer`.
`...`	For forward/backward compatability.

Value

A 3D tensor representing `activation(separable_conv1d(inputs, kernel) + bias)`.

Input Shape

- If data_format="channels_last": A 3D tensor with shape: (batch_shape, steps, channels)
- If data_format="channels_first": A 3D tensor with shape: (batch_shape, channels, steps)

Output Shape

- If data_format="channels_last": A 3D tensor with shape: (batch_shape, new_steps, filters)
- If data_format="channels_first": A 3D tensor with shape: (batch_shape, filters, new_steps)

Example

```
x <- random_uniform(c(4, 10, 12))
y <- layer_separable_conv_1d(x, 3, 2, 2, activation='relu')
shape(y)

## shape(4, 5, 3)
```

See Also

- https://keras.io/api/layers/convolution_layers/separable_convolution1d#separableconv1d-class

Other convolutional layers:
layer_conv_1d()
layer_conv_1d_transpose()
layer_conv_2d()
layer_conv_2d_transpose()
layer_conv_3d()
layer_conv_3d_transpose()
layer_depthwise_conv_1d()
layer_depthwise_conv_2d()
layer_separable_conv_2d()

Other layers:
Layer()
layer_activation()
layer_activation_elu()
layer_activation_leaky_relu()
layer_activation_parametric_relu()
layer_activation_relu()
layer_activation_softmax()
layer_activity_regularization()
layer_add()
layer_additive_attention()
layer_alpha_dropout()
layer_attention()
layer_average()
layer_average_pooling_1d()

layer_average_pooling_2d()
layer_average_pooling_3d()
layer_batch_normalization()
layer_bidirectional()
layer_category_encoding()
layer_center_crop()
layer_concatenate()
layer_conv_1d()
layer_conv_1d_transpose()
layer_conv_2d()
layer_conv_2d_transpose()
layer_conv_3d()
layer_conv_3d_transpose()
layer_conv_lstm_1d()
layer_conv_lstm_2d()
layer_conv_lstm_3d()
layer_cropping_1d()
layer_cropping_2d()
layer_cropping_3d()
layer_dense()
layer_depthwise_conv_1d()
layer_depthwise_conv_2d()
layer_discretization()
layer_dot()
layer_dropout()
layer_einsum_dense()
layer_embedding()
layer_feature_space()
layer_flatten()
layer_flax_module_wrapper()
layer_gaussian_dropout()
layer_gaussian_noise()
layer_global_average_pooling_1d()
layer_global_average_pooling_2d()
layer_global_average_pooling_3d()
layer_global_max_pooling_1d()
layer_global_max_pooling_2d()
layer_global_max_pooling_3d()
layer_group_normalization()
layer_group_query_attention()
layer_gru()
layer_hashed_crossing()
layer_hashing()
layer_identity()
layer_integer_lookup()
layer_jax_model_wrapper()
layer_lambda()
layer_layer_normalization()

layer_lstm()
layer_masking()
layer_max_pooling_1d()
layer_max_pooling_2d()
layer_max_pooling_3d()
layer_maximum()
layer_mel_spectrogram()
layer_minimum()
layer_multi_head_attention()
layer_multiply()
layer_normalization()
layer_permute()
layer_random_brightness()
layer_random_contrast()
layer_random_crop()
layer_random_flip()
layer_random_rotation()
layer_random_translation()
layer_random_zoom()
layer_repeat_vector()
layer_rescaling()
layer_reshape()
layer_resizing()
layer_rnn()
layer_separable_conv_2d()
layer_simple_rnn()
layer_spatial_dropout_1d()
layer_spatial_dropout_2d()
layer_spatial_dropout_3d()
layer_spectral_normalization()
layer_string_lookup()
layer_subtract()
layer_text_vectorization()
layer_tfsm()
layer_time_distributed()
layer_torch_module_wrapper()
layer_unit_normalization()
layer_upsampling_1d()
layer_upsampling_2d()
layer_upsampling_3d()
layer_zero_padding_1d()
layer_zero_padding_2d()
layer_zero_padding_3d()
rnn_cell_gru()
rnn_cell_lstm()
rnn_cell_simple()
rnn_cells_stack()

`layer_separable_conv_2d` *2D separable convolution layer.*

Description

This layer performs a depthwise convolution that acts separately on channels, followed by a point-wise convolution that mixes channels. If `use_bias` is TRUE and a bias initializer is provided, it adds a bias vector to the output. It then optionally applies an activation function to produce the final output.

Usage

```
layer_separable_conv_2d(
  object,
  filters,
  kernel_size,
  strides = list(1L, 1L),
  padding = "valid",
  data_format = NULL,
  dilation_rate = list(1L, 1L),
  depth_multiplier = 1L,
  activation = NULL,
  use_bias = TRUE,
  depthwise_initializer = "glorot_uniform",
  pointwise_initializer = "glorot_uniform",
  bias_initializer = "zeros",
  depthwise_regularizer = NULL,
  pointwise_regularizer = NULL,
  bias_regularizer = NULL,
  activity_regularizer = NULL,
  depthwise_constraint = NULL,
  pointwise_constraint = NULL,
  bias_constraint = NULL,
  ...
)
```

Arguments

`object`	Object to compose the layer with. A tensor, array, or sequential model.
`filters`	int, the dimensionality of the output space (i.e. the number of filters in the pointwise convolution).
`kernel_size`	int or list of 2 integers, specifying the size of the depthwise convolution window.
`strides`	int or list of 2 integers, specifying the stride length of the depthwise convolution. If only one int is specified, the same stride size will be used for all dimensions. `strides` > `1` is incompatible with `dilation_rate` > `1`.

padding	string, either "valid" or "same" (case-insensitive). "valid" means no padding. "same" results in padding evenly to the left/right or up/down of the input. When padding="same" and strides=1, the output has the same size as the input.
data_format	string, either "channels_last" or "channels_first". The ordering of the dimensions in the inputs. "channels_last" corresponds to inputs with shape (batch, height, width, channels) while "channels_first" corresponds to inputs with shape (batch, channels, height, width). It defaults to the image_data_format value found in your Keras config file at ~/.keras/keras.json. If you never set it, then it will be "channels_last".
dilation_rate	int or list of 2 integers, specifying the dilation rate to use for dilated convolution. If only one int is specified, the same dilation rate will be used for all dimensions.
depth_multiplier	The number of depthwise convolution output channels for each input channel. The total number of depthwise convolution output channels will be equal to input_channel * depth_multiplier.
activation	Activation function. If NULL, no activation is applied.
use_bias	bool, if TRUE, bias will be added to the output.
depthwise_initializer	An initializer for the depthwise convolution kernel. If NULL, then the default initializer ("glorot_uniform") will be used.
pointwise_initializer	An initializer for the pointwise convolution kernel. If NULL, then the default initializer ("glorot_uniform") will be used.
bias_initializer	An initializer for the bias vector. If NULL, the default initializer ('"zeros"') will be used.
depthwise_regularizer	Optional regularizer for the depthwise convolution kernel.
pointwise_regularizer	Optional regularizer for the pointwise convolution kernel.
bias_regularizer	Optional regularizer for the bias vector.
activity_regularizer	Optional regularizer function for the output.
depthwise_constraint	Optional projection function to be applied to the depthwise kernel after being updated by an Optimizer (e.g. used for norm constraints or value constraints for layer weights). The function must take as input the unprojected variable and must return the projected variable (which must have the same shape).
pointwise_constraint	Optional projection function to be applied to the pointwise kernel after being updated by an Optimizer.
bias_constraint	Optional projection function to be applied to the bias after being updated by an Optimizer.
...	For forward/backward compatability.

Value

A 4D tensor representing activation(separable_conv2d(inputs, kernel) + bias).

Input Shape

- If data_format="channels_last": A 4D tensor with shape: (batch_size, height, width, channels)
- If data_format="channels_first": A 4D tensor with shape: (batch_size, channels, height, width)

Output Shape

- If data_format="channels_last": A 4D tensor with shape: (batch_size, new_height, new_width, filters)
- If data_format="channels_first": A 4D tensor with shape: (batch_size, filters, new_height, new_width)

Example

```
x <- random_uniform(c(4, 10, 10, 12))
y <- layer_separable_conv_2d(x, 3, c(4, 3), 2, activation='relu')
shape(y)

## shape(4, 4, 4, 3)
```

See Also

- https://keras.io/api/layers/convolution_layers/separable_convolution2d#separableconv2d-class

Other convolutional layers:
layer_conv_1d()
layer_conv_1d_transpose()
layer_conv_2d()
layer_conv_2d_transpose()
layer_conv_3d()
layer_conv_3d_transpose()
layer_depthwise_conv_1d()
layer_depthwise_conv_2d()
layer_separable_conv_1d()

Other layers:
Layer()
layer_activation()
layer_activation_elu()
layer_activation_leaky_relu()
layer_activation_parametric_relu()
layer_activation_relu()
layer_activation_softmax()
layer_activity_regularization()
layer_add()
layer_additive_attention()

layer_alpha_dropout()
layer_attention()
layer_average()
layer_average_pooling_1d()
layer_average_pooling_2d()
layer_average_pooling_3d()
layer_batch_normalization()
layer_bidirectional()
layer_category_encoding()
layer_center_crop()
layer_concatenate()
layer_conv_1d()
layer_conv_1d_transpose()
layer_conv_2d()
layer_conv_2d_transpose()
layer_conv_3d()
layer_conv_3d_transpose()
layer_conv_lstm_1d()
layer_conv_lstm_2d()
layer_conv_lstm_3d()
layer_cropping_1d()
layer_cropping_2d()
layer_cropping_3d()
layer_dense()
layer_depthwise_conv_1d()
layer_depthwise_conv_2d()
layer_discretization()
layer_dot()
layer_dropout()
layer_einsum_dense()
layer_embedding()
layer_feature_space()
layer_flatten()
layer_flax_module_wrapper()
layer_gaussian_dropout()
layer_gaussian_noise()
layer_global_average_pooling_1d()
layer_global_average_pooling_2d()
layer_global_average_pooling_3d()
layer_global_max_pooling_1d()
layer_global_max_pooling_2d()
layer_global_max_pooling_3d()
layer_group_normalization()
layer_group_query_attention()
layer_gru()
layer_hashed_crossing()
layer_hashing()
layer_identity()

layer_integer_lookup()
layer_jax_model_wrapper()
layer_lambda()
layer_layer_normalization()
layer_lstm()
layer_masking()
layer_max_pooling_1d()
layer_max_pooling_2d()
layer_max_pooling_3d()
layer_maximum()
layer_mel_spectrogram()
layer_minimum()
layer_multi_head_attention()
layer_multiply()
layer_normalization()
layer_permute()
layer_random_brightness()
layer_random_contrast()
layer_random_crop()
layer_random_flip()
layer_random_rotation()
layer_random_translation()
layer_random_zoom()
layer_repeat_vector()
layer_rescaling()
layer_reshape()
layer_resizing()
layer_rnn()
layer_separable_conv_1d()
layer_simple_rnn()
layer_spatial_dropout_1d()
layer_spatial_dropout_2d()
layer_spatial_dropout_3d()
layer_spectral_normalization()
layer_string_lookup()
layer_subtract()
layer_text_vectorization()
layer_tfsm()
layer_time_distributed()
layer_torch_module_wrapper()
layer_unit_normalization()
layer_upsampling_1d()
layer_upsampling_2d()
layer_upsampling_3d()
layer_zero_padding_1d()
layer_zero_padding_2d()
layer_zero_padding_3d()
rnn_cell_gru()

rnn_cell_lstm()
rnn_cell_simple()
rnn_cells_stack()

layer_simple_rnn *Fully-connected RNN where the output is to be fed back as the new input.*

Description

Fully-connected RNN where the output is to be fed back as the new input.

Usage

```
layer_simple_rnn(
  object,
  units,
  activation = "tanh",
  use_bias = TRUE,
  kernel_initializer = "glorot_uniform",
  recurrent_initializer = "orthogonal",
  bias_initializer = "zeros",
  kernel_regularizer = NULL,
  recurrent_regularizer = NULL,
  bias_regularizer = NULL,
  activity_regularizer = NULL,
  kernel_constraint = NULL,
  recurrent_constraint = NULL,
  bias_constraint = NULL,
  dropout = 0,
  recurrent_dropout = 0,
  return_sequences = FALSE,
  return_state = FALSE,
  go_backwards = FALSE,
  stateful = FALSE,
  unroll = FALSE,
  seed = NULL,
  ...
)
```

Arguments

object	Object to compose the layer with. A tensor, array, or sequential model.
units	Positive integer, dimensionality of the output space.
activation	Activation function to use. Default: hyperbolic tangent (tanh). If you pass NULL, no activation is applied (ie. "linear" activation: a(x) = x).

`use_bias`	Boolean, (default `TRUE`), whether the layer uses a bias vector.
`kernel_initializer`	Initializer for the `kernel` weights matrix, used for the linear transformation of the inputs. Default: `"glorot_uniform"`.
`recurrent_initializer`	Initializer for the `recurrent_kernel` weights matrix, used for the linear transformation of the recurrent state. Default: `"orthogonal"`.
`bias_initializer`	Initializer for the bias vector. Default: `"zeros"`.
`kernel_regularizer`	Regularizer function applied to the `kernel` weights matrix. Default: `NULL`.
`recurrent_regularizer`	Regularizer function applied to the `recurrent_kernel` weights matrix. Default: `NULL`.
`bias_regularizer`	Regularizer function applied to the bias vector. Default: `NULL`.
`activity_regularizer`	Regularizer function applied to the output of the layer (its "activation"). Default: `NULL`.
`kernel_constraint`	Constraint function applied to the `kernel` weights matrix. Default: `NULL`.
`recurrent_constraint`	Constraint function applied to the `recurrent_kernel` weights matrix. Default: `NULL`.
`bias_constraint`	Constraint function applied to the bias vector. Default: `NULL`.
`dropout`	Float between 0 and 1. Fraction of the units to drop for the linear transformation of the inputs. Default: 0.
`recurrent_dropout`	Float between 0 and 1. Fraction of the units to drop for the linear transformation of the recurrent state. Default: 0.
`return_sequences`	Boolean. Whether to return the last output in the output sequence, or the full sequence. Default: `FALSE`.
`return_state`	Boolean. Whether to return the last state in addition to the output. Default: `FALSE`.
`go_backwards`	Boolean (default: `FALSE`). If `TRUE`, process the input sequence backwards and return the reversed sequence.
`stateful`	Boolean (default: `FALSE`). If `TRUE`, the last state for each sample at index i in a batch will be used as initial state for the sample of index i in the following batch.
`unroll`	Boolean (default: `FALSE`). If `TRUE`, the network will be unrolled, else a symbolic loop will be used. Unrolling can speed-up a RNN, although it tends to be more memory-intensive. Unrolling is only suitable for short sequences.
`seed`	Initial seed for the random number generator
`...`	For forward/backward compatability.

Value

The return value depends on the value provided for the first argument. If `object` is:

- a `keras_model_sequential()`, then the layer is added to the sequential model (which is modified in place). To enable piping, the sequential model is also returned, invisibly.
- a `keras_input()`, then the output tensor from calling `layer(input)` is returned.
- `NULL` or missing, then a `Layer` instance is returned.

Call Arguments

- `sequence`: A 3D tensor, with shape `[batch, timesteps, feature]`.
- `mask`: Binary tensor of shape `[batch, timesteps]` indicating whether a given timestep should be masked. An individual `TRUE` entry indicates that the corresponding timestep should be utilized, while a `FALSE` entry indicates that the corresponding timestep should be ignored.
- `training`: Python boolean indicating whether the layer should behave in training mode or in inference mode. This argument is passed to the cell when calling it. This is only relevant if `dropout` or `recurrent_dropout` is used.
- `initial_state`: List of initial state tensors to be passed to the first call of the cell.

Examples

```
inputs <- random_uniform(c(32, 10, 8))
simple_rnn <- layer_simple_rnn(units = 4)
output <- simple_rnn(inputs)  # The output has shape `(32, 4)`.
simple_rnn <- layer_simple_rnn(
    units = 4, return_sequences=TRUE, return_state=TRUE
)
# whole_sequence_output has shape `(32, 10, 4)`.
# final_state has shape `(32, 4)`.
c(whole_sequence_output, final_state) %<-% simple_rnn(inputs)
```

See Also

- https://keras.io/api/layers/recurrent_layers/simple_rnn#simplernn-class

Other simple rnn layers:
`rnn_cell_simple()`

Other rnn layers:
`layer_bidirectional()`
`layer_conv_lstm_1d()`
`layer_conv_lstm_2d()`
`layer_conv_lstm_3d()`
`layer_gru()`
`layer_lstm()`
`layer_rnn()`
`layer_time_distributed()`
`rnn_cell_gru()`

rnn_cell_lstm()
rnn_cell_simple()
rnn_cells_stack()

Other layers:
Layer()
layer_activation()
layer_activation_elu()
layer_activation_leaky_relu()
layer_activation_parametric_relu()
layer_activation_relu()
layer_activation_softmax()
layer_activity_regularization()
layer_add()
layer_additive_attention()
layer_alpha_dropout()
layer_attention()
layer_average()
layer_average_pooling_1d()
layer_average_pooling_2d()
layer_average_pooling_3d()
layer_batch_normalization()
layer_bidirectional()
layer_category_encoding()
layer_center_crop()
layer_concatenate()
layer_conv_1d()
layer_conv_1d_transpose()
layer_conv_2d()
layer_conv_2d_transpose()
layer_conv_3d()
layer_conv_3d_transpose()
layer_conv_lstm_1d()
layer_conv_lstm_2d()
layer_conv_lstm_3d()
layer_cropping_1d()
layer_cropping_2d()
layer_cropping_3d()
layer_dense()
layer_depthwise_conv_1d()
layer_depthwise_conv_2d()
layer_discretization()
layer_dot()
layer_dropout()
layer_einsum_dense()
layer_embedding()
layer_feature_space()
layer_flatten()

layer_flax_module_wrapper()
layer_gaussian_dropout()
layer_gaussian_noise()
layer_global_average_pooling_1d()
layer_global_average_pooling_2d()
layer_global_average_pooling_3d()
layer_global_max_pooling_1d()
layer_global_max_pooling_2d()
layer_global_max_pooling_3d()
layer_group_normalization()
layer_group_query_attention()
layer_gru()
layer_hashed_crossing()
layer_hashing()
layer_identity()
layer_integer_lookup()
layer_jax_model_wrapper()
layer_lambda()
layer_layer_normalization()
layer_lstm()
layer_masking()
layer_max_pooling_1d()
layer_max_pooling_2d()
layer_max_pooling_3d()
layer_maximum()
layer_mel_spectrogram()
layer_minimum()
layer_multi_head_attention()
layer_multiply()
layer_normalization()
layer_permute()
layer_random_brightness()
layer_random_contrast()
layer_random_crop()
layer_random_flip()
layer_random_rotation()
layer_random_translation()
layer_random_zoom()
layer_repeat_vector()
layer_rescaling()
layer_reshape()
layer_resizing()
layer_rnn()
layer_separable_conv_1d()
layer_separable_conv_2d()
layer_spatial_dropout_1d()
layer_spatial_dropout_2d()
layer_spatial_dropout_3d()

`layer_spectral_normalization()`
`layer_string_lookup()`
`layer_subtract()`
`layer_text_vectorization()`
`layer_tfsm()`
`layer_time_distributed()`
`layer_torch_module_wrapper()`
`layer_unit_normalization()`
`layer_upsampling_1d()`
`layer_upsampling_2d()`
`layer_upsampling_3d()`
`layer_zero_padding_1d()`
`layer_zero_padding_2d()`
`layer_zero_padding_3d()`
`rnn_cell_gru()`
`rnn_cell_lstm()`
`rnn_cell_simple()`
`rnn_cells_stack()`

`layer_spatial_dropout_1d`
Spatial 1D version of Dropout.

Description

This layer performs the same function as Dropout, however, it drops entire 1D feature maps instead of individual elements. If adjacent frames within feature maps are strongly correlated (as is normally the case in early convolution layers) then regular dropout will not regularize the activations and will otherwise just result in an effective learning rate decrease. In this case, `SpatialDropout1D` will help promote independence between feature maps and should be used instead.

Usage

```
layer_spatial_dropout_1d(object, rate, seed = NULL, name = NULL, dtype = NULL)
```

Arguments

`object`	Object to compose the layer with. A tensor, array, or sequential model.
`rate`	Float between 0 and 1. Fraction of the input units to drop.
`seed`	Initial seed for the random number generator
`name`	String, name for the object
`dtype`	datatype (e.g., `"float32"`).

Value

The return value depends on the value provided for the first argument. If `object` is:

- a `keras_model_sequential()`, then the layer is added to the sequential model (which is modified in place). To enable piping, the sequential model is also returned, invisibly.
- a `keras_input()`, then the output tensor from calling `layer(input)` is returned.
- `NULL` or missing, then a `Layer` instance is returned.

Call Arguments

- `inputs`: A 3D tensor.
- `training`: Python boolean indicating whether the layer should behave in training mode (applying dropout) or in inference mode (pass-through).

Input Shape

3D tensor with shape: `(samples, timesteps, channels)`

Output Shape

Same as input.

Reference

- Tompson et al., 2014

See Also

- `https://keras.io/api/layers/regularization_layers/spatial_dropout1d#spatialdropout1d-class`

Other spatial dropout regularization layers:
`layer_spatial_dropout_2d()`
`layer_spatial_dropout_3d()`

Other regularization layers:
`layer_activity_regularization()`
`layer_alpha_dropout()`
`layer_dropout()`
`layer_gaussian_dropout()`
`layer_gaussian_noise()`
`layer_spatial_dropout_2d()`
`layer_spatial_dropout_3d()`

Other layers:
`Layer()`
`layer_activation()`
`layer_activation_elu()`
`layer_activation_leaky_relu()`

layer_activation_parametric_relu()
layer_activation_relu()
layer_activation_softmax()
layer_activity_regularization()
layer_add()
layer_additive_attention()
layer_alpha_dropout()
layer_attention()
layer_average()
layer_average_pooling_1d()
layer_average_pooling_2d()
layer_average_pooling_3d()
layer_batch_normalization()
layer_bidirectional()
layer_category_encoding()
layer_center_crop()
layer_concatenate()
layer_conv_1d()
layer_conv_1d_transpose()
layer_conv_2d()
layer_conv_2d_transpose()
layer_conv_3d()
layer_conv_3d_transpose()
layer_conv_lstm_1d()
layer_conv_lstm_2d()
layer_conv_lstm_3d()
layer_cropping_1d()
layer_cropping_2d()
layer_cropping_3d()
layer_dense()
layer_depthwise_conv_1d()
layer_depthwise_conv_2d()
layer_discretization()
layer_dot()
layer_dropout()
layer_einsum_dense()
layer_embedding()
layer_feature_space()
layer_flatten()
layer_flax_module_wrapper()
layer_gaussian_dropout()
layer_gaussian_noise()
layer_global_average_pooling_1d()
layer_global_average_pooling_2d()
layer_global_average_pooling_3d()
layer_global_max_pooling_1d()
layer_global_max_pooling_2d()
layer_global_max_pooling_3d()

layer_group_normalization()
layer_group_query_attention()
layer_gru()
layer_hashed_crossing()
layer_hashing()
layer_identity()
layer_integer_lookup()
layer_jax_model_wrapper()
layer_lambda()
layer_layer_normalization()
layer_lstm()
layer_masking()
layer_max_pooling_1d()
layer_max_pooling_2d()
layer_max_pooling_3d()
layer_maximum()
layer_mel_spectrogram()
layer_minimum()
layer_multi_head_attention()
layer_multiply()
layer_normalization()
layer_permute()
layer_random_brightness()
layer_random_contrast()
layer_random_crop()
layer_random_flip()
layer_random_rotation()
layer_random_translation()
layer_random_zoom()
layer_repeat_vector()
layer_rescaling()
layer_reshape()
layer_resizing()
layer_rnn()
layer_separable_conv_1d()
layer_separable_conv_2d()
layer_simple_rnn()
layer_spatial_dropout_2d()
layer_spatial_dropout_3d()
layer_spectral_normalization()
layer_string_lookup()
layer_subtract()
layer_text_vectorization()
layer_tfsm()
layer_time_distributed()
layer_torch_module_wrapper()
layer_unit_normalization()
layer_upsampling_1d()

`layer_upsampling_2d()`
`layer_upsampling_3d()`
`layer_zero_padding_1d()`
`layer_zero_padding_2d()`
`layer_zero_padding_3d()`
`rnn_cell_gru()`
`rnn_cell_lstm()`
`rnn_cell_simple()`
`rnn_cells_stack()`

`layer_spatial_dropout_2d`

Spatial 2D version of Dropout.

Description

This version performs the same function as Dropout, however, it drops entire 2D feature maps instead of individual elements. If adjacent pixels within feature maps are strongly correlated (as is normally the case in early convolution layers) then regular dropout will not regularize the activations and will otherwise just result in an effective learning rate decrease. In this case, `SpatialDropout2D` will help promote independence between feature maps and should be used instead.

Usage

```
layer_spatial_dropout_2d(
  object,
  rate,
  data_format = NULL,
  seed = NULL,
  name = NULL,
  dtype = NULL
)
```

Arguments

`object`	Object to compose the layer with. A tensor, array, or sequential model.
`rate`	Float between 0 and 1. Fraction of the input units to drop.
`data_format`	`"channels_first"` or `"channels_last"`. In `"channels_first"` mode, the channels dimension (the depth) is at index 1, in `"channels_last"` mode is it at index 3. It defaults to the `image_data_format` value found in your Keras config file at `~/.keras/keras.json`. If you never set it, then it will be `"channels_last"`.
`seed`	Initial seed for the random number generator
`name`	String, name for the object
`dtype`	datatype (e.g., `"float32"`).

Value

The return value depends on the value provided for the first argument. If `object` is:

- a `keras_model_sequential()`, then the layer is added to the sequential model (which is modified in place). To enable piping, the sequential model is also returned, invisibly.
- a `keras_input()`, then the output tensor from calling `layer(input)` is returned.
- `NULL` or missing, then a `Layer` instance is returned.

Call Arguments

- `inputs`: A 4D tensor.
- `training`: Python boolean indicating whether the layer should behave in training mode (applying dropout) or in inference mode (pass-through).

Input Shape

4D tensor with shape: `(samples, channels, rows, cols)` if data_format='channels_first' or 4D tensor with shape: `(samples, rows, cols, channels)` if data_format='channels_last'.

Output Shape

Same as input.

Reference

- Tompson et al., 2014

See Also

- `https://keras.io/api/layers/regularization_layers/spatial_dropout2d#spatialdropout2d-class`

Other spatial dropout regularization layers:
`layer_spatial_dropout_1d()`
`layer_spatial_dropout_3d()`

Other regularization layers:
`layer_activity_regularization()`
`layer_alpha_dropout()`
`layer_dropout()`
`layer_gaussian_dropout()`
`layer_gaussian_noise()`
`layer_spatial_dropout_1d()`
`layer_spatial_dropout_3d()`

Other layers:
`Layer()`
`layer_activation()`
`layer_activation_elu()`

layer_activation_leaky_relu()
layer_activation_parametric_relu()
layer_activation_relu()
layer_activation_softmax()
layer_activity_regularization()
layer_add()
layer_additive_attention()
layer_alpha_dropout()
layer_attention()
layer_average()
layer_average_pooling_1d()
layer_average_pooling_2d()
layer_average_pooling_3d()
layer_batch_normalization()
layer_bidirectional()
layer_category_encoding()
layer_center_crop()
layer_concatenate()
layer_conv_1d()
layer_conv_1d_transpose()
layer_conv_2d()
layer_conv_2d_transpose()
layer_conv_3d()
layer_conv_3d_transpose()
layer_conv_lstm_1d()
layer_conv_lstm_2d()
layer_conv_lstm_3d()
layer_cropping_1d()
layer_cropping_2d()
layer_cropping_3d()
layer_dense()
layer_depthwise_conv_1d()
layer_depthwise_conv_2d()
layer_discretization()
layer_dot()
layer_dropout()
layer_einsum_dense()
layer_embedding()
layer_feature_space()
layer_flatten()
layer_flax_module_wrapper()
layer_gaussian_dropout()
layer_gaussian_noise()
layer_global_average_pooling_1d()
layer_global_average_pooling_2d()
layer_global_average_pooling_3d()
layer_global_max_pooling_1d()
layer_global_max_pooling_2d()

layer_global_max_pooling_3d()
layer_group_normalization()
layer_group_query_attention()
layer_gru()
layer_hashed_crossing()
layer_hashing()
layer_identity()
layer_integer_lookup()
layer_jax_model_wrapper()
layer_lambda()
layer_layer_normalization()
layer_lstm()
layer_masking()
layer_max_pooling_1d()
layer_max_pooling_2d()
layer_max_pooling_3d()
layer_maximum()
layer_mel_spectrogram()
layer_minimum()
layer_multi_head_attention()
layer_multiply()
layer_normalization()
layer_permute()
layer_random_brightness()
layer_random_contrast()
layer_random_crop()
layer_random_flip()
layer_random_rotation()
layer_random_translation()
layer_random_zoom()
layer_repeat_vector()
layer_rescaling()
layer_reshape()
layer_resizing()
layer_rnn()
layer_separable_conv_1d()
layer_separable_conv_2d()
layer_simple_rnn()
layer_spatial_dropout_1d()
layer_spatial_dropout_3d()
layer_spectral_normalization()
layer_string_lookup()
layer_subtract()
layer_text_vectorization()
layer_tfsm()
layer_time_distributed()
layer_torch_module_wrapper()
layer_unit_normalization()

layer_upsampling_1d()
layer_upsampling_2d()
layer_upsampling_3d()
layer_zero_padding_1d()
layer_zero_padding_2d()
layer_zero_padding_3d()
rnn_cell_gru()
rnn_cell_lstm()
rnn_cell_simple()
rnn_cells_stack()

---

layer_spatial_dropout_3d

Spatial 3D version of Dropout.

Description

This version performs the same function as Dropout, however, it drops entire 3D feature maps instead of individual elements. If adjacent voxels within feature maps are strongly correlated (as is normally the case in early convolution layers) then regular dropout will not regularize the activations and will otherwise just result in an effective learning rate decrease. In this case, SpatialDropout3D will help promote independence between feature maps and should be used instead.

Usage

```
layer_spatial_dropout_3d(
  object,
  rate,
  data_format = NULL,
  seed = NULL,
  name = NULL,
  dtype = NULL
)
```

Arguments

object	Object to compose the layer with. A tensor, array, or sequential model.
rate	Float between 0 and 1. Fraction of the input units to drop.
data_format	"channels_first" or "channels_last". In "channels_first" mode, the channels dimension (the depth) is at index 1, in "channels_last" mode is it at index 4. It defaults to the image_data_format value found in your Keras config file at ~/.keras/keras.json. If you never set it, then it will be "channels_last".
seed	Initial seed for the random number generator
name	String, name for the object
dtype	datatype (e.g., "float32").

Value

The return value depends on the value provided for the first argument. If `object` is:

- a `keras_model_sequential()`, then the layer is added to the sequential model (which is modified in place). To enable piping, the sequential model is also returned, invisibly.
- a `keras_input()`, then the output tensor from calling `layer(input)` is returned.
- `NULL` or missing, then a `Layer` instance is returned.

Call Arguments

- `inputs`: A 5D tensor.
- `training`: Python boolean indicating whether the layer should behave in training mode (applying dropout) or in inference mode (pass-through).

Input Shape

5D tensor with shape: `(samples, channels, dim1, dim2, dim3)` if data_format='channels_first' or 5D tensor with shape: `(samples, dim1, dim2, dim3, channels)` if data_format='channels_last'.

Output Shape

Same as input.

Reference

- Tompson et al., 2014

See Also

- `https://keras.io/api/layers/regularization_layers/spatial_dropout3d#spatialdropout3d-class`

Other spatial dropout regularization layers:
`layer_spatial_dropout_1d()`
`layer_spatial_dropout_2d()`

Other regularization layers:
`layer_activity_regularization()`
`layer_alpha_dropout()`
`layer_dropout()`
`layer_gaussian_dropout()`
`layer_gaussian_noise()`
`layer_spatial_dropout_1d()`
`layer_spatial_dropout_2d()`

Other layers:
`Layer()`
`layer_activation()`
`layer_activation_elu()`

layer_activation_leaky_relu()
layer_activation_parametric_relu()
layer_activation_relu()
layer_activation_softmax()
layer_activity_regularization()
layer_add()
layer_additive_attention()
layer_alpha_dropout()
layer_attention()
layer_average()
layer_average_pooling_1d()
layer_average_pooling_2d()
layer_average_pooling_3d()
layer_batch_normalization()
layer_bidirectional()
layer_category_encoding()
layer_center_crop()
layer_concatenate()
layer_conv_1d()
layer_conv_1d_transpose()
layer_conv_2d()
layer_conv_2d_transpose()
layer_conv_3d()
layer_conv_3d_transpose()
layer_conv_lstm_1d()
layer_conv_lstm_2d()
layer_conv_lstm_3d()
layer_cropping_1d()
layer_cropping_2d()
layer_cropping_3d()
layer_dense()
layer_depthwise_conv_1d()
layer_depthwise_conv_2d()
layer_discretization()
layer_dot()
layer_dropout()
layer_einsum_dense()
layer_embedding()
layer_feature_space()
layer_flatten()
layer_flax_module_wrapper()
layer_gaussian_dropout()
layer_gaussian_noise()
layer_global_average_pooling_1d()
layer_global_average_pooling_2d()
layer_global_average_pooling_3d()
layer_global_max_pooling_1d()
layer_global_max_pooling_2d()

layer_global_max_pooling_3d()
layer_group_normalization()
layer_group_query_attention()
layer_gru()
layer_hashed_crossing()
layer_hashing()
layer_identity()
layer_integer_lookup()
layer_jax_model_wrapper()
layer_lambda()
layer_layer_normalization()
layer_lstm()
layer_masking()
layer_max_pooling_1d()
layer_max_pooling_2d()
layer_max_pooling_3d()
layer_maximum()
layer_mel_spectrogram()
layer_minimum()
layer_multi_head_attention()
layer_multiply()
layer_normalization()
layer_permute()
layer_random_brightness()
layer_random_contrast()
layer_random_crop()
layer_random_flip()
layer_random_rotation()
layer_random_translation()
layer_random_zoom()
layer_repeat_vector()
layer_rescaling()
layer_reshape()
layer_resizing()
layer_rnn()
layer_separable_conv_1d()
layer_separable_conv_2d()
layer_simple_rnn()
layer_spatial_dropout_1d()
layer_spatial_dropout_2d()
layer_spectral_normalization()
layer_string_lookup()
layer_subtract()
layer_text_vectorization()
layer_tfsm()
layer_time_distributed()
layer_torch_module_wrapper()
layer_unit_normalization()

layer_upsampling_1d()
layer_upsampling_2d()
layer_upsampling_3d()
layer_zero_padding_1d()
layer_zero_padding_2d()
layer_zero_padding_3d()
rnn_cell_gru()
rnn_cell_lstm()
rnn_cell_simple()
rnn_cells_stack()

---

layer_spectral_normalization

Performs spectral normalization on the weights of a target layer.

Description

This wrapper controls the Lipschitz constant of the weights of a layer by constraining their spectral norm, which can stabilize the training of GANs.

Usage

```
layer_spectral_normalization(object, layer, power_iterations = 1L, ...)
```

Arguments

`object`	Object to compose the layer with. A tensor, array, or sequential model.
`layer`	A `Layer` instance that has either a `kernel` (e.g. `layer_conv_2d`, `layer_dense`...) or an `embeddings` attribute (`layer_embedding` layer).
`power_iterations`	int, the number of iterations during normalization.
`...`	Base wrapper keyword arguments.

Value

The return value depends on the value provided for the first argument. If `object` is:

- a `keras_model_sequential()`, then the layer is added to the sequential model (which is modified in place). To enable piping, the sequential model is also returned, invisibly.
- a `keras_input()`, then the output tensor from calling `layer(input)` is returned.
- `NULL` or missing, then a `Layer` instance is returned.

Examples

Wrap layer_conv_2d:

```
x <- random_uniform(c(1, 10, 10, 1))
conv2d <- layer_spectral_normalization(
  layer = layer_conv_2d(filters = 2, kernel_size = 2)
)
y <- conv2d(x)
shape(y)

## shape(1, 9, 9, 2)
```

Wrap layer_dense:

```
x <- random_uniform(c(1, 10, 10, 1))
dense <- layer_spectral_normalization(layer = layer_dense(units = 10))
y <- dense(x)
shape(y)

## shape(1, 10, 10, 10)
```

Reference

- Spectral Normalization for GAN.

See Also

Other normalization layers:
layer_batch_normalization()
layer_group_normalization()
layer_layer_normalization()
layer_unit_normalization()

Other layers:
Layer()
layer_activation()
layer_activation_elu()
layer_activation_leaky_relu()
layer_activation_parametric_relu()
layer_activation_relu()
layer_activation_softmax()
layer_activity_regularization()
layer_add()
layer_additive_attention()
layer_alpha_dropout()
layer_attention()

layer_average()
layer_average_pooling_1d()
layer_average_pooling_2d()
layer_average_pooling_3d()
layer_batch_normalization()
layer_bidirectional()
layer_category_encoding()
layer_center_crop()
layer_concatenate()
layer_conv_1d()
layer_conv_1d_transpose()
layer_conv_2d()
layer_conv_2d_transpose()
layer_conv_3d()
layer_conv_3d_transpose()
layer_conv_lstm_1d()
layer_conv_lstm_2d()
layer_conv_lstm_3d()
layer_cropping_1d()
layer_cropping_2d()
layer_cropping_3d()
layer_dense()
layer_depthwise_conv_1d()
layer_depthwise_conv_2d()
layer_discretization()
layer_dot()
layer_dropout()
layer_einsum_dense()
layer_embedding()
layer_feature_space()
layer_flatten()
layer_flax_module_wrapper()
layer_gaussian_dropout()
layer_gaussian_noise()
layer_global_average_pooling_1d()
layer_global_average_pooling_2d()
layer_global_average_pooling_3d()
layer_global_max_pooling_1d()
layer_global_max_pooling_2d()
layer_global_max_pooling_3d()
layer_group_normalization()
layer_group_query_attention()
layer_gru()
layer_hashed_crossing()
layer_hashing()
layer_identity()
layer_integer_lookup()
layer_jax_model_wrapper()

layer_lambda()
layer_layer_normalization()
layer_lstm()
layer_masking()
layer_max_pooling_1d()
layer_max_pooling_2d()
layer_max_pooling_3d()
layer_maximum()
layer_mel_spectrogram()
layer_minimum()
layer_multi_head_attention()
layer_multiply()
layer_normalization()
layer_permute()
layer_random_brightness()
layer_random_contrast()
layer_random_crop()
layer_random_flip()
layer_random_rotation()
layer_random_translation()
layer_random_zoom()
layer_repeat_vector()
layer_rescaling()
layer_reshape()
layer_resizing()
layer_rnn()
layer_separable_conv_1d()
layer_separable_conv_2d()
layer_simple_rnn()
layer_spatial_dropout_1d()
layer_spatial_dropout_2d()
layer_spatial_dropout_3d()
layer_string_lookup()
layer_subtract()
layer_text_vectorization()
layer_tfsm()
layer_time_distributed()
layer_torch_module_wrapper()
layer_unit_normalization()
layer_upsampling_1d()
layer_upsampling_2d()
layer_upsampling_3d()
layer_zero_padding_1d()
layer_zero_padding_2d()
layer_zero_padding_3d()
rnn_cell_gru()
rnn_cell_lstm()
rnn_cell_simple()

rnn_cells_stack()

`layer_string_lookup` *A preprocessing layer that maps strings to (possibly encoded) indices.*

Description

This layer translates a set of arbitrary strings into integer output via a table-based vocabulary lookup. This layer will perform no splitting or transformation of input strings. For a layer than can split and tokenize natural language, see the `layer_text_vectorization` layer.

The vocabulary for the layer must be either supplied on construction or learned via `adapt()`. During `adapt()`, the layer will analyze a data set, determine the frequency of individual strings tokens, and create a vocabulary from them. If the vocabulary is capped in size, the most frequent tokens will be used to create the vocabulary and all others will be treated as out-of-vocabulary (OOV).

There are two possible output modes for the layer. When `output_mode` is `"int"`, input strings are converted to their index in the vocabulary (an integer). When `output_mode` is `"multi_hot"`, `"count"`, or `"tf_idf"`, input strings are encoded into an array where each dimension corresponds to an element in the vocabulary.

The vocabulary can optionally contain a mask token as well as an OOV token (which can optionally occupy multiple indices in the vocabulary, as set by `num_oov_indices`). The position of these tokens in the vocabulary is fixed. When `output_mode` is `"int"`, the vocabulary will begin with the mask token (if set), followed by OOV indices, followed by the rest of the vocabulary. When `output_mode` is `"multi_hot"`, `"count"`, or `"tf_idf"` the vocabulary will begin with OOV indices and instances of the mask token will be dropped.

Note: This layer uses TensorFlow internally. It cannot be used as part of the compiled computation graph of a model with any backend other than TensorFlow. It can however be used with any backend when running eagerly. It can also always be used as part of an input preprocessing pipeline with any backend (outside the model itself), which is how we recommend to use this layer.

Note: This layer is safe to use inside a `tf.data` pipeline (independently of which backend you're using).

Usage

```
layer_string_lookup(
  object,
  max_tokens = NULL,
  num_oov_indices = 1L,
  mask_token = NULL,
  oov_token = "[UNK]",
  vocabulary = NULL,
  idf_weights = NULL,
  invert = FALSE,
  output_mode = "int",
  pad_to_max_tokens = FALSE,
  sparse = FALSE,
  encoding = "utf-8",
  name = NULL,
  ...,
  vocabulary_dtype = NULL
)
```

Arguments

object	Object to compose the layer with. A tensor, array, or sequential model.
max_tokens	Maximum size of the vocabulary for this layer. This should only be specified when adapting the vocabulary or when setting pad_to_max_tokens=TRUE. If NULL, there is no cap on the size of the vocabulary. Note that this size includes the OOV and mask tokens. Defaults to NULL.
num_oov_indices	The number of out-of-vocabulary tokens to use. If this value is more than 1, OOV inputs are modulated to determine their OOV value. If this value is 0, OOV inputs will cause an error when calling the layer. Defaults to 1.
mask_token	A token that represents masked inputs. When output_mode is "int", the token is included in vocabulary and mapped to index 0. In other output modes, the token will not appear in the vocabulary and instances of the mask token in the input will be dropped. If set to NULL, no mask term will be added. Defaults to NULL.
oov_token	Only used when invert is TRUE. The token to return for OOV indices. Defaults to "[UNK]".
vocabulary	Optional. Either an array of integers or a string path to a text file. If passing an array, can pass a list, list, 1D NumPy array, or 1D tensor containing the integer vocbulary terms. If passing a file path, the file should contain one line per term in the vocabulary. If this argument is set, there is no need to adapt() the layer.
idf_weights	Only valid when output_mode is "tf_idf". A list, list, 1D NumPy array, or 1D tensor or the same length as the vocabulary, containing the floating point inverse document frequency weights, which will be multiplied by per sample term counts for the final TF-IDF weight. If the vocabulary argument is set, and output_mode is "tf_idf", this argument must be supplied.
invert	Only valid when output_mode is "int". If TRUE, this layer will map indices to vocabulary items instead of mapping vocabulary items to indices. Defaults to FALSE.
output_mode	Specification for the output of the layer. Values can be "int", "one_hot", "multi_hot", "count", or "tf_idf" configuring the layer as follows: • "int": Return the vocabulary indices of the input tokens. • "one_hot": Encodes each individual element in the input into an array the same size as the vocabulary, containing a 1 at the element index. If the last dimension is size 1, will encode on that dimension. If the last dimension is not size 1, will append a new dimension for the encoded output. • "multi_hot": Encodes each sample in the input into a single array the same size as the vocabulary, containing a 1 for each vocabulary term present

in the sample. Treats the last dimension as the sample dimension, if input shape is `(..., sample_length)`, output shape will be `(..., num_tokens)`.
- `"count"`: As `"multi_hot"`, but the int array contains a count of the number of times the token at that index appeared in the sample.
- `"tf_idf"`: As `"multi_hot"`, but the TF-IDF algorithm is applied to find the value in each token slot. For `"int"` output, any shape of input and output is supported. For all other output modes, currently only output up to rank 2 is supported. Defaults to `"int"`.

`pad_to_max_tokens`
Only applicable when `output_mode` is `"multi_hot"`, `"count"`, or `"tf_idf"`. If `TRUE`, the output will have its feature axis padded to `max_tokens` even if the number of unique tokens in the vocabulary is less than `max_tokens`, resulting in a tensor of shape `(batch_size, max_tokens)` regardless of vocabulary size. Defaults to `FALSE`.

`sparse` Boolean. Only applicable to `"multi_hot"`, `"count"`, and `"tf_idf"` output modes. Only supported with TensorFlow backend. If `TRUE`, returns a `SparseTensor` instead of a dense `Tensor`. Defaults to `FALSE`.

`encoding` Optional. The text encoding to use to interpret the input strings. Defaults to `"utf-8"`.

`name` String, name for the object

`...` For forward/backward compatability.

`vocabulary_dtype`
The dtype of the vocabulary terms, for example `"int64"` or `"int32"`. Defaults to `"int64"`.

Value

The return value depends on the value provided for the first argument. If `object` is:

- a `keras_model_sequential()`, then the layer is added to the sequential model (which is modified in place). To enable piping, the sequential model is also returned, invisibly.
- a `keras_input()`, then the output tensor from calling `layer(input)` is returned.
- `NULL` or missing, then a `Layer` instance is returned.

Examples

Creating a lookup layer with a known vocabulary

This example creates a lookup layer with a pre-existing vocabulary.

```
vocab <- c("a", "b", "c", "d")
data <- rbind(c("a", "c", "d"), c("d", "z", "b"))
layer <- layer_string_lookup(vocabulary=vocab)
layer(data)

## tf.Tensor(
## [[1 3 4]
##  [4 0 2]], shape=(2, 3), dtype=int64)
```

Creating a lookup layer with an adapted vocabulary

This example creates a lookup layer and generates the vocabulary by analyzing the dataset.

```
data <- rbind(c("a", "c", "d"), c("d", "z", "b"))
layer <- layer_string_lookup()
layer %>% adapt(data)
get_vocabulary(layer)
```

```
## [1] "[UNK]" "d"     "z"     "c"     "b"     "a"
```

Note that the OOV token `"[UNK]"` has been added to the vocabulary. The remaining tokens are sorted by frequency (`"d"`, which has 2 occurrences, is first) then by inverse sort order.

```
data <- rbind(c("a", "c", "d"), c("d", "z", "b"))
layer <- layer_string_lookup()
layer %>% adapt(data)
layer(data)
```

```
## tf.Tensor(
## [[5 3 1]
##  [1 2 4]], shape=(2, 3), dtype=int64)
```

Lookups with multiple OOV indices

This example demonstrates how to use a lookup layer with multiple OOV indices. When a layer is created with more than one OOV index, any OOV values are hashed into the number of OOV buckets, distributing OOV values in a deterministic fashion across the set.

```
vocab <- c("a", "b", "c", "d")
data <- rbind(c("a", "c", "d"), c("m", "z", "b"))
layer <- layer_string_lookup(vocabulary = vocab, num_oov_indices = 2)
layer(data)
```

```
## tf.Tensor(
## [[2 4 5]
##  [0 1 3]], shape=(2, 3), dtype=int64)
```

Note that the output for OOV value 'm' is 0, while the output for OOV value `"z"` is 1. The in-vocab terms have their output index increased by 1 from earlier examples (a maps to 2, etc) in order to make space for the extra OOV value.

One-hot output

Configure the layer with `output_mode='one_hot'`. Note that the first `num_oov_indices` dimensions in the ont_hot encoding represent OOV values.

```
vocab <- c("a", "b", "c", "d")
data <- c("a", "b", "c", "d", "z")
layer <- layer_string_lookup(vocabulary = vocab, output_mode = 'one_hot')
layer(data)

## tf.Tensor(
## [[0 1 0 0 0]
##  [0 0 1 0 0]
##  [0 0 0 1 0]
##  [0 0 0 0 1]
##  [1 0 0 0 0]], shape=(5, 5), dtype=int64)
```

Multi-hot output

Configure the layer with `output_mode='multi_hot'`. Note that the first `num_oov_indices` dimensions in the multi_hot encoding represent OOV values.

```
vocab <- c("a", "b", "c", "d")
data <- rbind(c("a", "c", "d", "d"), c("d", "z", "b", "z"))
layer <- layer_string_lookup(vocabulary = vocab, output_mode = 'multi_hot')
layer(data)

## tf.Tensor(
## [[0 1 0 1 1]
##  [1 0 1 0 1]], shape=(2, 5), dtype=int64)
```

Token count output

Configure the layer with `output_mode='count'`. As with multi_hot output, the first `num_oov_indices` dimensions in the output represent OOV values.

```
vocab <- c("a", "b", "c", "d")
data <- rbind(c("a", "c", "d", "d"), c("d", "z", "b", "z"))
layer <- layer_string_lookup(vocabulary = vocab, output_mode = 'count')
layer(data)

## tf.Tensor(
## [[0 1 0 1 2]
##  [2 0 1 0 1]], shape=(2, 5), dtype=int64)
```

TF-IDF output

Configure the layer with `output_mode="tf_idf"`. As with multi_hot output, the first `num_oov_indices` dimensions in the output represent OOV values.

Each token bin will output `token_count * idf_weight`, where the idf weights are the inverse document frequency weights per token. These should be provided along with the vocabulary. Note that the `idf_weight` for OOV values will default to the average of all idf weights passed in.

```
vocab <- c("a", "b", "c", "d")
idf_weights <- c(0.25, 0.75, 0.6, 0.4)
data <- rbind(c("a", "c", "d", "d"), c("d", "z", "b", "z"))
layer <- layer_string_lookup(output_mode = "tf_idf")
layer %>% set_vocabulary(vocab, idf_weights=idf_weights)
layer(data)

## tf.Tensor(
## [[0.   0.25 0.   0.6  0.8 ]
##  [1.   0.   0.75 0.   0.4 ]], shape=(2, 5), dtype=float32)
```

To specify the idf weights for oov values, you will need to pass the entire vocabulary including the leading oov token.

```
vocab <- c("[UNK]", "a", "b", "c", "d")
idf_weights <- c(0.9, 0.25, 0.75, 0.6, 0.4)
data <- rbind(c("a", "c", "d", "d"), c("d", "z", "b", "z"))
layer <- layer_string_lookup(output_mode = "tf_idf")
layer %>% set_vocabulary(vocab, idf_weights=idf_weights)
layer(data)

## tf.Tensor(
## [[0.   0.25 0.   0.6  0.8 ]
##  [1.8  0.   0.75 0.   0.4 ]], shape=(2, 5), dtype=float32)
```

When adapting the layer in `"tf_idf"` mode, each input sample will be considered a document, and IDF weight per token will be calculated as `log(1 + num_documents / (1 + token_document_count))`.

Inverse lookup

This example demonstrates how to map indices to strings using this layer. (You can also use `adapt()` with `inverse=TRUE`, but for simplicity we'll pass the vocab in this example.)

```
vocab <- c("a", "b", "c", "d")
data <- rbind(c(1, 3, 4), c(4, 0, 2))
layer <- layer_string_lookup(vocabulary = vocab, invert = TRUE)
layer(data)

## tf.Tensor(
## [[b'a' b'c' b'd']
##  [b'd' b'[UNK]' b'b']], shape=(2, 3), dtype=string)
```

Note that the first index correspond to the oov token by default.

Forward and inverse lookup pairs

This example demonstrates how to use the vocabulary of a standard lookup layer to create an inverse lookup layer.

```
vocab <- c("a", "b", "c", "d")
data <- rbind(c("a", "c", "d"), c("d", "z", "b"))
layer <- layer_string_lookup(vocabulary = vocab)
i_layer <- layer_string_lookup(vocabulary = vocab, invert = TRUE)
int_data <- layer(data)
i_layer(int_data)
```

```
## tf.Tensor(
## [[b'a' b'c' b'd']
##  [b'd' b'[UNK]' b'b']], shape=(2, 3), dtype=string)
```

In this example, the input value `"z"` resulted in an output of `"[UNK]"`, since 1000 was not in the vocabulary - it got represented as an OOV, and all OOV values are returned as `"[UNK]"` in the inverse layer. Also, note that for the inverse to work, you must have already set the forward layer vocabulary either directly or via `adapt()` before calling `get_vocabulary()`.

See Also

- https://keras.io/api/layers/preprocessing_layers/categorical/string_lookup#stringlookup-class

Other categorical features preprocessing layers:
`layer_category_encoding()`
`layer_hashed_crossing()`
`layer_hashing()`
`layer_integer_lookup()`

Other preprocessing layers:
`layer_category_encoding()`
`layer_center_crop()`
`layer_discretization()`
`layer_feature_space()`
`layer_hashed_crossing()`
`layer_hashing()`
`layer_integer_lookup()`
`layer_mel_spectrogram()`
`layer_normalization()`
`layer_random_brightness()`
`layer_random_contrast()`
`layer_random_crop()`
`layer_random_flip()`
`layer_random_rotation()`
`layer_random_translation()`
`layer_random_zoom()`
`layer_rescaling()`
`layer_resizing()`
`layer_text_vectorization()`

Other layers:
Layer()
layer_activation()
layer_activation_elu()
layer_activation_leaky_relu()
layer_activation_parametric_relu()
layer_activation_relu()
layer_activation_softmax()
layer_activity_regularization()
layer_add()
layer_additive_attention()
layer_alpha_dropout()
layer_attention()
layer_average()
layer_average_pooling_1d()
layer_average_pooling_2d()
layer_average_pooling_3d()
layer_batch_normalization()
layer_bidirectional()
layer_category_encoding()
layer_center_crop()
layer_concatenate()
layer_conv_1d()
layer_conv_1d_transpose()
layer_conv_2d()
layer_conv_2d_transpose()
layer_conv_3d()
layer_conv_3d_transpose()
layer_conv_lstm_1d()
layer_conv_lstm_2d()
layer_conv_lstm_3d()
layer_cropping_1d()
layer_cropping_2d()
layer_cropping_3d()
layer_dense()
layer_depthwise_conv_1d()
layer_depthwise_conv_2d()
layer_discretization()
layer_dot()
layer_dropout()
layer_einsum_dense()
layer_embedding()
layer_feature_space()
layer_flatten()
layer_flax_module_wrapper()
layer_gaussian_dropout()
layer_gaussian_noise()
layer_global_average_pooling_1d()

layer_global_average_pooling_2d()
layer_global_average_pooling_3d()
layer_global_max_pooling_1d()
layer_global_max_pooling_2d()
layer_global_max_pooling_3d()
layer_group_normalization()
layer_group_query_attention()
layer_gru()
layer_hashed_crossing()
layer_hashing()
layer_identity()
layer_integer_lookup()
layer_jax_model_wrapper()
layer_lambda()
layer_layer_normalization()
layer_lstm()
layer_masking()
layer_max_pooling_1d()
layer_max_pooling_2d()
layer_max_pooling_3d()
layer_maximum()
layer_mel_spectrogram()
layer_minimum()
layer_multi_head_attention()
layer_multiply()
layer_normalization()
layer_permute()
layer_random_brightness()
layer_random_contrast()
layer_random_crop()
layer_random_flip()
layer_random_rotation()
layer_random_translation()
layer_random_zoom()
layer_repeat_vector()
layer_rescaling()
layer_reshape()
layer_resizing()
layer_rnn()
layer_separable_conv_1d()
layer_separable_conv_2d()
layer_simple_rnn()
layer_spatial_dropout_1d()
layer_spatial_dropout_2d()
layer_spatial_dropout_3d()
layer_spectral_normalization()
layer_subtract()
layer_text_vectorization()

layer_tfsm()
layer_time_distributed()
layer_torch_module_wrapper()
layer_unit_normalization()
layer_upsampling_1d()
layer_upsampling_2d()
layer_upsampling_3d()
layer_zero_padding_1d()
layer_zero_padding_2d()
layer_zero_padding_3d()
rnn_cell_gru()
rnn_cell_lstm()
rnn_cell_simple()
rnn_cells_stack()

`layer_subtract` *Performs elementwise subtraction.*

Description

It takes as input a list of tensors of size 2 both of the same shape, and returns a single tensor (`inputs[0] - inputs[1]`) of same shape.

Usage

```
layer_subtract(inputs, ...)
```

Arguments

`inputs`	layers to combine
`...`	For forward/backward compatability.

Value

The return value depends on the value provided for the first argument. If `object` is:

- a `keras_model_sequential()`, then the layer is added to the sequential model (which is modified in place). To enable piping, the sequential model is also returned, invisibly.
- a `keras_input()`, then the output tensor from calling `layer(input)` is returned.
- `NULL` or missing, then a `Layer` instance is returned.

Examples

```
input_shape <- c(2, 3, 4)
x1 <- random_uniform(input_shape)
x2 <- random_uniform(input_shape)
y <- layer_subtract(list(x1, x2))
```

Usage in a Keras model:

```
input1 <- layer_input(shape = 16)
x1 <- layer_dense(input1, units = 8, activation = 'relu')
input2 <- layer_input(shape = 32)
x2 <- layer_dense(input2, units = 8, activation = 'relu')
subtracted <- layer_subtract(list(x1, x2))
out <- layer_dense(subtracted, units = 4)
model <- keras_model(inputs = list(input1, input2), outputs = out)
```

See Also

- https://keras.io/api/layers/merging_layers/subtract#subtract-class

Other merging layers:
layer_add()
layer_average()
layer_concatenate()
layer_dot()
layer_maximum()
layer_minimum()
layer_multiply()

Other layers:
Layer()
layer_activation()
layer_activation_elu()
layer_activation_leaky_relu()
layer_activation_parametric_relu()
layer_activation_relu()
layer_activation_softmax()
layer_activity_regularization()
layer_add()
layer_additive_attention()
layer_alpha_dropout()
layer_attention()
layer_average()
layer_average_pooling_1d()
layer_average_pooling_2d()
layer_average_pooling_3d()
layer_batch_normalization()
layer_bidirectional()
layer_category_encoding()

layer_center_crop()
layer_concatenate()
layer_conv_1d()
layer_conv_1d_transpose()
layer_conv_2d()
layer_conv_2d_transpose()
layer_conv_3d()
layer_conv_3d_transpose()
layer_conv_lstm_1d()
layer_conv_lstm_2d()
layer_conv_lstm_3d()
layer_cropping_1d()
layer_cropping_2d()
layer_cropping_3d()
layer_dense()
layer_depthwise_conv_1d()
layer_depthwise_conv_2d()
layer_discretization()
layer_dot()
layer_dropout()
layer_einsum_dense()
layer_embedding()
layer_feature_space()
layer_flatten()
layer_flax_module_wrapper()
layer_gaussian_dropout()
layer_gaussian_noise()
layer_global_average_pooling_1d()
layer_global_average_pooling_2d()
layer_global_average_pooling_3d()
layer_global_max_pooling_1d()
layer_global_max_pooling_2d()
layer_global_max_pooling_3d()
layer_group_normalization()
layer_group_query_attention()
layer_gru()
layer_hashed_crossing()
layer_hashing()
layer_identity()
layer_integer_lookup()
layer_jax_model_wrapper()
layer_lambda()
layer_layer_normalization()
layer_lstm()
layer_masking()
layer_max_pooling_1d()
layer_max_pooling_2d()
layer_max_pooling_3d()

layer_maximum()
layer_mel_spectrogram()
layer_minimum()
layer_multi_head_attention()
layer_multiply()
layer_normalization()
layer_permute()
layer_random_brightness()
layer_random_contrast()
layer_random_crop()
layer_random_flip()
layer_random_rotation()
layer_random_translation()
layer_random_zoom()
layer_repeat_vector()
layer_rescaling()
layer_reshape()
layer_resizing()
layer_rnn()
layer_separable_conv_1d()
layer_separable_conv_2d()
layer_simple_rnn()
layer_spatial_dropout_1d()
layer_spatial_dropout_2d()
layer_spatial_dropout_3d()
layer_spectral_normalization()
layer_string_lookup()
layer_text_vectorization()
layer_tfsm()
layer_time_distributed()
layer_torch_module_wrapper()
layer_unit_normalization()
layer_upsampling_1d()
layer_upsampling_2d()
layer_upsampling_3d()
layer_zero_padding_1d()
layer_zero_padding_2d()
layer_zero_padding_3d()
rnn_cell_gru()
rnn_cell_lstm()
rnn_cell_simple()
rnn_cells_stack()

---

layer_text_vectorization

A preprocessing layer which maps text features to integer sequences.

Description

This layer has basic options for managing text in a Keras model. It transforms a batch of strings (one example = one string) into either a list of token indices (one example = 1D tensor of integer token indices) or a dense representation (one example = 1D tensor of float values representing data about the example's tokens). This layer is meant to handle natural language inputs. To handle simple string inputs (categorical strings or pre-tokenized strings) see `layer_string_lookup()`.

The vocabulary for the layer must be either supplied on construction or learned via `adapt()`. When this layer is adapted, it will analyze the dataset, determine the frequency of individual string values, and create a vocabulary from them. This vocabulary can have unlimited size or be capped, depending on the configuration options for this layer; if there are more unique values in the input than the maximum vocabulary size, the most frequent terms will be used to create the vocabulary.

The processing of each example contains the following steps:

1. Standardize each example (usually lowercasing + punctuation stripping)
2. Split each example into substrings (usually words)
3. Recombine substrings into tokens (usually ngrams)
4. Index tokens (associate a unique int value with each token)
5. Transform each example using this index, either into a vector of ints or a dense float vector.

Some notes on passing callables to customize splitting and normalization for this layer:

1. Any callable can be passed to this Layer, but if you want to serialize this object you should only pass functions that are registered Keras serializables (see `register_keras_serializable()` for more details).
2. When using a custom callable for `standardize`, the data received by the callable will be exactly as passed to this layer. The callable should return a tensor of the same shape as the input.
3. When using a custom callable for `split`, the data received by the callable will have the 1st dimension squeezed out - instead of `list("string to split", "another string to split")`, the Callable will see `c("string to split", "another string to split")`. The callable should return a `tf.Tensor` of dtype `string` with the first dimension containing the split tokens - in this example, we should see something like `list(c("string", "to", "split"), c("another", "string", "to", "split"))`.

Note: This layer uses TensorFlow internally. It cannot be used as part of the compiled computation graph of a model with any backend other than TensorFlow. It can however be used with any backend when running eagerly. It can also always be used as part of an input preprocessing pipeline with any backend (outside the model itself), which is how we recommend to use this layer.

Note: This layer is safe to use inside a `tf.data` pipeline (independently of which backend you're using).

Usage

```
layer_text_vectorization(
  object,
  max_tokens = NULL,
  standardize = "lower_and_strip_punctuation",
  split = "whitespace",
  ngrams = NULL,
  output_mode = "int",
  output_sequence_length = NULL,
  pad_to_max_tokens = FALSE,
  vocabulary = NULL,
  idf_weights = NULL,
  sparse = FALSE,
  ragged = FALSE,
  encoding = "utf-8",
  name = NULL,
  ...
)

get_vocabulary(object, include_special_tokens = TRUE)

set_vocabulary(object, vocabulary, idf_weights = NULL, ...)
```

Arguments

`object`	Object to compose the layer with. A tensor, array, or sequential model.
`max_tokens`	Maximum size of the vocabulary for this layer. This should only be specified when adapting a vocabulary or when setting `pad_to_max_tokens=TRUE`. Note that this vocabulary contains 1 OOV token, so the effective number of tokens is `(max_tokens - 1 - (1 if output_mode == "int" else 0))`.
`standardize`	Optional specification for standardization to apply to the input text. Values can be: • `NULL`: No standardization. • `"lower_and_strip_punctuation"`: Text will be lowercased and all punctuation removed. • `"lower"`: Text will be lowercased. • `"strip_punctuation"`: All punctuation will be removed. • Callable: Inputs will passed to the callable function, which should be standardized and returned.
`split`	Optional specification for splitting the input text. Values can be: • `NULL`: No splitting. • `"whitespace"`: Split on whitespace. • `"character"`: Split on each unicode character. • Callable: Standardized inputs will passed to the callable function, which should be split and returned.
`ngrams`	Optional specification for ngrams to create from the possibly-split input text. Values can be `NULL`, an integer or list of integers; passing an integer will create ngrams up to that integer, and passing a list of integers will create ngrams for the specified values in the list. Passing `NULL` means that no ngrams will be created.
`output_mode`	Optional specification for the output of the layer. Values can be `"int"`, `"multi_hot"`, `"count"` or `"tf_idf"`, configuring the layer as follows:

- `"int"`: Outputs integer indices, one integer index per split string token. When `output_mode == "int"`, 0 is reserved for masked locations; this reduces the vocab size to `max_tokens - 2` instead of `max_tokens - 1`.
- `"multi_hot"`: Outputs a single int array per batch, of either vocab_size or max_tokens size, containing 1s in all elements where the token mapped to that index exists at least once in the batch item.
- `"count"`: Like `"multi_hot"`, but the int array contains a count of the number of times the token at that index appeared in the batch item.
- `"tf_idf"`: Like `"multi_hot"`, but the TF-IDF algorithm is applied to find the value in each token slot. For `"int"` output, any shape of input and output is supported. For all other output modes, currently only rank 1 inputs (and rank 2 outputs after splitting) are supported.

`output_sequence_length`
: Only valid in INT mode. If set, the output will have its time dimension padded or truncated to exactly `output_sequence_length` values, resulting in a tensor of shape `(batch_size, output_sequence_length)` regardless of how many tokens resulted from the splitting step. Defaults to `NULL`. If `ragged` is `TRUE` then `output_sequence_length` may still truncate the output.

`pad_to_max_tokens`
: Only valid in `"multi_hot"`, `"count"`, and `"tf_idf"` modes. If `TRUE`, the output will have its feature axis padded to `max_tokens` even if the number of unique tokens in the vocabulary is less than `max_tokens`, resulting in a tensor of shape `(batch_size, max_tokens)` regardless of vocabulary size. Defaults to `FALSE`.

`vocabulary`
: Optional. Either an array of strings or a string path to a text file. If passing an array, can pass a list, list, 1D NumPy array, or 1D tensor containing the string vocabulary terms. If passing a file path, the file should contain one line per term in the vocabulary. If this argument is set, there is no need to `adapt()` the layer.

`idf_weights`
: An R vector, 1D numpy array, or 1D tensor of inverse document frequency weights with equal length to vocabulary. Must be set if output_mode is "tf_idf". Should not be set otherwise.

`sparse`
: Boolean. Only applicable to `"multi_hot"`, `"count"`, and `"tf_idf"` output modes. Only supported with TensorFlow backend. If `TRUE`, returns a `SparseTensor` instead of a dense `Tensor`. Defaults to `FALSE`.

`ragged`
: Boolean. Only applicable to `"int"` output mode. Only supported with TensorFlow backend. If `TRUE`, returns a `RaggedTensor` instead of a dense `Tensor`, where each sequence may have a different length after string splitting. Defaults to `FALSE`.

`encoding`
: Optional. The text encoding to use to interpret the input strings. Defaults to `"utf-8"`.

`name`
: String, name for the object

`...`
: For forward/backward compatability.

`include_special_tokens`
: If TRUE, the returned vocabulary will include the padding and OOV tokens, and a term's index in the vocabulary will equal the term's index when calling the layer. If FALSE, the returned vocabulary will not include any padding or OOV tokens.

Value

The return value depends on the value provided for the first argument. If `object` is:

- a `keras_model_sequential()`, then the layer is added to the sequential model (which is modified in place). To enable piping, the sequential model is also returned, invisibly.
- a `keras_input()`, then the output tensor from calling `layer(input)` is returned.
- `NULL` or missing, then a `Layer` instance is returned.

Examples

This example instantiates a `TextVectorization` layer that lowercases text, splits on whitespace, strips punctuation, and outputs integer vocab indices.

```
max_tokens <- 5000  # Maximum vocab size.
max_len <- 4  # Sequence length to pad the outputs to.
# Create the layer.
vectorize_layer <- layer_text_vectorization(
    max_tokens = max_tokens,
    output_mode = 'int',
    output_sequence_length = max_len)

# Now that the vocab layer has been created, call `adapt` on the
# list of strings to create the vocabulary.
vectorize_layer %>% adapt(c("foo bar", "bar baz", "baz bada boom"))

# Now, the layer can map strings to integers -- you can use an
# embedding layer to map these integers to learned embeddings.
input_data <- rbind("foo qux bar", "qux baz")
vectorize_layer(input_data)

## tf.Tensor(
## [[4 1 3 0]
##  [1 2 0 0]], shape=(2, 4), dtype=int64)
```

This example instantiates a `TextVectorization` layer by passing a list of vocabulary terms to the layer's `initialize()` method.

```
vocab_data <- c("earth", "wind", "and", "fire")
max_len <- 4  # Sequence length to pad the outputs to.
# Create the layer, passing the vocab directly. You can also pass the
# vocabulary arg a path to a file containing one vocabulary word per
# line.
vectorize_layer <- layer_text_vectorization(
    max_tokens = max_tokens,
    output_mode = 'int',
    output_sequence_length = max_len,
    vocabulary = vocab_data)
```

```
# Because we've passed the vocabulary directly, we don't need to adapt
# the layer - the vocabulary is already set. The vocabulary contains the
# padding token ('') and OOV token ('[UNK]')
# as well as the passed tokens.
vectorize_layer %>% get_vocabulary()

## [1] ""      "[UNK]" "earth" "wind"  "and"   "fire"


# ['', '[UNK]', 'earth', 'wind', 'and', 'fire']
```

See Also

- https://keras.io/api/layers/preprocessing_layers/text/text_vectorization#textvectorization-clas

Other preprocessing layers:
layer_category_encoding()
layer_center_crop()
layer_discretization()
layer_feature_space()
layer_hashed_crossing()
layer_hashing()
layer_integer_lookup()
layer_mel_spectrogram()
layer_normalization()
layer_random_brightness()
layer_random_contrast()
layer_random_crop()
layer_random_flip()
layer_random_rotation()
layer_random_translation()
layer_random_zoom()
layer_rescaling()
layer_resizing()
layer_string_lookup()

Other layers:
Layer()
layer_activation()
layer_activation_elu()
layer_activation_leaky_relu()
layer_activation_parametric_relu()
layer_activation_relu()
layer_activation_softmax()
layer_activity_regularization()
layer_add()
layer_additive_attention()
layer_alpha_dropout()
layer_attention()

layer_average()
layer_average_pooling_1d()
layer_average_pooling_2d()
layer_average_pooling_3d()
layer_batch_normalization()
layer_bidirectional()
layer_category_encoding()
layer_center_crop()
layer_concatenate()
layer_conv_1d()
layer_conv_1d_transpose()
layer_conv_2d()
layer_conv_2d_transpose()
layer_conv_3d()
layer_conv_3d_transpose()
layer_conv_lstm_1d()
layer_conv_lstm_2d()
layer_conv_lstm_3d()
layer_cropping_1d()
layer_cropping_2d()
layer_cropping_3d()
layer_dense()
layer_depthwise_conv_1d()
layer_depthwise_conv_2d()
layer_discretization()
layer_dot()
layer_dropout()
layer_einsum_dense()
layer_embedding()
layer_feature_space()
layer_flatten()
layer_flax_module_wrapper()
layer_gaussian_dropout()
layer_gaussian_noise()
layer_global_average_pooling_1d()
layer_global_average_pooling_2d()
layer_global_average_pooling_3d()
layer_global_max_pooling_1d()
layer_global_max_pooling_2d()
layer_global_max_pooling_3d()
layer_group_normalization()
layer_group_query_attention()
layer_gru()
layer_hashed_crossing()
layer_hashing()
layer_identity()
layer_integer_lookup()
layer_jax_model_wrapper()

layer_lambda()
layer_layer_normalization()
layer_lstm()
layer_masking()
layer_max_pooling_1d()
layer_max_pooling_2d()
layer_max_pooling_3d()
layer_maximum()
layer_mel_spectrogram()
layer_minimum()
layer_multi_head_attention()
layer_multiply()
layer_normalization()
layer_permute()
layer_random_brightness()
layer_random_contrast()
layer_random_crop()
layer_random_flip()
layer_random_rotation()
layer_random_translation()
layer_random_zoom()
layer_repeat_vector()
layer_rescaling()
layer_reshape()
layer_resizing()
layer_rnn()
layer_separable_conv_1d()
layer_separable_conv_2d()
layer_simple_rnn()
layer_spatial_dropout_1d()
layer_spatial_dropout_2d()
layer_spatial_dropout_3d()
layer_spectral_normalization()
layer_string_lookup()
layer_subtract()
layer_tfsm()
layer_time_distributed()
layer_torch_module_wrapper()
layer_unit_normalization()
layer_upsampling_1d()
layer_upsampling_2d()
layer_upsampling_3d()
layer_zero_padding_1d()
layer_zero_padding_2d()
layer_zero_padding_3d()
rnn_cell_gru()
rnn_cell_lstm()
rnn_cell_simple()

rnn_cells_stack()

layer_tfsm — *Reload a Keras model/layer that was saved via* `export_savedmodel()`.

Description

Reload a Keras model/layer that was saved via `export_savedmodel()`.

Usage

```
layer_tfsm(
  object,
  filepath,
  call_endpoint = "serve",
  call_training_endpoint = NULL,
  trainable = TRUE,
  name = NULL,
  dtype = NULL
)
```

Arguments

`object`	Object to compose the layer with. A tensor, array, or sequential model.
`filepath`	string, the path to the SavedModel.
`call_endpoint`	Name of the endpoint to use as the `call()` method of the reloaded layer. If the SavedModel was created via `export_savedmodel()`, then the default endpoint name is `'serve'`. In other cases it may be named `'serving_default'`.
`call_training_endpoint`	see description
`trainable`	see description
`name`	String, name for the object
`dtype`	datatype (e.g., `"float32"`).

Value

The return value depends on the value provided for the first argument. If `object` is:

- a `keras_model_sequential()`, then the layer is added to the sequential model (which is modified in place). To enable piping, the sequential model is also returned, invisibly.
- a `keras_input()`, then the output tensor from calling `layer(input)` is returned.
- `NULL` or missing, then a `Layer` instance is returned.

Examples

```
model <- keras_model_sequential(input_shape = c(784)) |> layer_dense(10)
model |> export_savedmodel("path/to/artifact")

## Saved artifact at 'path/to/artifact'. The following endpoints are available:
##
## * Endpoint 'serve'
##  args_0 (POSITIONAL_ONLY): TensorSpec(shape=(None, 784), dtype=tf.float32, name='keras_tensor')
## Output Type:
##   TensorSpec(shape=(None, 10), dtype=tf.float32, name=None)
## Captures:
##   129813643620448: TensorSpec(shape=(), dtype=tf.resource, name=None)
##   129813643612000: TensorSpec(shape=(), dtype=tf.resource, name=None)


reloaded_layer <- layer_tfsm(filepath = "path/to/artifact")
input <- random_normal(c(2, 784))
output <- reloaded_layer(input)
stopifnot(all.equal(as.array(output), as.array(model(input))))
```

The reloaded object can be used like a regular Keras layer, and supports training/fine-tuning of its trainable weights. Note that the reloaded object retains none of the internal structure or custom methods of the original object – it's a brand new layer created around the saved function.

Limitations:

- Only call endpoints with a single `inputs` tensor argument (which may optionally be a named list/list of tensors) are supported. For endpoints with multiple separate input tensor arguments, consider subclassing `layer_tfsm` and implementing a `call()` method with a custom signature.
- If you need training-time behavior to differ from inference-time behavior (i.e. if you need the reloaded object to support a `training=TRUE` argument in `__call__()`), make sure that the training-time call function is saved as a standalone endpoint in the artifact, and provide its name to the `layer_tfsm` via the `call_training_endpoint` argument.

See Also

Other layers:
`Layer()`
`layer_activation()`
`layer_activation_elu()`
`layer_activation_leaky_relu()`
`layer_activation_parametric_relu()`
`layer_activation_relu()`
`layer_activation_softmax()`
`layer_activity_regularization()`
`layer_add()`
`layer_additive_attention()`
`layer_alpha_dropout()`

layer_attention()
layer_average()
layer_average_pooling_1d()
layer_average_pooling_2d()
layer_average_pooling_3d()
layer_batch_normalization()
layer_bidirectional()
layer_category_encoding()
layer_center_crop()
layer_concatenate()
layer_conv_1d()
layer_conv_1d_transpose()
layer_conv_2d()
layer_conv_2d_transpose()
layer_conv_3d()
layer_conv_3d_transpose()
layer_conv_lstm_1d()
layer_conv_lstm_2d()
layer_conv_lstm_3d()
layer_cropping_1d()
layer_cropping_2d()
layer_cropping_3d()
layer_dense()
layer_depthwise_conv_1d()
layer_depthwise_conv_2d()
layer_discretization()
layer_dot()
layer_dropout()
layer_einsum_dense()
layer_embedding()
layer_feature_space()
layer_flatten()
layer_flax_module_wrapper()
layer_gaussian_dropout()
layer_gaussian_noise()
layer_global_average_pooling_1d()
layer_global_average_pooling_2d()
layer_global_average_pooling_3d()
layer_global_max_pooling_1d()
layer_global_max_pooling_2d()
layer_global_max_pooling_3d()
layer_group_normalization()
layer_group_query_attention()
layer_gru()
layer_hashed_crossing()
layer_hashing()
layer_identity()
layer_integer_lookup()

layer_jax_model_wrapper()
layer_lambda()
layer_layer_normalization()
layer_lstm()
layer_masking()
layer_max_pooling_1d()
layer_max_pooling_2d()
layer_max_pooling_3d()
layer_maximum()
layer_mel_spectrogram()
layer_minimum()
layer_multi_head_attention()
layer_multiply()
layer_normalization()
layer_permute()
layer_random_brightness()
layer_random_contrast()
layer_random_crop()
layer_random_flip()
layer_random_rotation()
layer_random_translation()
layer_random_zoom()
layer_repeat_vector()
layer_rescaling()
layer_reshape()
layer_resizing()
layer_rnn()
layer_separable_conv_1d()
layer_separable_conv_2d()
layer_simple_rnn()
layer_spatial_dropout_1d()
layer_spatial_dropout_2d()
layer_spatial_dropout_3d()
layer_spectral_normalization()
layer_string_lookup()
layer_subtract()
layer_text_vectorization()
layer_time_distributed()
layer_torch_module_wrapper()
layer_unit_normalization()
layer_upsampling_1d()
layer_upsampling_2d()
layer_upsampling_3d()
layer_zero_padding_1d()
layer_zero_padding_2d()
layer_zero_padding_3d()
rnn_cell_gru()
rnn_cell_lstm()

rnn_cell_simple()
rnn_cells_stack()

Other saving and loading functions:
export_savedmodel.keras.src.models.model.Model()
load_model()
load_model_weights()
register_keras_serializable()
save_model()
save_model_config()
save_model_weights()
with_custom_object_scope()

---

layer_time_distributed

This wrapper allows to apply a layer to every temporal slice of an input.

Description

Every input should be at least 3D, and the dimension of index one of the first input will be considered to be the temporal dimension.

Consider a batch of 32 video samples, where each sample is a 128x128 RGB image with channels_last data format, across 10 timesteps. The batch input shape is (32, 10, 128, 128, 3).

You can then use TimeDistributed to apply the same Conv2D layer to each of the 10 timesteps, independently:

```
inputs <- keras_input(shape = c(10, 128, 128, 3), batch_size = 32)
conv_2d_layer <- layer_conv_2d(filters = 64, kernel_size = c(3, 3))
outputs <- layer_time_distributed(inputs, layer = conv_2d_layer)
shape(outputs)
```

```
## shape(32, 10, 126, 126, 64)
```

Because layer_time_distributed applies the same instance of layer_conv2d to each of the timestamps, the same set of weights are used at each timestamp.

Usage

```
layer_time_distributed(object, layer, ...)
```

Arguments

`object`	Object to compose the layer with. A tensor, array, or sequential model.
`layer`	A `Layer` instance.
`...`	For forward/backward compatability.

Value

The return value depends on the value provided for the first argument. If `object` is:

- a `keras_model_sequential()`, then the layer is added to the sequential model (which is modified in place). To enable piping, the sequential model is also returned, invisibly.
- a `keras_input()`, then the output tensor from calling `layer(input)` is returned.
- `NULL` or missing, then a `Layer` instance is returned.

Call Arguments

- `inputs`: Input tensor of shape (batch, time, ...) or nested tensors, and each of which has shape (batch, time, ...).
- `training`: Boolean indicating whether the layer should behave in training mode or in inference mode. This argument is passed to the wrapped layer (only if the layer supports this argument).
- `mask`: Binary tensor of shape `(samples, timesteps)` indicating whether a given timestep should be masked. This argument is passed to the wrapped layer (only if the layer supports this argument).

See Also

- https://keras.io/api/layers/recurrent_layers/time_distributed#timedistributed-class

Other rnn layers:
`layer_bidirectional()`
`layer_conv_lstm_1d()`
`layer_conv_lstm_2d()`
`layer_conv_lstm_3d()`
`layer_gru()`
`layer_lstm()`
`layer_rnn()`
`layer_simple_rnn()`
`rnn_cell_gru()`
`rnn_cell_lstm()`
`rnn_cell_simple()`
`rnn_cells_stack()`

Other layers:
`Layer()`
`layer_activation()`
`layer_activation_elu()`

layer_activation_leaky_relu()
layer_activation_parametric_relu()
layer_activation_relu()
layer_activation_softmax()
layer_activity_regularization()
layer_add()
layer_additive_attention()
layer_alpha_dropout()
layer_attention()
layer_average()
layer_average_pooling_1d()
layer_average_pooling_2d()
layer_average_pooling_3d()
layer_batch_normalization()
layer_bidirectional()
layer_category_encoding()
layer_center_crop()
layer_concatenate()
layer_conv_1d()
layer_conv_1d_transpose()
layer_conv_2d()
layer_conv_2d_transpose()
layer_conv_3d()
layer_conv_3d_transpose()
layer_conv_lstm_1d()
layer_conv_lstm_2d()
layer_conv_lstm_3d()
layer_cropping_1d()
layer_cropping_2d()
layer_cropping_3d()
layer_dense()
layer_depthwise_conv_1d()
layer_depthwise_conv_2d()
layer_discretization()
layer_dot()
layer_dropout()
layer_einsum_dense()
layer_embedding()
layer_feature_space()
layer_flatten()
layer_flax_module_wrapper()
layer_gaussian_dropout()
layer_gaussian_noise()
layer_global_average_pooling_1d()
layer_global_average_pooling_2d()
layer_global_average_pooling_3d()
layer_global_max_pooling_1d()
layer_global_max_pooling_2d()

layer_global_max_pooling_3d()
layer_group_normalization()
layer_group_query_attention()
layer_gru()
layer_hashed_crossing()
layer_hashing()
layer_identity()
layer_integer_lookup()
layer_jax_model_wrapper()
layer_lambda()
layer_layer_normalization()
layer_lstm()
layer_masking()
layer_max_pooling_1d()
layer_max_pooling_2d()
layer_max_pooling_3d()
layer_maximum()
layer_mel_spectrogram()
layer_minimum()
layer_multi_head_attention()
layer_multiply()
layer_normalization()
layer_permute()
layer_random_brightness()
layer_random_contrast()
layer_random_crop()
layer_random_flip()
layer_random_rotation()
layer_random_translation()
layer_random_zoom()
layer_repeat_vector()
layer_rescaling()
layer_reshape()
layer_resizing()
layer_rnn()
layer_separable_conv_1d()
layer_separable_conv_2d()
layer_simple_rnn()
layer_spatial_dropout_1d()
layer_spatial_dropout_2d()
layer_spatial_dropout_3d()
layer_spectral_normalization()
layer_string_lookup()
layer_subtract()
layer_text_vectorization()
layer_tfsm()
layer_torch_module_wrapper()
layer_unit_normalization()

`layer_upsampling_1d()`
`layer_upsampling_2d()`
`layer_upsampling_3d()`
`layer_zero_padding_1d()`
`layer_zero_padding_2d()`
`layer_zero_padding_3d()`
`rnn_cell_gru()`
`rnn_cell_lstm()`
`rnn_cell_simple()`
`rnn_cells_stack()`

`layer_torch_module_wrapper`

Torch module wrapper layer.

Description

`layer_torch_module_wrapper` is a wrapper class that can turn any `torch.nn.Module` into a Keras layer, in particular by making its parameters trackable by Keras.

Usage

```
layer_torch_module_wrapper(object, module, name = NULL, ...)
```

Arguments

`object`	Object to compose the layer with. A tensor, array, or sequential model.
`module`	`torch.nn.Module` instance. If it's a `LazyModule` instance, then its parameters must be initialized before passing the instance to `layer_torch_module_wrapper` (e.g. by calling it once).
`name`	The name of the layer (string).
`...`	For forward/backward compatability.

Value

The return value depends on the value provided for the first argument. If `object` is:

- a `keras_model_sequential()`, then the layer is added to the sequential model (which is modified in place). To enable piping, the sequential model is also returned, invisibly.
- a `keras_input()`, then the output tensor from calling `layer(input)` is returned.
- `NULL` or missing, then a `Layer` instance is returned.

Example

Here's an example of how the `layer_torch_module_wrapper()` can be used with vanilla PyTorch modules.

```
# reticulate::py_install(
#   packages = c("torch", "torchvision", "torchaudio"),
#   envname = "r-keras",
#   pip_options = c("--index-url https://download.pytorch.org/whl/cpu")
# )
library(keras3)
use_backend("torch")
torch <- reticulate::import("torch")
nn <- reticulate::import("torch.nn")
nnf <- reticulate::import("torch.nn.functional")

Classifier(keras$Model) \%py_class\% {
  initialize <- function(...) {
    super$initialize(...)

    self$conv1 <- layer_torch_module_wrapper(module = nn$Conv2d(
      in_channels = 1L,
      out_channels = 32L,
      kernel_size = tuple(3L, 3L)
    ))
    self$conv2 <- layer_torch_module_wrapper(module = nn$Conv2d(
      in_channels = 32L,
      out_channels = 64L,
      kernel_size = tuple(3L, 3L)
    ))
    self$pool <- nn$MaxPool2d(kernel_size = tuple(2L, 2L))
    self$flatten <- nn$Flatten()
    self$dropout <- nn$Dropout(p = 0.5)
    self$fc <-
      layer_torch_module_wrapper(module = nn$Linear(1600L, 10L))
  }

  call <- function(inputs) {
    x <- nnf$relu(self$conv1(inputs))
    x <- self$pool(x)
    x <- nnf$relu(self$conv2(x))
    x <- self$pool(x)
    x <- self$flatten(x)
    x <- self$dropout(x)
    x <- self$fc(x)
    nnf$softmax(x, dim = 1L)
  }
}
model <- Classifier()
```

```
model$build(shape(1, 28, 28))
cat("Output shape:", format(shape(model(torch$ones(1L, 1L, 28L, 28L)))))

model |> compile(loss = "sparse_categorical_crossentropy",
                 optimizer = "adam",
                 metrics = "accuracy")

model |> fit(train_loader, epochs = 5)
```

See Also

Other wrapping layers:
layer_flax_module_wrapper()
layer_jax_model_wrapper()

Other layers:
Layer()
layer_activation()
layer_activation_elu()
layer_activation_leaky_relu()
layer_activation_parametric_relu()
layer_activation_relu()
layer_activation_softmax()
layer_activity_regularization()
layer_add()
layer_additive_attention()
layer_alpha_dropout()
layer_attention()
layer_average()
layer_average_pooling_1d()
layer_average_pooling_2d()
layer_average_pooling_3d()
layer_batch_normalization()
layer_bidirectional()
layer_category_encoding()
layer_center_crop()
layer_concatenate()
layer_conv_1d()
layer_conv_1d_transpose()
layer_conv_2d()
layer_conv_2d_transpose()
layer_conv_3d()
layer_conv_3d_transpose()
layer_conv_lstm_1d()
layer_conv_lstm_2d()
layer_conv_lstm_3d()
layer_cropping_1d()
layer_cropping_2d()

layer_cropping_3d()
layer_dense()
layer_depthwise_conv_1d()
layer_depthwise_conv_2d()
layer_discretization()
layer_dot()
layer_dropout()
layer_einsum_dense()
layer_embedding()
layer_feature_space()
layer_flatten()
layer_flax_module_wrapper()
layer_gaussian_dropout()
layer_gaussian_noise()
layer_global_average_pooling_1d()
layer_global_average_pooling_2d()
layer_global_average_pooling_3d()
layer_global_max_pooling_1d()
layer_global_max_pooling_2d()
layer_global_max_pooling_3d()
layer_group_normalization()
layer_group_query_attention()
layer_gru()
layer_hashed_crossing()
layer_hashing()
layer_identity()
layer_integer_lookup()
layer_jax_model_wrapper()
layer_lambda()
layer_layer_normalization()
layer_lstm()
layer_masking()
layer_max_pooling_1d()
layer_max_pooling_2d()
layer_max_pooling_3d()
layer_maximum()
layer_mel_spectrogram()
layer_minimum()
layer_multi_head_attention()
layer_multiply()
layer_normalization()
layer_permute()
layer_random_brightness()
layer_random_contrast()
layer_random_crop()
layer_random_flip()
layer_random_rotation()
layer_random_translation()

layer_random_zoom()
layer_repeat_vector()
layer_rescaling()
layer_reshape()
layer_resizing()
layer_rnn()
layer_separable_conv_1d()
layer_separable_conv_2d()
layer_simple_rnn()
layer_spatial_dropout_1d()
layer_spatial_dropout_2d()
layer_spatial_dropout_3d()
layer_spectral_normalization()
layer_string_lookup()
layer_subtract()
layer_text_vectorization()
layer_tfsm()
layer_time_distributed()
layer_unit_normalization()
layer_upsampling_1d()
layer_upsampling_2d()
layer_upsampling_3d()
layer_zero_padding_1d()
layer_zero_padding_2d()
layer_zero_padding_3d()
rnn_cell_gru()
rnn_cell_lstm()
rnn_cell_simple()
rnn_cells_stack()

layer_unit_normalization

Unit normalization layer.

Description

Normalize a batch of inputs so that each input in the batch has a L2 norm equal to 1 (across the axes specified in axis).

Usage

```
layer_unit_normalization(object, axis = -1L, ...)
```

Arguments

object	Object to compose the layer with. A tensor, array, or sequential model.

axis	Integer or list. The axis or axes to normalize across. Typically, this is the features axis or axes. The left-out axes are typically the batch axis or axes. -1 is the last dimension in the input. Defaults to -1.
...	For forward/backward compatability.

Value

The return value depends on the value provided for the first argument. If object is:

- a keras_model_sequential(), then the layer is added to the sequential model (which is modified in place). To enable piping, the sequential model is also returned, invisibly.
- a keras_input(), then the output tensor from calling layer(input) is returned.
- NULL or missing, then a Layer instance is returned.

Examples

```
data <- op_reshape(1:6, newshape = c(2, 3))
normalized_data <- layer_unit_normalization(data)
op_sum(normalized_data[1,]^2)

## tf.Tensor(0.9999999, shape=(), dtype=float32)
```

See Also

- https://keras.io/api/layers/normalization_layers/unit_normalization#unitnormalization-class

Other normalization layers:
layer_batch_normalization()
layer_group_normalization()
layer_layer_normalization()
layer_spectral_normalization()

Other layers:
Layer()
layer_activation()
layer_activation_elu()
layer_activation_leaky_relu()
layer_activation_parametric_relu()
layer_activation_relu()
layer_activation_softmax()
layer_activity_regularization()
layer_add()
layer_additive_attention()
layer_alpha_dropout()
layer_attention()
layer_average()
layer_average_pooling_1d()
layer_average_pooling_2d()

layer_average_pooling_3d()
layer_batch_normalization()
layer_bidirectional()
layer_category_encoding()
layer_center_crop()
layer_concatenate()
layer_conv_1d()
layer_conv_1d_transpose()
layer_conv_2d()
layer_conv_2d_transpose()
layer_conv_3d()
layer_conv_3d_transpose()
layer_conv_lstm_1d()
layer_conv_lstm_2d()
layer_conv_lstm_3d()
layer_cropping_1d()
layer_cropping_2d()
layer_cropping_3d()
layer_dense()
layer_depthwise_conv_1d()
layer_depthwise_conv_2d()
layer_discretization()
layer_dot()
layer_dropout()
layer_einsum_dense()
layer_embedding()
layer_feature_space()
layer_flatten()
layer_flax_module_wrapper()
layer_gaussian_dropout()
layer_gaussian_noise()
layer_global_average_pooling_1d()
layer_global_average_pooling_2d()
layer_global_average_pooling_3d()
layer_global_max_pooling_1d()
layer_global_max_pooling_2d()
layer_global_max_pooling_3d()
layer_group_normalization()
layer_group_query_attention()
layer_gru()
layer_hashed_crossing()
layer_hashing()
layer_identity()
layer_integer_lookup()
layer_jax_model_wrapper()
layer_lambda()
layer_layer_normalization()
layer_lstm()

layer_masking()
layer_max_pooling_1d()
layer_max_pooling_2d()
layer_max_pooling_3d()
layer_maximum()
layer_mel_spectrogram()
layer_minimum()
layer_multi_head_attention()
layer_multiply()
layer_normalization()
layer_permute()
layer_random_brightness()
layer_random_contrast()
layer_random_crop()
layer_random_flip()
layer_random_rotation()
layer_random_translation()
layer_random_zoom()
layer_repeat_vector()
layer_rescaling()
layer_reshape()
layer_resizing()
layer_rnn()
layer_separable_conv_1d()
layer_separable_conv_2d()
layer_simple_rnn()
layer_spatial_dropout_1d()
layer_spatial_dropout_2d()
layer_spatial_dropout_3d()
layer_spectral_normalization()
layer_string_lookup()
layer_subtract()
layer_text_vectorization()
layer_tfsm()
layer_time_distributed()
layer_torch_module_wrapper()
layer_upsampling_1d()
layer_upsampling_2d()
layer_upsampling_3d()
layer_zero_padding_1d()
layer_zero_padding_2d()
layer_zero_padding_3d()
rnn_cell_gru()
rnn_cell_lstm()
rnn_cell_simple()
rnn_cells_stack()

layer_upsampling_1d *Upsampling layer for 1D inputs.*

Description

Repeats each temporal step `size` times along the time axis.

Usage

```
layer_upsampling_1d(object, size = 2L, ...)
```

Arguments

`object`	Object to compose the layer with. A tensor, array, or sequential model.
`size`	Integer. Upsampling factor.
`...`	For forward/backward compatability.

Value

The return value depends on the value provided for the first argument. If `object` is:

- a `keras_model_sequential()`, then the layer is added to the sequential model (which is modified in place). To enable piping, the sequential model is also returned, invisibly.
- a `keras_input()`, then the output tensor from calling `layer(input)` is returned.
- `NULL` or missing, then a `Layer` instance is returned.

Example

```
input_shape <- c(2, 2, 3)
x <- seq_len(prod(input_shape)) %>% op_reshape(input_shape)
x
```

```
## tf.Tensor(
## [[[ 1  2  3]
##   [ 4  5  6]]
##
##  [[ 7  8  9]
##   [10 11 12]]], shape=(2, 2, 3), dtype=int32)
```

```
y <- layer_upsampling_1d(x, size = 2)
y
```

```
## tf.Tensor(
## [[[ 1  2  3]
##   [ 1  2  3]
##   [ 4  5  6]
##   [ 4  5  6]]
##
##  [[ 7  8  9]
##   [ 7  8  9]
##   [10 11 12]
##   [10 11 12]]], shape=(2, 4, 3), dtype=int32)
```

Input Shape

3D tensor with shape: `(batch_size, steps, features)`.

Output Shape

3D tensor with shape: `(batch_size, upsampled_steps, features)`.

See Also

- https://keras.io/api/layers/reshaping_layers/up_sampling1d#upsampling1d-class

Other reshaping layers:
`layer_cropping_1d()`
`layer_cropping_2d()`
`layer_cropping_3d()`
`layer_flatten()`
`layer_permute()`
`layer_repeat_vector()`
`layer_reshape()`
`layer_upsampling_2d()`
`layer_upsampling_3d()`
`layer_zero_padding_1d()`
`layer_zero_padding_2d()`
`layer_zero_padding_3d()`

Other layers:
`Layer()`
`layer_activation()`
`layer_activation_elu()`
`layer_activation_leaky_relu()`
`layer_activation_parametric_relu()`
`layer_activation_relu()`
`layer_activation_softmax()`
`layer_activity_regularization()`
`layer_add()`
`layer_additive_attention()`

layer_alpha_dropout()
layer_attention()
layer_average()
layer_average_pooling_1d()
layer_average_pooling_2d()
layer_average_pooling_3d()
layer_batch_normalization()
layer_bidirectional()
layer_category_encoding()
layer_center_crop()
layer_concatenate()
layer_conv_1d()
layer_conv_1d_transpose()
layer_conv_2d()
layer_conv_2d_transpose()
layer_conv_3d()
layer_conv_3d_transpose()
layer_conv_lstm_1d()
layer_conv_lstm_2d()
layer_conv_lstm_3d()
layer_cropping_1d()
layer_cropping_2d()
layer_cropping_3d()
layer_dense()
layer_depthwise_conv_1d()
layer_depthwise_conv_2d()
layer_discretization()
layer_dot()
layer_dropout()
layer_einsum_dense()
layer_embedding()
layer_feature_space()
layer_flatten()
layer_flax_module_wrapper()
layer_gaussian_dropout()
layer_gaussian_noise()
layer_global_average_pooling_1d()
layer_global_average_pooling_2d()
layer_global_average_pooling_3d()
layer_global_max_pooling_1d()
layer_global_max_pooling_2d()
layer_global_max_pooling_3d()
layer_group_normalization()
layer_group_query_attention()
layer_gru()
layer_hashed_crossing()
layer_hashing()
layer_identity()

layer_integer_lookup()
layer_jax_model_wrapper()
layer_lambda()
layer_layer_normalization()
layer_lstm()
layer_masking()
layer_max_pooling_1d()
layer_max_pooling_2d()
layer_max_pooling_3d()
layer_maximum()
layer_mel_spectrogram()
layer_minimum()
layer_multi_head_attention()
layer_multiply()
layer_normalization()
layer_permute()
layer_random_brightness()
layer_random_contrast()
layer_random_crop()
layer_random_flip()
layer_random_rotation()
layer_random_translation()
layer_random_zoom()
layer_repeat_vector()
layer_rescaling()
layer_reshape()
layer_resizing()
layer_rnn()
layer_separable_conv_1d()
layer_separable_conv_2d()
layer_simple_rnn()
layer_spatial_dropout_1d()
layer_spatial_dropout_2d()
layer_spatial_dropout_3d()
layer_spectral_normalization()
layer_string_lookup()
layer_subtract()
layer_text_vectorization()
layer_tfsm()
layer_time_distributed()
layer_torch_module_wrapper()
layer_unit_normalization()
layer_upsampling_2d()
layer_upsampling_3d()
layer_zero_padding_1d()
layer_zero_padding_2d()
layer_zero_padding_3d()
rnn_cell_gru()

rnn_cell_lstm()
rnn_cell_simple()
rnn_cells_stack()

---

layer_upsampling_2d *Upsampling layer for 2D inputs.*

Description

The implementation uses interpolative resizing, given the resize method (specified by the `interpolation` argument). Use `interpolation=nearest` to repeat the rows and columns of the data.

Usage

```
layer_upsampling_2d(
  object,
  size = list(2L, 2L),
  data_format = NULL,
  interpolation = "nearest",
  ...
)
```

Arguments

`object`	Object to compose the layer with. A tensor, array, or sequential model.
`size`	Int, or list of 2 integers. The upsampling factors for rows and columns.
`data_format`	A string, one of `"channels_last"` (default) or `"channels_first"`. The ordering of the dimensions in the inputs. `"channels_last"` corresponds to inputs with shape `(batch_size, height, width, channels)` while `"channels_first"` corresponds to inputs with shape `(batch_size, channels, height, width)`. When unspecified, uses `image_data_format` value found in your Keras config file at `~/.keras/keras.json` (if exists) else `"channels_last"`. Defaults to `"channels_last"`.
`interpolation`	A string, one of `"bicubic"`, `"bilinear"`, `"lanczos3"`, `"lanczos5"`, `"nearest"`.
`...`	For forward/backward compatability.

Value

The return value depends on the value provided for the first argument. If `object` is:

- a `keras_model_sequential()`, then the layer is added to the sequential model (which is modified in place). To enable piping, the sequential model is also returned, invisibly.
- a `keras_input()`, then the output tensor from calling `layer(input)` is returned.
- `NULL` or missing, then a `Layer` instance is returned.

Example

```
input_shape <- c(2, 2, 1, 3)
x <- op_reshape(seq_len(prod(input_shape)), input_shape)
print(x)

## tf.Tensor(
## [[[[ 1  2  3]]
##
##   [[ 4  5  6]]]
##
##
##  [[[ 7  8  9]]
##
##   [[10 11 12]]]], shape=(2, 2, 1, 3), dtype=int32)


y <- layer_upsampling_2d(x, size = c(1, 2))
print(y)

## tf.Tensor(
## [[[[ 1  2  3]
##    [ 1  2  3]]
##
##   [[ 4  5  6]
##    [ 4  5  6]]]
##
##
##  [[[ 7  8  9]
##    [ 7  8  9]]
##
##   [[10 11 12]
##    [10 11 12]]]], shape=(2, 2, 2, 3), dtype=int32)
```

Input Shape

4D tensor with shape:

- If data_format is "channels_last": (batch_size, rows, cols, channels)
- If data_format is "channels_first": (batch_size, channels, rows, cols)

Output Shape

4D tensor with shape:

- If data_format is "channels_last": (batch_size, upsampled_rows, upsampled_cols, channels)
- If data_format is "channels_first": (batch_size, channels, upsampled_rows, upsampled_cols)

See Also

- https://keras.io/api/layers/reshaping_layers/up_sampling2d#upsampling2d-class

Other reshaping layers:
layer_cropping_1d()
layer_cropping_2d()
layer_cropping_3d()
layer_flatten()
layer_permute()
layer_repeat_vector()
layer_reshape()
layer_upsampling_1d()
layer_upsampling_3d()
layer_zero_padding_1d()
layer_zero_padding_2d()
layer_zero_padding_3d()

Other layers:
Layer()
layer_activation()
layer_activation_elu()
layer_activation_leaky_relu()
layer_activation_parametric_relu()
layer_activation_relu()
layer_activation_softmax()
layer_activity_regularization()
layer_add()
layer_additive_attention()
layer_alpha_dropout()
layer_attention()
layer_average()
layer_average_pooling_1d()
layer_average_pooling_2d()
layer_average_pooling_3d()
layer_batch_normalization()
layer_bidirectional()
layer_category_encoding()
layer_center_crop()
layer_concatenate()
layer_conv_1d()
layer_conv_1d_transpose()
layer_conv_2d()
layer_conv_2d_transpose()
layer_conv_3d()
layer_conv_3d_transpose()
layer_conv_lstm_1d()
layer_conv_lstm_2d()
layer_conv_lstm_3d()

layer_cropping_1d()
layer_cropping_2d()
layer_cropping_3d()
layer_dense()
layer_depthwise_conv_1d()
layer_depthwise_conv_2d()
layer_discretization()
layer_dot()
layer_dropout()
layer_einsum_dense()
layer_embedding()
layer_feature_space()
layer_flatten()
layer_flax_module_wrapper()
layer_gaussian_dropout()
layer_gaussian_noise()
layer_global_average_pooling_1d()
layer_global_average_pooling_2d()
layer_global_average_pooling_3d()
layer_global_max_pooling_1d()
layer_global_max_pooling_2d()
layer_global_max_pooling_3d()
layer_group_normalization()
layer_group_query_attention()
layer_gru()
layer_hashed_crossing()
layer_hashing()
layer_identity()
layer_integer_lookup()
layer_jax_model_wrapper()
layer_lambda()
layer_layer_normalization()
layer_lstm()
layer_masking()
layer_max_pooling_1d()
layer_max_pooling_2d()
layer_max_pooling_3d()
layer_maximum()
layer_mel_spectrogram()
layer_minimum()
layer_multi_head_attention()
layer_multiply()
layer_normalization()
layer_permute()
layer_random_brightness()
layer_random_contrast()
layer_random_crop()
layer_random_flip()

layer_random_rotation()
layer_random_translation()
layer_random_zoom()
layer_repeat_vector()
layer_rescaling()
layer_reshape()
layer_resizing()
layer_rnn()
layer_separable_conv_1d()
layer_separable_conv_2d()
layer_simple_rnn()
layer_spatial_dropout_1d()
layer_spatial_dropout_2d()
layer_spatial_dropout_3d()
layer_spectral_normalization()
layer_string_lookup()
layer_subtract()
layer_text_vectorization()
layer_tfsm()
layer_time_distributed()
layer_torch_module_wrapper()
layer_unit_normalization()
layer_upsampling_1d()
layer_upsampling_3d()
layer_zero_padding_1d()
layer_zero_padding_2d()
layer_zero_padding_3d()
rnn_cell_gru()
rnn_cell_lstm()
rnn_cell_simple()
rnn_cells_stack()

layer_upsampling_3d *Upsampling layer for 3D inputs.*

Description

Repeats the 1st, 2nd and 3rd dimensions of the data by `size[0]`, `size[1]` and `size[2]` respectively.

Usage

```
layer_upsampling_3d(object, size = list(2L, 2L, 2L), data_format = NULL, ...)
```

Arguments

object	Object to compose the layer with. A tensor, array, or sequential model.

`size`	Int, or list of 3 integers. The upsampling factors for dim1, dim2 and dim3.
`data_format`	A string, one of `"channels_last"` (default) or `"channels_first"`. The ordering of the dimensions in the inputs. `"channels_last"` corresponds to inputs with shape `(batch_size, spatial_dim1, spatial_dim2, spatial_dim3, channels)` while `"channels_first"` corresponds to inputs with shape `(batch_size, channels, spatial_dim1, s` When unspecified, uses `image_data_format` value found in your Keras config file at `~/.keras/keras.json` (if exists) else `"channels_last"`. Defaults to `"channels_last"`.
`...`	For forward/backward compatability.

Value

The return value depends on the value provided for the first argument. If `object` is:

- a `keras_model_sequential()`, then the layer is added to the sequential model (which is modified in place). To enable piping, the sequential model is also returned, invisibly.
- a `keras_input()`, then the output tensor from calling `layer(input)` is returned.
- `NULL` or missing, then a `Layer` instance is returned.

Example

```
input_shape <- c(2, 1, 2, 1, 3)
x <- array(1, dim = input_shape)
y <- layer_upsampling_3d(x, size = c(2, 2, 2))
shape(y)
```

```
## shape(2, 2, 4, 2, 3)
```

Input Shape

5D tensor with shape:

- If `data_format` is `"channels_last"`: `(batch_size, dim1, dim2, dim3, channels)`
- If `data_format` is `"channels_first"`: `(batch_size, channels, dim1, dim2, dim3)`

Output Shape

5D tensor with shape:

- If `data_format` is `"channels_last"`: `(batch_size, upsampled_dim1, upsampled_dim2, upsampled_dim3, chan`
- If `data_format` is `"channels_first"`: `(batch_size, channels, upsampled_dim1, upsampled_dim2, upsampled`

See Also

- https://keras.io/api/layers/reshaping_layers/up_sampling3d#upsampling3d-class

Other reshaping layers:
layer_cropping_1d()
layer_cropping_2d()
layer_cropping_3d()
layer_flatten()
layer_permute()
layer_repeat_vector()
layer_reshape()
layer_upsampling_1d()
layer_upsampling_2d()
layer_zero_padding_1d()
layer_zero_padding_2d()
layer_zero_padding_3d()

Other layers:
Layer()
layer_activation()
layer_activation_elu()
layer_activation_leaky_relu()
layer_activation_parametric_relu()
layer_activation_relu()
layer_activation_softmax()
layer_activity_regularization()
layer_add()
layer_additive_attention()
layer_alpha_dropout()
layer_attention()
layer_average()
layer_average_pooling_1d()
layer_average_pooling_2d()
layer_average_pooling_3d()
layer_batch_normalization()
layer_bidirectional()
layer_category_encoding()
layer_center_crop()
layer_concatenate()
layer_conv_1d()
layer_conv_1d_transpose()
layer_conv_2d()
layer_conv_2d_transpose()
layer_conv_3d()
layer_conv_3d_transpose()
layer_conv_lstm_1d()
layer_conv_lstm_2d()
layer_conv_lstm_3d()

layer_cropping_1d()
layer_cropping_2d()
layer_cropping_3d()
layer_dense()
layer_depthwise_conv_1d()
layer_depthwise_conv_2d()
layer_discretization()
layer_dot()
layer_dropout()
layer_einsum_dense()
layer_embedding()
layer_feature_space()
layer_flatten()
layer_flax_module_wrapper()
layer_gaussian_dropout()
layer_gaussian_noise()
layer_global_average_pooling_1d()
layer_global_average_pooling_2d()
layer_global_average_pooling_3d()
layer_global_max_pooling_1d()
layer_global_max_pooling_2d()
layer_global_max_pooling_3d()
layer_group_normalization()
layer_group_query_attention()
layer_gru()
layer_hashed_crossing()
layer_hashing()
layer_identity()
layer_integer_lookup()
layer_jax_model_wrapper()
layer_lambda()
layer_layer_normalization()
layer_lstm()
layer_masking()
layer_max_pooling_1d()
layer_max_pooling_2d()
layer_max_pooling_3d()
layer_maximum()
layer_mel_spectrogram()
layer_minimum()
layer_multi_head_attention()
layer_multiply()
layer_normalization()
layer_permute()
layer_random_brightness()
layer_random_contrast()
layer_random_crop()
layer_random_flip()

layer_random_rotation()
layer_random_translation()
layer_random_zoom()
layer_repeat_vector()
layer_rescaling()
layer_reshape()
layer_resizing()
layer_rnn()
layer_separable_conv_1d()
layer_separable_conv_2d()
layer_simple_rnn()
layer_spatial_dropout_1d()
layer_spatial_dropout_2d()
layer_spatial_dropout_3d()
layer_spectral_normalization()
layer_string_lookup()
layer_subtract()
layer_text_vectorization()
layer_tfsm()
layer_time_distributed()
layer_torch_module_wrapper()
layer_unit_normalization()
layer_upsampling_1d()
layer_upsampling_2d()
layer_zero_padding_1d()
layer_zero_padding_2d()
layer_zero_padding_3d()
rnn_cell_gru()
rnn_cell_lstm()
rnn_cell_simple()
rnn_cells_stack()

layer_zero_padding_1d *Zero-padding layer for 1D input (e.g. temporal sequence).*

Description

Zero-padding layer for 1D input (e.g. temporal sequence).

Usage

```
layer_zero_padding_1d(object, padding = 1L, ...)
```

Arguments

object	Object to compose the layer with. A tensor, array, or sequential model.

`padding`	Int, or list of int (length 2), or named listionary. • If int: how many zeros to add at the beginning and end of the padding dimension (axis 1). • If list of 2 ints: how many zeros to add at the beginning and the end of the padding dimension (`(left_pad, right_pad)`).
`...`	For forward/backward compatability.

Value

The return value depends on the value provided for the first argument. If `object` is:

- a `keras_model_sequential()`, then the layer is added to the sequential model (which is modified in place). To enable piping, the sequential model is also returned, invisibly.
- a `keras_input()`, then the output tensor from calling `layer(input)` is returned.
- `NULL` or missing, then a `Layer` instance is returned.

Example

```
input_shape <- c(2, 2, 3)
x <- op_reshape(seq_len(prod(input_shape)), input_shape)
x

## tf.Tensor(
## [[[ 1  2  3]
##   [ 4  5  6]]
##
##  [[ 7  8  9]
##   [10 11 12]]], shape=(2, 2, 3), dtype=int32)


y <- layer_zero_padding_1d(x, padding = 2)
y

## tf.Tensor(
## [[[ 0  0  0]
##   [ 0  0  0]
##   [ 1  2  3]
##   [ 4  5  6]
##   [ 0  0  0]
##   [ 0  0  0]]
##
##  [[ 0  0  0]
##   [ 0  0  0]
##   [ 7  8  9]
##   [10 11 12]
##   [ 0  0  0]
##   [ 0  0  0]]], shape=(2, 6, 3), dtype=int32)
```

Input Shape

3D tensor with shape `(batch_size, axis_to_pad, features)`

Output Shape

3D tensor with shape `(batch_size, padded_axis, features)`

See Also

- https://keras.io/api/layers/reshaping_layers/zero_padding1d#zeropadding1d-class

Other reshaping layers:
`layer_cropping_1d()`
`layer_cropping_2d()`
`layer_cropping_3d()`
`layer_flatten()`
`layer_permute()`
`layer_repeat_vector()`
`layer_reshape()`
`layer_upsampling_1d()`
`layer_upsampling_2d()`
`layer_upsampling_3d()`
`layer_zero_padding_2d()`
`layer_zero_padding_3d()`

Other layers:
`Layer()`
`layer_activation()`
`layer_activation_elu()`
`layer_activation_leaky_relu()`
`layer_activation_parametric_relu()`
`layer_activation_relu()`
`layer_activation_softmax()`
`layer_activity_regularization()`
`layer_add()`
`layer_additive_attention()`
`layer_alpha_dropout()`
`layer_attention()`
`layer_average()`
`layer_average_pooling_1d()`
`layer_average_pooling_2d()`
`layer_average_pooling_3d()`
`layer_batch_normalization()`
`layer_bidirectional()`
`layer_category_encoding()`
`layer_center_crop()`
`layer_concatenate()`
`layer_conv_1d()`
`layer_conv_1d_transpose()`

layer_conv_2d()
layer_conv_2d_transpose()
layer_conv_3d()
layer_conv_3d_transpose()
layer_conv_lstm_1d()
layer_conv_lstm_2d()
layer_conv_lstm_3d()
layer_cropping_1d()
layer_cropping_2d()
layer_cropping_3d()
layer_dense()
layer_depthwise_conv_1d()
layer_depthwise_conv_2d()
layer_discretization()
layer_dot()
layer_dropout()
layer_einsum_dense()
layer_embedding()
layer_feature_space()
layer_flatten()
layer_flax_module_wrapper()
layer_gaussian_dropout()
layer_gaussian_noise()
layer_global_average_pooling_1d()
layer_global_average_pooling_2d()
layer_global_average_pooling_3d()
layer_global_max_pooling_1d()
layer_global_max_pooling_2d()
layer_global_max_pooling_3d()
layer_group_normalization()
layer_group_query_attention()
layer_gru()
layer_hashed_crossing()
layer_hashing()
layer_identity()
layer_integer_lookup()
layer_jax_model_wrapper()
layer_lambda()
layer_layer_normalization()
layer_lstm()
layer_masking()
layer_max_pooling_1d()
layer_max_pooling_2d()
layer_max_pooling_3d()
layer_maximum()
layer_mel_spectrogram()
layer_minimum()
layer_multi_head_attention()

layer_multiply()
layer_normalization()
layer_permute()
layer_random_brightness()
layer_random_contrast()
layer_random_crop()
layer_random_flip()
layer_random_rotation()
layer_random_translation()
layer_random_zoom()
layer_repeat_vector()
layer_rescaling()
layer_reshape()
layer_resizing()
layer_rnn()
layer_separable_conv_1d()
layer_separable_conv_2d()
layer_simple_rnn()
layer_spatial_dropout_1d()
layer_spatial_dropout_2d()
layer_spatial_dropout_3d()
layer_spectral_normalization()
layer_string_lookup()
layer_subtract()
layer_text_vectorization()
layer_tfsm()
layer_time_distributed()
layer_torch_module_wrapper()
layer_unit_normalization()
layer_upsampling_1d()
layer_upsampling_2d()
layer_upsampling_3d()
layer_zero_padding_2d()
layer_zero_padding_3d()
rnn_cell_gru()
rnn_cell_lstm()
rnn_cell_simple()
rnn_cells_stack()

layer_zero_padding_2d *Zero-padding layer for 2D input (e.g. picture).*

Description

This layer can add rows and columns of zeros at the top, bottom, left and right side of an image tensor.

Usage

```
layer_zero_padding_2d(object, padding = list(1L, 1L), data_format = NULL, ...)
```

Arguments

`object`	Object to compose the layer with. A tensor, array, or sequential model.
`padding`	Int, or list of 2 ints, or list of 2 lists of 2 ints. • If int: the same symmetric padding is applied to height and width. • If list of 2 ints: interpreted as two different symmetric padding values for height and width: `(symmetric_height_pad, symmetric_width_pad)`. • If list of 2 lists of 2 ints: interpreted as `((top_pad, bottom_pad), (left_pad, right_pad))`.
`data_format`	A string, one of `"channels_last"` (default) or `"channels_first"`. The ordering of the dimensions in the inputs. `"channels_last"` corresponds to inputs with shape `(batch_size, height, width, channels)` while `"channels_first"` corresponds to inputs with shape `(batch_size, channels, height, width)`. When unspecified, uses `image_data_format` value found in your Keras config file at `~/.keras/keras.json` (if exists). Defaults to `"channels_last"`.
`...`	For forward/backward compatability.

Value

The return value depends on the value provided for the first argument. If `object` is:

- a `keras_model_sequential()`, then the layer is added to the sequential model (which is modified in place). To enable piping, the sequential model is also returned, invisibly.
- a `keras_input()`, then the output tensor from calling `layer(input)` is returned.
- `NULL` or missing, then a `Layer` instance is returned.

Example

```
input_shape <- c(1, 1, 2, 2)
x <- op_reshape(seq_len(prod(input_shape)), input_shape)
x

## tf.Tensor(
## [[[[1 2]
##    [3 4]]]], shape=(1, 1, 2, 2), dtype=int32)


y <- layer_zero_padding_2d(x, padding = 1)
y

## tf.Tensor(
## [[[[0 0]
##    [0 0]
##    [0 0]
##     [0 0]]
##
##    [[0 0]
##     [1 2]
##     [3 4]
##     [0 0]]
##
##    [[0 0]
##     [0 0]
##     [0 0]
##     [0 0]]]], shape=(1, 3, 4, 2), dtype=int32)
```

Input Shape

4D tensor with shape:

- If `data_format` is `"channels_last"`: `(batch_size, height, width, channels)`
- If `data_format` is `"channels_first"`: `(batch_size, channels, height, width)`

Output Shape

4D tensor with shape:

- If `data_format` is `"channels_last"`: `(batch_size, padded_height, padded_width, channels)`
- If `data_format` is `"channels_first"`: `(batch_size, channels, padded_height, padded_width)`

See Also

- `https://keras.io/api/layers/reshaping_layers/zero_padding2d#zeropadding2d-class`

Other reshaping layers:
`layer_cropping_1d()`
`layer_cropping_2d()`
`layer_cropping_3d()`
`layer_flatten()`
`layer_permute()`
`layer_repeat_vector()`
`layer_reshape()`
`layer_upsampling_1d()`
`layer_upsampling_2d()`
`layer_upsampling_3d()`
`layer_zero_padding_1d()`
`layer_zero_padding_3d()`

Other layers:
`Layer()`
`layer_activation()`
`layer_activation_elu()`
`layer_activation_leaky_relu()`

layer_activation_parametric_relu()
layer_activation_relu()
layer_activation_softmax()
layer_activity_regularization()
layer_add()
layer_additive_attention()
layer_alpha_dropout()
layer_attention()
layer_average()
layer_average_pooling_1d()
layer_average_pooling_2d()
layer_average_pooling_3d()
layer_batch_normalization()
layer_bidirectional()
layer_category_encoding()
layer_center_crop()
layer_concatenate()
layer_conv_1d()
layer_conv_1d_transpose()
layer_conv_2d()
layer_conv_2d_transpose()
layer_conv_3d()
layer_conv_3d_transpose()
layer_conv_lstm_1d()
layer_conv_lstm_2d()
layer_conv_lstm_3d()
layer_cropping_1d()
layer_cropping_2d()
layer_cropping_3d()
layer_dense()
layer_depthwise_conv_1d()
layer_depthwise_conv_2d()
layer_discretization()
layer_dot()
layer_dropout()
layer_einsum_dense()
layer_embedding()
layer_feature_space()
layer_flatten()
layer_flax_module_wrapper()
layer_gaussian_dropout()
layer_gaussian_noise()
layer_global_average_pooling_1d()
layer_global_average_pooling_2d()
layer_global_average_pooling_3d()
layer_global_max_pooling_1d()
layer_global_max_pooling_2d()
layer_global_max_pooling_3d()

layer_group_normalization()
layer_group_query_attention()
layer_gru()
layer_hashed_crossing()
layer_hashing()
layer_identity()
layer_integer_lookup()
layer_jax_model_wrapper()
layer_lambda()
layer_layer_normalization()
layer_lstm()
layer_masking()
layer_max_pooling_1d()
layer_max_pooling_2d()
layer_max_pooling_3d()
layer_maximum()
layer_mel_spectrogram()
layer_minimum()
layer_multi_head_attention()
layer_multiply()
layer_normalization()
layer_permute()
layer_random_brightness()
layer_random_contrast()
layer_random_crop()
layer_random_flip()
layer_random_rotation()
layer_random_translation()
layer_random_zoom()
layer_repeat_vector()
layer_rescaling()
layer_reshape()
layer_resizing()
layer_rnn()
layer_separable_conv_1d()
layer_separable_conv_2d()
layer_simple_rnn()
layer_spatial_dropout_1d()
layer_spatial_dropout_2d()
layer_spatial_dropout_3d()
layer_spectral_normalization()
layer_string_lookup()
layer_subtract()
layer_text_vectorization()
layer_tfsm()
layer_time_distributed()
layer_torch_module_wrapper()
layer_unit_normalization()

layer_upsampling_1d()
layer_upsampling_2d()
layer_upsampling_3d()
layer_zero_padding_1d()
layer_zero_padding_3d()
rnn_cell_gru()
rnn_cell_lstm()
rnn_cell_simple()
rnn_cells_stack()

---

layer_zero_padding_3d *Zero-padding layer for 3D data (spatial or spatio-temporal).*

Description

Zero-padding layer for 3D data (spatial or spatio-temporal).

Usage

```
layer_zero_padding_3d(
  object,
  padding = list(list(1L, 1L), list(1L, 1L), list(1L, 1L)),
  data_format = NULL,
  ...
)
```

Arguments

`object`	Object to compose the layer with. A tensor, array, or sequential model.
`padding`	Int, or list of 3 ints, or list of 3 lists of 2 ints. • If int: the same symmetric padding is applied to depth, height, and width. • If list of 3 ints: interpreted as three different symmetric padding values for depth, height, and width: `(symmetric_dim1_pad, symmetric_dim2_pad, symmetric_dim3_pad)`. • If list of 3 lists of 2 ints: interpreted as `((left_dim1_pad, right_dim1_pad), (left_dim2_pad, r`
`data_format`	A string, one of `"channels_last"` (default) or `"channels_first"`. The ordering of the dimensions in the inputs. `"channels_last"` corresponds to inputs with shape `(batch_size, spatial_dim1, spatial_dim2, spatial_dim3, channels)` while `"channels_first"` corresponds to inputs with shape `(batch_size, channels, spatial_dim1, s` When unspecified, uses `image_data_format` value found in your Keras config file at `~/.keras/keras.json` (if exists). Defaults to `"channels_last"`.
`...`	For forward/backward compatability.

Value

The return value depends on the value provided for the first argument. If `object` is:

- a `keras_model_sequential()`, then the layer is added to the sequential model (which is modified in place). To enable piping, the sequential model is also returned, invisibly.
- a `keras_input()`, then the output tensor from calling `layer(input)` is returned.
- `NULL` or missing, then a `Layer` instance is returned.

Example

```
input_shape <- c(1, 1, 2, 2, 3)
x <- op_reshape(seq_len(prod(input_shape)), input_shape)
x
```

```
## tf.Tensor(
## [[[[[ 1  2  3]
##     [ 4  5  6]]
##
##    [[ 7  8  9]
##     [10 11 12]]]]], shape=(1, 1, 2, 2, 3), dtype=int32)
```

```
y <- layer_zero_padding_3d(x, padding = 2)
shape(y)
```

```
## shape(1, 5, 6, 6, 3)
```

Input Shape

5D tensor with shape:

- If `data_format` is `"channels_last"`: `(batch_size, first_axis_to_pad, second_axis_to_pad, third_axis_to`
- If `data_format` is `"channels_first"`: `(batch_size, depth, first_axis_to_pad, second_axis_to_pad, third`

Output Shape

5D tensor with shape:

- If `data_format` is `"channels_last"`: `(batch_size, first_padded_axis, second_padded_axis, third_axis_to`
- If `data_format` is `"channels_first"`: `(batch_size, depth, first_padded_axis, second_padded_axis, third`

See Also

- https://keras.io/api/layers/reshaping_layers/zero_padding3d#zeropadding3d-class

Other reshaping layers:
layer_cropping_1d()
layer_cropping_2d()
layer_cropping_3d()
layer_flatten()
layer_permute()
layer_repeat_vector()
layer_reshape()
layer_upsampling_1d()
layer_upsampling_2d()
layer_upsampling_3d()
layer_zero_padding_1d()
layer_zero_padding_2d()

Other layers:
Layer()
layer_activation()
layer_activation_elu()
layer_activation_leaky_relu()
layer_activation_parametric_relu()
layer_activation_relu()
layer_activation_softmax()
layer_activity_regularization()
layer_add()
layer_additive_attention()
layer_alpha_dropout()
layer_attention()
layer_average()
layer_average_pooling_1d()
layer_average_pooling_2d()
layer_average_pooling_3d()
layer_batch_normalization()
layer_bidirectional()
layer_category_encoding()
layer_center_crop()
layer_concatenate()
layer_conv_1d()
layer_conv_1d_transpose()
layer_conv_2d()
layer_conv_2d_transpose()
layer_conv_3d()
layer_conv_3d_transpose()
layer_conv_lstm_1d()
layer_conv_lstm_2d()
layer_conv_lstm_3d()

layer_cropping_1d()
layer_cropping_2d()
layer_cropping_3d()
layer_dense()
layer_depthwise_conv_1d()
layer_depthwise_conv_2d()
layer_discretization()
layer_dot()
layer_dropout()
layer_einsum_dense()
layer_embedding()
layer_feature_space()
layer_flatten()
layer_flax_module_wrapper()
layer_gaussian_dropout()
layer_gaussian_noise()
layer_global_average_pooling_1d()
layer_global_average_pooling_2d()
layer_global_average_pooling_3d()
layer_global_max_pooling_1d()
layer_global_max_pooling_2d()
layer_global_max_pooling_3d()
layer_group_normalization()
layer_group_query_attention()
layer_gru()
layer_hashed_crossing()
layer_hashing()
layer_identity()
layer_integer_lookup()
layer_jax_model_wrapper()
layer_lambda()
layer_layer_normalization()
layer_lstm()
layer_masking()
layer_max_pooling_1d()
layer_max_pooling_2d()
layer_max_pooling_3d()
layer_maximum()
layer_mel_spectrogram()
layer_minimum()
layer_multi_head_attention()
layer_multiply()
layer_normalization()
layer_permute()
layer_random_brightness()
layer_random_contrast()
layer_random_crop()
layer_random_flip()

layer_random_rotation()
layer_random_translation()
layer_random_zoom()
layer_repeat_vector()
layer_rescaling()
layer_reshape()
layer_resizing()
layer_rnn()
layer_separable_conv_1d()
layer_separable_conv_2d()
layer_simple_rnn()
layer_spatial_dropout_1d()
layer_spatial_dropout_2d()
layer_spatial_dropout_3d()
layer_spectral_normalization()
layer_string_lookup()
layer_subtract()
layer_text_vectorization()
layer_tfsm()
layer_time_distributed()
layer_torch_module_wrapper()
layer_unit_normalization()
layer_upsampling_1d()
layer_upsampling_2d()
layer_upsampling_3d()
layer_zero_padding_1d()
layer_zero_padding_2d()
rnn_cell_gru()
rnn_cell_lstm()
rnn_cell_simple()
rnn_cells_stack()

LearningRateSchedule *Define a custom* LearningRateSchedule *class*

Description

Subclass the keras learning rate schedule base class.

You can use a learning rate schedule to modulate how the learning rate of your optimizer changes over time.

Several built-in learning rate schedules are available, such as learning_rate_schedule_exponential_decay() or learning_rate_schedule_piecewise_constant_decay():

```
lr_schedule <- learning_rate_schedule_exponential_decay(
  initial_learning_rate = 1e-2,
  decay_steps = 10000,
  decay_rate = 0.9
)
optimizer <- optimizer_sgd(learning_rate = lr_schedule)
```

A `LearningRateSchedule()` instance can be passed in as the `learning_rate` argument of any optimizer.

To implement your own schedule object, you should implement the `call` method, which takes a `step` argument (a scalar integer backend tensor, the current training step count). Note that `step` is 0-based (i.e., the first step is `0`). Like for any other Keras object, you can also optionally make your object serializable by implementing the `get_config()` and `from_config()` methods.

Usage

```
LearningRateSchedule(
  classname,
  call = NULL,
  initialize = NULL,
  get_config = NULL,
  ...,
  public = list(),
  private = list(),
  inherit = NULL,
  parent_env = parent.frame()
)
```

Arguments

`classname`	String, the name of the custom class. (Conventionally, CamelCase).
`call`, `initialize`, `get_config`	Recommended methods to implement. See description and details sections.
`...`, `public`	Additional methods or public members of the custom class.
`private`	Named list of R objects (typically, functions) to include in instance private environments. `private` methods will have all the same symbols in scope as public methods (See section "Symbols in Scope"). Each instance will have it's own `private` environment. Any objects in `private` will be invisible from the Keras framework and the Python runtime.
`inherit`	What the custom class will subclass. By default, the base keras class.
`parent_env`	The R environment that all class methods will have as a grandparent.

Value

A function that returns `LearningRateSchedule` instances, similar to the built-in `learning_rate_schedule_*` family of functions.

Example

```
my_custom_learning_rate_schedule <- LearningRateSchedule(
  classname = "MyLRSchedule",
  initialize = function( initial_learning_rate) {

    self$initial_learning_rate <- initial_learning_rate
  },


  call = function(step) {
    # note that `step` is a tensor
    # and call() will be traced via tf_function() or similar.

    str(step) # <KerasVariable shape=(), dtype=int64, path=SGD/iteration>

    # print 'step' every 1000 steps
    op_cond((step %% 1000) == 0,
            \() {tensorflow::tf$print(step); NULL},
            \() {NULL})
    self$initial_learning_rate / (step + 1)
  }
)

optimizer <- optimizer_sgd(
  learning_rate = my_custom_learning_rate_schedule(0.1)
)

# You can also call schedule instances directly
# (e.g., for interactive testing, or if implementing a custom optimizer)
schedule <- my_custom_learning_rate_schedule(0.1)
step <- keras$Variable(initializer = op_ones,
                       shape = shape(),
                       dtype = "int64")
schedule(step)

## <KerasVariable shape=(), dtype=int64, path=variable>


## tf.Tensor(0.0, shape=(), dtype=float64)
```

Methods available:

- `get_config()`

Symbols in scope

All R function custom methods (public and private) will have the following symbols in scope:

- `self`: The custom class instance.
- `super`: The custom class superclass.
- `private`: An R environment specific to the class instance. Any objects assigned here are invisible to the Keras framework.
- `__class__` and `as.symbol(classname)`: the custom class type object.

See Also

Other optimizer learning rate schedules:
`learning_rate_schedule_cosine_decay()`
`learning_rate_schedule_cosine_decay_restarts()`
`learning_rate_schedule_exponential_decay()`
`learning_rate_schedule_inverse_time_decay()`
`learning_rate_schedule_piecewise_constant_decay()`
`learning_rate_schedule_polynomial_decay()`

`learning_rate_schedule_cosine_decay`
A `LearningRateSchedule` *that uses a cosine decay with optional warmup.*

Description

See Loshchilov & Hutter, ICLR2016, SGDR: Stochastic Gradient Descent with Warm Restarts.

For the idea of a linear warmup of our learning rate, see Goyal et al..

When we begin training a model, we often want an initial increase in our learning rate followed by a decay. If `warmup_target` is an int, this schedule applies a linear increase per optimizer step to our learning rate from `initial_learning_rate` to `warmup_target` for a duration of `warmup_steps`. Afterwards, it applies a cosine decay function taking our learning rate from `warmup_target` to `alpha` for a duration of `decay_steps`. If `warmup_target` is NULL we skip warmup and our decay will take our learning rate from `initial_learning_rate` to `alpha`. It requires a `step` value to compute the learning rate. You can just pass a backend variable that you increment at each training step.

The schedule is a 1-arg callable that produces a warmup followed by a decayed learning rate when passed the current optimizer step. This can be useful for changing the learning rate value across different invocations of optimizer functions.

Our warmup is computed as:

```
warmup_learning_rate <- function(step) {
  completed_fraction <- step / warmup_steps
  total_delta <- target_warmup - initial_learning_rate
  completed_fraction * total_delta
}
```

And our decay is computed as:

```
if (is.null(warmup_target)) {
  initial_decay_lr <- initial_learning_rate
} else {
  initial_decay_lr <- warmup_target
}

decayed_learning_rate <- function(step) {
  step <- min(step, decay_steps)
  cosine_decay <- 0.5 * (1 + cos(pi * step / decay_steps))
  decayed <- (1 - alpha) * cosine_decay + alpha
  initial_decay_lr * decayed
}
```

Example usage without warmup:

```
decay_steps <- 1000
initial_learning_rate <- 0.1
lr_decayed_fn <- learning_rate_schedule_cosine_decay(
    initial_learning_rate, decay_steps)
```

Example usage with warmup:

```
decay_steps <- 1000
initial_learning_rate <- 0
warmup_steps <- 1000
target_learning_rate <- 0.1
lr_warmup_decayed_fn <- learning_rate_schedule_cosine_decay(
    initial_learning_rate, decay_steps, warmup_target = target_learning_rate,
    warmup_steps = warmup_steps
)
```

You can pass this schedule directly into a `optimizer` as the learning rate. The learning rate schedule is also serializable and deserializable using `keras$optimizers$schedules$serialize` and `keras$optimizers$schedules$deserialize`.

Usage

```
learning_rate_schedule_cosine_decay(
  initial_learning_rate,
  decay_steps,
  alpha = 0,
  name = "CosineDecay",
  warmup_target = NULL,
  warmup_steps = 0L
)
```

Arguments

`initial_learning_rate`	A float. The initial learning rate.
`decay_steps`	A int. Number of steps to decay over.
`alpha`	A float. Minimum learning rate value for decay as a fraction of `initial_learning_rate`.
`name`	String. Optional name of the operation. Defaults to `"CosineDecay"`.
`warmup_target`	A float. The target learning rate for our warmup phase. Will cast to the `initial_learning_rate` datatype. Setting to `NULL` will skip warmup and begins decay phase from `initial_learning_rate`. Otherwise scheduler will warmup from `initial_learning_rate` to `warmup_target`.
`warmup_steps`	A int. Number of steps to warmup over.

Value

A 1-arg callable learning rate schedule that takes the current optimizer step and outputs the decayed learning rate, a scalar tensor of the same type as `initial_learning_rate`.

See Also

- `https://keras.io/api/optimizers/learning_rate_schedules/cosine_decay#cosinedecay-class`

Other optimizer learning rate schedules:
`LearningRateSchedule()`
`learning_rate_schedule_cosine_decay_restarts()`
`learning_rate_schedule_exponential_decay()`
`learning_rate_schedule_inverse_time_decay()`
`learning_rate_schedule_piecewise_constant_decay()`
`learning_rate_schedule_polynomial_decay()`

`learning_rate_schedule_cosine_decay_restarts`

A `LearningRateSchedule` *that uses a cosine decay schedule with restarts.*

Description

See Loshchilov & Hutter, ICLR2016, SGDR: Stochastic Gradient Descent with Warm Restarts.

When training a model, it is often useful to lower the learning rate as the training progresses. This schedule applies a cosine decay function with restarts to an optimizer step, given a provided initial learning rate. It requires a `step` value to compute the decayed learning rate. You can just pass a backend variable that you increment at each training step.

The schedule is a 1-arg callable that produces a decayed learning rate when passed the current optimizer step. This can be useful for changing the learning rate value across different invocations of optimizer functions.

The learning rate multiplier first decays from 1 to `alpha` for `first_decay_steps` steps. Then, a warm restart is performed. Each new warm restart runs for `t_mul` times more steps and with `m_mul` times initial learning rate as the new learning rate.

Usage

```
learning_rate_schedule_cosine_decay_restarts(
  initial_learning_rate,
  first_decay_steps,
  t_mul = 2,
  m_mul = 1,
  alpha = 0,
  name = "SGDRDecay"
)
```

Arguments

`initial_learning_rate`	A float. The initial learning rate.
`first_decay_steps`	An integer. Number of steps to decay over.
`t_mul`	A float. Used to derive the number of iterations in the i-th period.
`m_mul`	A float. Used to derive the initial learning rate of the i-th period.
`alpha`	A float. Minimum learning rate value as a fraction of the `initial_learning_rate`.
`name`	String. Optional name of the operation. Defaults to `"SGDRDecay"`.

Value

A 1-arg callable learning rate schedule that takes the current optimizer step and outputs the decayed learning rate, a scalar tensor of the same type as `initial_learning_rate`.

Example

```
first_decay_steps <- 1000
lr_decayed_fn <- learning_rate_schedule_cosine_decay_restarts(
        0.001,
        first_decay_steps)
```

You can pass this schedule directly into a `optimizer` as the learning rate. The learning rate schedule is also serializable and deserializable using `keras$optimizers$schedules$serialize` and `keras$optimizers$schedules$deserialize`.

See Also

- `https://keras.io/api/optimizers/learning_rate_schedules/cosine_decay_restarts#cosinedecayrestarts-class`

Other optimizer learning rate schedules:
`LearningRateSchedule()`
`learning_rate_schedule_cosine_decay()`
`learning_rate_schedule_exponential_decay()`
`learning_rate_schedule_inverse_time_decay()`
`learning_rate_schedule_piecewise_constant_decay()`
`learning_rate_schedule_polynomial_decay()`

learning_rate_schedule_exponential_decay

A `LearningRateSchedule` *that uses an exponential decay schedule.*

Description

When training a model, it is often useful to lower the learning rate as the training progresses. This schedule applies an exponential decay function to an optimizer step, given a provided initial learning rate.

The schedule is a 1-arg callable that produces a decayed learning rate when passed the current optimizer step. This can be useful for changing the learning rate value across different invocations of optimizer functions. It is computed as:

```
decayed_learning_rate <- function(step) {
  initial_learning_rate * decay_rate ^ (step / decay_steps)
}
```

If the argument `staircase` is `TRUE`, then `step / decay_steps` is an integer division and the decayed learning rate follows a staircase function.

You can pass this schedule directly into a `optimizer` as the learning rate.

Usage

```
learning_rate_schedule_exponential_decay(
  initial_learning_rate,
  decay_steps,
  decay_rate,
  staircase = FALSE,
  name = "ExponentialDecay"
)
```

Arguments

`initial_learning_rate`	A float. The initial learning rate.
`decay_steps`	A integer. Must be positive. See the decay computation above.
`decay_rate`	A float. The decay rate.
`staircase`	Boolean. If `TRUE` decay the learning rate at discrete intervals.
`name`	String. Optional name of the operation. Defaults to `"ExponentialDecay"`.

Value

A 1-arg callable learning rate schedule that takes the current optimizer step and outputs the decayed learning rate, a scalar tensor of the same type as `initial_learning_rate`.

Examples

When fitting a Keras model, decay every 100000 steps with a base of 0.96:

```
initial_learning_rate <- 0.1
lr_schedule <- learning_rate_schedule_exponential_decay(
    initial_learning_rate,
    decay_steps=100000,
    decay_rate=0.96,
    staircase=TRUE)

model %>% compile(
  optimizer = optimizer_sgd(learning_rate = lr_schedule),
  loss = 'sparse_categorical_crossentropy',
  metrics = c('accuracy'))

model %>% fit(data, labels, epochs=5)
```

The learning rate schedule is also serializable and deserializable using `keras$optimizers$schedules$serialize` and `keras$optimizers$schedules$deserialize`.

See Also

- https://keras.io/api/optimizers/learning_rate_schedules/exponential_decay#exponentialdecay-clas

Other optimizer learning rate schedules:
`LearningRateSchedule()`
`learning_rate_schedule_cosine_decay()`
`learning_rate_schedule_cosine_decay_restarts()`
`learning_rate_schedule_inverse_time_decay()`
`learning_rate_schedule_piecewise_constant_decay()`
`learning_rate_schedule_polynomial_decay()`

---

learning_rate_schedule_inverse_time_decay

A `LearningRateSchedule` *that uses an inverse time decay schedule.*

Description

When training a model, it is often useful to lower the learning rate as the training progresses. This schedule applies the inverse decay function to an optimizer step, given a provided initial learning rate. It requires a `step` value to compute the decayed learning rate. You can just pass a backend variable that you increment at each training step.

The schedule is a 1-arg callable that produces a decayed learning rate when passed the current optimizer step. This can be useful for changing the learning rate value across different invocations of optimizer functions. It is computed as:

```
decayed_learning_rate <- function(step) {
  initial_learning_rate / (1 + decay_rate * step / decay_step)
}
```

or, if `staircase` is `TRUE`, as:

```
decayed_learning_rate <- function(step) {
  initial_learning_rate /
           (1 + decay_rate * floor(step / decay_step))
}
```

You can pass this schedule directly into a `optimizer_*` as the learning rate.

Usage

```
learning_rate_schedule_inverse_time_decay(
  initial_learning_rate,
  decay_steps,
  decay_rate,
  staircase = FALSE,
  name = "InverseTimeDecay"
)
```

Arguments

`initial_learning_rate`	A float. The initial learning rate.
`decay_steps`	How often to apply decay.
`decay_rate`	A number. The decay rate.
`staircase`	Whether to apply decay in a discrete staircase, as o pposed to continuous, fashion.
`name`	String. Optional name of the operation. Defaults to `"InverseTimeDecay"`.

Value

A 1-arg callable learning rate schedule that takes the current optimizer step and outputs the decayed learning rate, a scalar tensor of the same type as `initial_learning_rate`.

Examples

Fit a Keras model when decaying 1/t with a rate of 0.5:

```
...
initial_learning_rate <- 0.1
decay_steps <- 1.0
decay_rate <- 0.5
learning_rate_fn <- learning_rate_schedule_inverse_time_decay(
    initial_learning_rate, decay_steps, decay_rate)
```

```
model %>% compile(
  optimizer = optimizer_sgd(learning_rate=learning_rate_fn),
  loss = 'sparse_categorical_crossentropy',
  metrics = 'accuracy')
)

model %>% fit(data, labels, epochs=5)
```

See Also

- https://keras.io/api/optimizers/learning_rate_schedules/inverse_time_decay#inversetimedecay-class

Other optimizer learning rate schedules:
LearningRateSchedule()
learning_rate_schedule_cosine_decay()
learning_rate_schedule_cosine_decay_restarts()
learning_rate_schedule_exponential_decay()
learning_rate_schedule_piecewise_constant_decay()
learning_rate_schedule_polynomial_decay()

learning_rate_schedule_piecewise_constant_decay

A LearningRateSchedule *that uses a piecewise constant decay schedule.*

Description

The function returns a 1-arg callable to compute the piecewise constant when passed the current optimizer step. This can be useful for changing the learning rate value across different invocations of optimizer functions.

Usage

```
learning_rate_schedule_piecewise_constant_decay(
  boundaries,
  values,
  name = "PiecewiseConstant"
)
```

Arguments

boundaries	A list of Python numbers with strictly increasing entries, and with all elements having the same type as the optimizer step.

values	A list of Python numbers that specifies the values for the intervals defined by `boundaries`. It should have one more element than `boundaries`, and all elements should have the same type.
name	A string. Optional name of the operation. Defaults to `"PiecewiseConstant"`.

Value

A 1-arg callable learning rate schedule that takes the current optimizer step and outputs the decayed learning rate, a scalar tensor of the same type as the boundary tensors.

The output of the 1-arg function that takes the `step` is `values[0]` when `step` <= `boundaries[0]`, `values[1]` when `step` > `boundaries[0]` and `step` <= `boundaries[1]`, ..., and `values[-1]` when `step` > `boundaries[-1]`.

Examples

use a learning rate that's 1.0 for the first 100001 steps, 0.5 for the next 10000 steps, and 0.1 for any additional steps.

```
step <- 0
boundaries <- c(100000, 110000)
values <- c(1.0, 0.5, 0.1)
learning_rate_fn <- learning_rate_schedule_piecewise_constant_decay(
  boundaries, values)

# Later, whenever we perform an optimization step, we pass in the step.
learning_rate <- learning_rate_fn(step)
```

You can pass this schedule directly into a `optimizer` as the learning rate. The learning rate schedule is also serializable and deserializable using `keras$optimizers$schedules$serialize` and `keras$optimizers$schedules$deserialize`.

Raises

ValueError: if the number of elements in the `boundaries` and `values` lists do not match.

See Also

- https://keras.io/api/optimizers/learning_rate_schedules/piecewise_constant_decay#piecewiseconstantdecay-class

Other optimizer learning rate schedules:
`LearningRateSchedule()`
`learning_rate_schedule_cosine_decay()`
`learning_rate_schedule_cosine_decay_restarts()`
`learning_rate_schedule_exponential_decay()`
`learning_rate_schedule_inverse_time_decay()`
`learning_rate_schedule_polynomial_decay()`

learning_rate_schedule_polynomial_decay

A LearningRateSchedule *that uses a polynomial decay schedule.*

Description

It is commonly observed that a monotonically decreasing learning rate, whose degree of change is carefully chosen, results in a better performing model. This schedule applies a polynomial decay function to an optimizer step, given a provided initial_learning_rate, to reach an end_learning_rate in the given decay_steps.

It requires a step value to compute the decayed learning rate. You can just pass a backend variable that you increment at each training step.

The schedule is a 1-arg callable that produces a decayed learning rate when passed the current optimizer step. This can be useful for changing the learning rate value across different invocations of optimizer functions. It is computed as:

```
decayed_learning_rate <- function(step) {
  step = min(step, decay_steps)
  ((initial_learning_rate - end_learning_rate) *
    (1 - step / decay_steps) ^ (power)) +
    end_learning_rate
}
```

If cycle is TRUE then a multiple of decay_steps is used, the first one that is bigger than step.

```
decayed_learning_rate <- function(step) {
  decay_steps = decay_steps * ceil(step / decay_steps)
  ((initial_learning_rate - end_learning_rate) *
      (1 - step / decay_steps) ^ (power)) +
    end_learning_rate
}
```

You can pass this schedule directly into a Optimizer as the learning rate.

Usage

```
learning_rate_schedule_polynomial_decay(
  initial_learning_rate,
  decay_steps,
  end_learning_rate = 1e-04,
  power = 1,
  cycle = FALSE,
  name = "PolynomialDecay"
)
```

Arguments

initial_learning_rate	A float. The initial learning rate.
decay_steps	A integer. Must be positive. See the decay computation above.
end_learning_rate	A float. The minimal end learning rate.
power	A float. The power of the polynomial. Defaults to `1.0`.
cycle	A boolean, whether it should cycle beyond decay_steps.
name	String. Optional name of the operation. Defaults to `"PolynomialDecay"`.

Value

A 1-arg callable learning rate schedule that takes the current optimizer step and outputs the decayed learning rate, a scalar tensor of the same type as `initial_learning_rate`.

Examples

Fit a model while decaying from 0.1 to 0.01 in 10000 steps using sqrt (i.e. power=0.5):

```
...
starter_learning_rate <- 0.1
end_learning_rate <- 0.01
decay_steps <- 10000
learning_rate_fn <- learning_rate_schedule_polynomial_decay(
    starter_learning_rate,
    decay_steps,
    end_learning_rate,
    power=0.5)

model %>% compile(
  optimizer = optimizer_sgd(learning_rate=learning_rate_fn),
  loss = 'sparse_categorical_crossentropy',
  metrics = 'accuracy'
)

model %>% fit(data, labels, epochs=5)
```

The learning rate schedule is also serializable and deserializable using `keras$optimizers$schedules$serialize` and `keras$optimizers$schedules$deserialize`.

See Also

- https://keras.io/api/optimizers/learning_rate_schedules/polynomial_decay#polynomialdecay-class

Other optimizer learning rate schedules:
`LearningRateSchedule()`
`learning_rate_schedule_cosine_decay()`
`learning_rate_schedule_cosine_decay_restarts()`

learning_rate_schedule_exponential_decay()
learning_rate_schedule_inverse_time_decay()
learning_rate_schedule_piecewise_constant_decay()

load_model — *Loads a model saved via* save_model().

Description

Loads a model saved via save_model().

Usage

```
load_model(model, custom_objects = NULL, compile = TRUE, safe_mode = TRUE)
```

Arguments

model	string, path to the saved model file, or a raw vector, as returned by save_model(filepath = NULL)
custom_objects	Optional named list mapping names to custom classes or functions to be considered during deserialization.
compile	Boolean, whether to compile the model after loading.
safe_mode	Boolean, whether to disallow unsafe lambda deserialization. When safe_mode=FALSE, loading an object has the potential to trigger arbitrary code execution. This argument is only applicable to the Keras v3 model format. Defaults to TRUE.

Value

A Keras model instance. If the original model was compiled, and the argument compile = TRUE is set, then the returned model will be compiled. Otherwise, the model will be left uncompiled.

Examples

```
model <- keras_model_sequential(input_shape = c(3)) |>
  layer_dense(5) |>
  layer_activation_softmax()

model |> save_model("model.keras")
loaded_model <- load_model("model.keras")


x <- random_uniform(c(10, 3))
stopifnot(all.equal(
  model |> predict(x),
  loaded_model |> predict(x)
))
```

Note that the model variables may have different name values (`var$name` property, e.g. `"dense_1/kernel:0"`) after being reloaded. It is recommended that you use layer attributes to access specific variables, e.g. `model |> get_layer("dense_1") |> _$kernel`.

See Also

- `https://keras.io/api/models/model_saving_apis/model_saving_and_loading#loadmodel-function`

Other saving and loading functions:
`export_savedmodel.keras.src.models.model.Model()`
`layer_tfsm()`
`load_model_weights()`
`register_keras_serializable()`
`save_model()`
`save_model_config()`
`save_model_weights()`
`with_custom_object_scope()`

`load_model_weights` *Load weights from a file saved via* `save_model_weights()`.

Description

Weights are loaded based on the network's topology. This means the architecture should be the same as when the weights were saved. Note that layers that don't have weights are not taken into account in the topological ordering, so adding or removing layers is fine as long as they don't have weights.

Partial weight loading

If you have modified your model, for instance by adding a new layer (with weights) or by changing the shape of the weights of a layer, you can choose to ignore errors and continue loading by setting `skip_mismatch=TRUE`. In this case any layer with mismatching weights will be skipped. A warning will be displayed for each skipped layer.

Usage

```
load_model_weights(model, filepath, skip_mismatch = FALSE, ...)
```

Arguments

`model`	A keras model.
`filepath`	String, path to the weights file to load. It can either be a `.weights.h5` file or a legacy `.h5` weights file.
`skip_mismatch`	Boolean, whether to skip loading of layers where there is a mismatch in the number of weights, or a mismatch in the shape of the weights.
`...`	For forward/backward compatability.

Value

This is called primarily for side effects. `model` is returned, invisibly, to enable usage with the pipe.

See Also

- https://keras.io/api/models/model_saving_apis/weights_saving_and_loading#loadweights-method

Other saving and loading functions:
`export_savedmodel.keras.src.models.model.Model()`
`layer_tfsm()`
`load_model()`
`register_keras_serializable()`
`save_model()`
`save_model_config()`
`save_model_weights()`
`with_custom_object_scope()`

`Loss` *Subclass the base* `Loss` *class*

Description

Use this to define a custom loss class. Note, in most cases you do not need to subclass `Loss` to define a custom loss: you can also pass a bare R function, or a named R function defined with `custom_metric()`, as a loss function to `compile()`.

Usage

```
Loss(
  classname,
  call = NULL,
  ...,
  public = list(),
  private = list(),
  inherit = NULL,
  parent_env = parent.frame()
)
```

Arguments

`classname`	String, the name of the custom class. (Conventionally, CamelCase).
`call`	`function(y_true, y_pred)` Method to be implemented by subclasses: Function that contains the logic for loss calculation using `y_true`, `y_pred`.
`..., public`	Additional methods or public members of the custom class.

`private`	Named list of R objects (typically, functions) to include in instance private environments. `private` methods will have all the same symbols in scope as public methods (See section "Symbols in Scope"). Each instance will have it's own `private` environment. Any objects in `private` will be invisible from the Keras framework and the Python runtime.
`inherit`	What the custom class will subclass. By default, the base keras class.
`parent_env`	The R environment that all class methods will have as a grandparent.

Details

Example subclass implementation:

```
loss_custom_mse <- Loss(
  classname = "CustomMeanSquaredError",
  call = function(y_true, y_pred) {
    op_mean(op_square(y_pred - y_true), axis = -1)
  }
)

# Usage in compile()
model <- keras_model_sequential(input_shape = 10) |> layer_dense(10)
model |> compile(loss = loss_custom_mse())

# Standalone usage
mse <- loss_custom_mse(name = "my_custom_mse_instance")

y_true <- op_arange(20) |> op_reshape(c(4, 5))
y_pred <- op_arange(20) |> op_reshape(c(4, 5)) * 2
(loss <- mse(y_true, y_pred))

## tf.Tensor(123.5, shape=(), dtype=float32)


loss2 <- (y_pred - y_true)^2 |>
  op_mean(axis = -1) |>
  op_mean()

stopifnot(all.equal(as.array(loss), as.array(loss2)))

sample_weight <-array(c(.25, .25, 1, 1))
(weighted_loss <- mse(y_true, y_pred, sample_weight = sample_weight))

## tf.Tensor(112.8125, shape=(), dtype=float32)


weighted_loss2 <- (y_true - y_pred)^2 |>
  op_mean(axis = -1) |>
  op_multiply(sample_weight) |>
  op_mean()

stopifnot(all.equal(as.array(weighted_loss),
                    as.array(weighted_loss2)))
```

Value

A function that returns `Loss` instances, similar to the builtin loss functions.

Methods defined by base `Loss` class:

- `initialize(name=NULL, reduction="sum_over_batch_size", dtype=NULL)`
  Args:
  - `name`
  - `reduction`: Valid values are one of `{"sum_over_batch_size", "sum", NULL, "none"}`
  - `dtype`
- `__call__(y_true, y_pred, sample_weight=NULL)`
  Call the loss instance as a function, optionally with `sample_weight`.
- `get_config()`

Symbols in scope

All R function custom methods (public and private) will have the following symbols in scope:

- `self`: The custom class instance.
- `super`: The custom class superclass.
- `private`: An R environment specific to the class instance. Any objects assigned here are invisible to the Keras framework.
- `__class__` and `as.symbol(classname)`: the custom class type object.

See Also

Other losses:
`loss_binary_crossentropy()`
`loss_binary_focal_crossentropy()`
`loss_categorical_crossentropy()`
`loss_categorical_focal_crossentropy()`
`loss_categorical_hinge()`
`loss_cosine_similarity()`
`loss_dice()`
`loss_hinge()`
`loss_huber()`
`loss_kl_divergence()`
`loss_log_cosh()`
`loss_mean_absolute_error()`
`loss_mean_absolute_percentage_error()`
`loss_mean_squared_error()`

loss_mean_squared_logarithmic_error()
loss_poisson()
loss_sparse_categorical_crossentropy()
loss_squared_hinge()
metric_binary_crossentropy()
metric_binary_focal_crossentropy()
metric_categorical_crossentropy()
metric_categorical_focal_crossentropy()
metric_categorical_hinge()
metric_hinge()
metric_huber()
metric_kl_divergence()
metric_log_cosh()
metric_mean_absolute_error()
metric_mean_absolute_percentage_error()
metric_mean_squared_error()
metric_mean_squared_logarithmic_error()
metric_poisson()
metric_sparse_categorical_crossentropy()
metric_squared_hinge()

---

loss_binary_crossentropy

Computes the cross-entropy loss between true labels and predicted labels.

Description

Use this cross-entropy loss for binary (0 or 1) classification applications. The loss function requires the following inputs:

- y_true (true label): This is either 0 or 1.
- y_pred (predicted value): This is the model's prediction, i.e, a single floating-point value which either represents a logit, (i.e, value in [-inf, inf] when from_logits=TRUE) or a probability (i.e, value in [0., 1.] when from_logits=FALSE).

Usage

```
loss_binary_crossentropy(
  y_true,
  y_pred,
  from_logits = FALSE,
  label_smoothing = 0,
  axis = -1L,
  ...,
  reduction = "sum_over_batch_size",
  name = "binary_crossentropy"
)
```

Arguments

y_true	Ground truth values. shape = [batch_size, d0, .. dN].
y_pred	The predicted values. shape = [batch_size, d0, .. dN].
from_logits	Whether to interpret y_pred as a tensor of logit values. By default, we assume that y_pred is probabilities (i.e., values in [0, 1]).
label_smoothing	Float in range [0, 1]. When 0, no smoothing occurs. When > 0, we compute the loss between the predicted labels and a smoothed version of the true labels, where the smoothing squeezes the labels towards 0.5. Larger values of label_smoothing correspond to heavier smoothing.
axis	The axis along which to compute crossentropy (the features axis). Defaults to -1.
...	For forward/backward compatability.
reduction	Type of reduction to apply to the loss. In almost all cases this should be "sum_over_batch_size". Supported options are "sum", "sum_over_batch_size" or NULL.
name	Optional name for the loss instance.

Value

Binary crossentropy loss value. shape = [batch_size, d0, .. dN-1].

Examples

```
y_true <- rbind(c(0, 1), c(0, 0))
y_pred <- rbind(c(0.6, 0.4), c(0.4, 0.6))
loss <- loss_binary_crossentropy(y_true, y_pred)
loss

## tf.Tensor([0.91629073 0.71355818], shape=(2), dtype=float64)
```

Recommended Usage: (set from_logits=TRUE)

With compile() API:

```
model %>% compile(
    loss = loss_binary_crossentropy(from_logits=TRUE),
    ...
)
```

As a standalone function:

```
# Example 1: (batch_size = 1, number of samples = 4)
y_true <- op_array(c(0, 1, 0, 0))
y_pred <- op_array(c(-18.6, 0.51, 2.94, -12.8))
bce <- loss_binary_crossentropy(from_logits = TRUE)
bce(y_true, y_pred)
```

```
## tf.Tensor(0.865458, shape=(), dtype=float32)
```

```
# Example 2: (batch_size = 2, number of samples = 4)
y_true <- rbind(c(0, 1), c(0, 0))
y_pred <- rbind(c(-18.6, 0.51), c(2.94, -12.8))
# Using default 'auto'/'sum_over_batch_size' reduction type.
bce <- loss_binary_crossentropy(from_logits = TRUE)
bce(y_true, y_pred)
```

```
## tf.Tensor(0.865458, shape=(), dtype=float32)
```

```
# Using 'sample_weight' attribute
bce(y_true, y_pred, sample_weight = c(0.8, 0.2))
```

```
## tf.Tensor(0.2436386, shape=(), dtype=float32)
```

```
# 0.243
# Using 'sum' reduction` type.
bce <- loss_binary_crossentropy(from_logits = TRUE, reduction = "sum")
bce(y_true, y_pred)
```

```
## tf.Tensor(1.730916, shape=(), dtype=float32)
```

```
# Using 'none' reduction type.
bce <- loss_binary_crossentropy(from_logits = TRUE, reduction = NULL)
bce(y_true, y_pred)
```

```
## tf.Tensor([0.23515666 1.4957594 ], shape=(2), dtype=float32)
```

Default Usage: (set `from_logits=FALSE`)

```
# Make the following updates to the above "Recommended Usage" section
# 1. Set `from_logits=FALSE`
loss_binary_crossentropy() # OR ...('from_logits=FALSE')
```

```
## <keras.src.losses.losses.BinaryCrossentropy object>
```

```
# 2. Update `y_pred` to use probabilities instead of logits
y_pred <- c(0.6, 0.3, 0.2, 0.8) # OR [[0.6, 0.3], [0.2, 0.8]]
```

See Also

- https://keras.io/api/losses/probabilistic_losses#binarycrossentropy-class

Other losses:
Loss()
loss_binary_focal_crossentropy()
loss_categorical_crossentropy()
loss_categorical_focal_crossentropy()
loss_categorical_hinge()
loss_cosine_similarity()
loss_dice()
loss_hinge()
loss_huber()
loss_kl_divergence()
loss_log_cosh()
loss_mean_absolute_error()
loss_mean_absolute_percentage_error()
loss_mean_squared_error()
loss_mean_squared_logarithmic_error()
loss_poisson()
loss_sparse_categorical_crossentropy()
loss_squared_hinge()
metric_binary_crossentropy()
metric_binary_focal_crossentropy()
metric_categorical_crossentropy()
metric_categorical_focal_crossentropy()
metric_categorical_hinge()
metric_hinge()
metric_huber()
metric_kl_divergence()
metric_log_cosh()
metric_mean_absolute_error()
metric_mean_absolute_percentage_error()
metric_mean_squared_error()
metric_mean_squared_logarithmic_error()
metric_poisson()
metric_sparse_categorical_crossentropy()
metric_squared_hinge()

---

loss_binary_focal_crossentropy

Computes focal cross-entropy loss between true labels and predictions.

Description

According to Lin et al., 2018, it helps to apply a focal factor to down-weight easy examples and focus more on hard examples. By default, the focal tensor is computed as follows:

`focal_factor = (1 - output)^gamma` for class 1 `focal_factor = output^gamma` for class 0 where `gamma` is a focusing parameter. When `gamma` = 0, there is no focal effect on the binary crossentropy loss.

If `apply_class_balancing == TRUE`, this function also takes into account a weight balancing factor for the binary classes 0 and 1 as follows:

`weight = alpha` for class 1 (`target == 1`) `weight = 1 - alpha` for class 0 where `alpha` is a float in the range of `[0, 1]`.

Binary cross-entropy loss is often used for binary (0 or 1) classification tasks. The loss function requires the following inputs:

- `y_true` (true label): This is either 0 or 1.
- `y_pred` (predicted value): This is the model's prediction, i.e, a single floating-point value which either represents a logit, (i.e, value in `[-inf, inf]` when `from_logits=TRUE`) or a probability (i.e, value in `[0., 1.]` when `from_logits=FALSE`).

According to Lin et al., 2018, it helps to apply a "focal factor" to down-weight easy examples and focus more on hard examples. By default, the focal tensor is computed as follows:

`focal_factor = (1 - output) ** gamma` for class 1 `focal_factor = output ** gamma` for class 0 where `gamma` is a focusing parameter. When `gamma=0`, this function is equivalent to the binary crossentropy loss.

Usage

```
loss_binary_focal_crossentropy(
  y_true,
  y_pred,
  apply_class_balancing = FALSE,
  alpha = 0.25,
  gamma = 2,
  from_logits = FALSE,
  label_smoothing = 0,
  axis = -1L,
  ...,
  reduction = "sum_over_batch_size",
  name = "binary_focal_crossentropy"
)
```

Arguments

`y_true`	Ground truth values, of shape `(batch_size, d0, .. dN)`.
`y_pred`	The predicted values, of shape `(batch_size, d0, .. dN)`.
`apply_class_balancing`	A bool, whether to apply weight balancing on the binary classes 0 and 1.

`alpha`	A weight balancing factor for class 1, default is `0.25` as mentioned in reference Lin et al., 2018. The weight for class 0 is `1.0 - alpha`.
`gamma`	A focusing parameter used to compute the focal factor, default is `2.0` as mentioned in the reference Lin et al., 2018.
`from_logits`	Whether to interpret `y_pred` as a tensor of logit values. By default, we assume that `y_pred` are probabilities (i.e., values in `[0, 1]`).
`label_smoothing`	Float in `[0, 1]`. When `0`, no smoothing occurs. When > `0`, we compute the loss between the predicted labels and a smoothed version of the true labels, where the smoothing squeezes the labels towards `0.5`. Larger values of `label_smoothing` correspond to heavier smoothing.
`axis`	The axis along which to compute crossentropy (the features axis). Defaults to `-1`.
`...`	For forward/backward compatability.
`reduction`	Type of reduction to apply to the loss. In almost all cases this should be `"sum_over_batch_size"`. Supported options are `"sum"`, `"sum_over_batch_size"` or `NULL`.
`name`	Optional name for the loss instance.

Value

Binary focal crossentropy loss value with shape = `[batch_size, d0, .. dN-1]`.

Examples

```
y_true <- rbind(c(0, 1), c(0, 0))
y_pred <- rbind(c(0.6, 0.4), c(0.4, 0.6))
loss <- loss_binary_focal_crossentropy(y_true, y_pred, gamma = 2)
loss

## tf.Tensor([0.32986466 0.20579838], shape=(2), dtype=float64)
```

With the `compile()` API:

```
model %>% compile(
    loss = loss_binary_focal_crossentropy(
        gamma = 2.0, from_logits = TRUE),
    ...
)
```

As a standalone function:

```
# Example 1: (batch_size = 1, number of samples = 4)
y_true <- op_array(c(0, 1, 0, 0))
y_pred <- op_array(c(-18.6, 0.51, 2.94, -12.8))
loss <- loss_binary_focal_crossentropy(gamma = 2, from_logits = TRUE)
loss(y_true, y_pred)
```

```
## tf.Tensor(0.6912122, shape=(), dtype=float32)


# Apply class weight
loss <- loss_binary_focal_crossentropy(
  apply_class_balancing = TRUE, gamma = 2, from_logits = TRUE)
loss(y_true, y_pred)

## tf.Tensor(0.5101333, shape=(), dtype=float32)


# Example 2: (batch_size = 2, number of samples = 4)
y_true <- rbind(c(0, 1), c(0, 0))
y_pred <- rbind(c(-18.6, 0.51), c(2.94, -12.8))
# Using default 'auto'/'sum_over_batch_size' reduction type.
loss <- loss_binary_focal_crossentropy(
    gamma = 3, from_logits = TRUE)
loss(y_true, y_pred)

## tf.Tensor(0.6469951, shape=(), dtype=float32)


# Apply class weight
loss <- loss_binary_focal_crossentropy(
     apply_class_balancing = TRUE, gamma = 3, from_logits = TRUE)
loss(y_true, y_pred)

## tf.Tensor(0.48214132, shape=(), dtype=float32)


# Using 'sample_weight' attribute with focal effect
loss <- loss_binary_focal_crossentropy(
    gamma = 3, from_logits = TRUE)
loss(y_true, y_pred, sample_weight = c(0.8, 0.2))

## tf.Tensor(0.13312504, shape=(), dtype=float32)


# Apply class weight
loss <- loss_binary_focal_crossentropy(
     apply_class_balancing = TRUE, gamma = 3, from_logits = TRUE)
loss(y_true, y_pred, sample_weight = c(0.8, 0.2))

## tf.Tensor(0.09735977, shape=(), dtype=float32)
```

```
# Using 'sum' reduction` type.
loss <- loss_binary_focal_crossentropy(
    gamma = 4, from_logits = TRUE,
    reduction = "sum")
loss(y_true, y_pred)

## tf.Tensor(1.2218808, shape=(), dtype=float32)


# Apply class weight
loss <- loss_binary_focal_crossentropy(
    apply_class_balancing = TRUE, gamma = 4, from_logits = TRUE,
    reduction = "sum")
loss(y_true, y_pred)

## tf.Tensor(0.9140807, shape=(), dtype=float32)


# Using 'none' reduction type.
loss <- loss_binary_focal_crossentropy(
    gamma = 5, from_logits = TRUE,
    reduction = NULL)
loss(y_true, y_pred)

## tf.Tensor([0.00174837 1.1561027 ], shape=(2), dtype=float32)


# Apply class weight
loss <- loss_binary_focal_crossentropy(
    apply_class_balancing = TRUE, gamma = 5, from_logits = TRUE,
    reduction = NULL)
loss(y_true, y_pred)

## tf.Tensor([4.3709317e-04 8.6707699e-01], shape=(2), dtype=float32)
```

See Also

Other losses:
`Loss()`
`loss_binary_crossentropy()`
`loss_categorical_crossentropy()`
`loss_categorical_focal_crossentropy()`
`loss_categorical_hinge()`
`loss_cosine_similarity()`
`loss_dice()`
`loss_hinge()`
`loss_huber()`

loss_kl_divergence()
loss_log_cosh()
loss_mean_absolute_error()
loss_mean_absolute_percentage_error()
loss_mean_squared_error()
loss_mean_squared_logarithmic_error()
loss_poisson()
loss_sparse_categorical_crossentropy()
loss_squared_hinge()
metric_binary_crossentropy()
metric_binary_focal_crossentropy()
metric_categorical_crossentropy()
metric_categorical_focal_crossentropy()
metric_categorical_hinge()
metric_hinge()
metric_huber()
metric_kl_divergence()
metric_log_cosh()
metric_mean_absolute_error()
metric_mean_absolute_percentage_error()
metric_mean_squared_error()
metric_mean_squared_logarithmic_error()
metric_poisson()
metric_sparse_categorical_crossentropy()
metric_squared_hinge()

---

loss_categorical_crossentropy

Computes the crossentropy loss between the labels and predictions.

Description

Use this crossentropy loss function when there are two or more label classes. We expect labels to be provided in a one_hot representation. If you want to provide labels as integers, please use SparseCategoricalCrossentropy loss. There should be num_classes floating point values per feature, i.e., the shape of both y_pred and y_true are [batch_size, num_classes].

Usage

```
loss_categorical_crossentropy(
  y_true,
  y_pred,
  from_logits = FALSE,
  label_smoothing = 0,
  axis = -1L,
  ...,
  reduction = "sum_over_batch_size",
  name = "categorical_crossentropy"
)
```

Arguments

`y_true`	Tensor of one-hot true targets.
`y_pred`	Tensor of predicted targets.
`from_logits`	Whether `y_pred` is expected to be a logits tensor. By default, we assume that `y_pred` encodes a probability distribution.
`label_smoothing`	Float in `[0, 1]`. When > 0, label values are smoothed, meaning the confidence on label values are relaxed. For example, if `0.1`, use `0.1` / `num_classes` for non-target labels and `0.9` + `0.1` / `num_classes` for target labels.
`axis`	The axis along which to compute crossentropy (the features axis). Defaults to `-1`.
`...`	For forward/backward compatability.
`reduction`	Type of reduction to apply to the loss. In almost all cases this should be `"sum_over_batch_size"`. Supported options are `"sum"`, `"sum_over_batch_size"` or `NULL`.
`name`	Optional name for the loss instance.

Value

Categorical crossentropy loss value.

Examples

```
y_true <- rbind(c(0, 1, 0), c(0, 0, 1))
y_pred <- rbind(c(0.05, 0.95, 0), c(0.1, 0.8, 0.1))
loss <- loss_categorical_crossentropy(y_true, y_pred)
loss

## tf.Tensor([0.05129329 2.30258509], shape=(2), dtype=float64)
```

Standalone usage:

```
y_true <- rbind(c(0, 1, 0), c(0, 0, 1))
y_pred <- rbind(c(0.05, 0.95, 0), c(0.1, 0.8, 0.1))
# Using 'auto'/'sum_over_batch_size' reduction type.
cce <- loss_categorical_crossentropy()
cce(y_true, y_pred)

## tf.Tensor(1.1769392, shape=(), dtype=float32)
```

```
# Calling with 'sample_weight'.
cce(y_true, y_pred, sample_weight = op_array(c(0.3, 0.7)))

## tf.Tensor(0.8135988, shape=(), dtype=float32)


# Using 'sum' reduction type.
cce <- loss_categorical_crossentropy(reduction = "sum")
cce(y_true, y_pred)

## tf.Tensor(2.3538785, shape=(), dtype=float32)


# Using 'none' reduction type.
cce <- loss_categorical_crossentropy(reduction = NULL)
cce(y_true, y_pred)

## tf.Tensor([0.05129331 2.3025851 ], shape=(2), dtype=float32)
```

Usage with the `compile()` API:

```
model %>% compile(optimizer = 'sgd',
              loss=loss_categorical_crossentropy())
```

See Also

- https://keras.io/api/losses/probabilistic_losses#categoricalcrossentropy-class

Other losses:
`Loss()`
`loss_binary_crossentropy()`
`loss_binary_focal_crossentropy()`
`loss_categorical_focal_crossentropy()`
`loss_categorical_hinge()`
`loss_cosine_similarity()`
`loss_dice()`
`loss_hinge()`
`loss_huber()`
`loss_kl_divergence()`
`loss_log_cosh()`
`loss_mean_absolute_error()`
`loss_mean_absolute_percentage_error()`
`loss_mean_squared_error()`
`loss_mean_squared_logarithmic_error()`
`loss_poisson()`
`loss_sparse_categorical_crossentropy()`
`loss_squared_hinge()`
`metric_binary_crossentropy()`

metric_binary_focal_crossentropy()
metric_categorical_crossentropy()
metric_categorical_focal_crossentropy()
metric_categorical_hinge()
metric_hinge()
metric_huber()
metric_kl_divergence()
metric_log_cosh()
metric_mean_absolute_error()
metric_mean_absolute_percentage_error()
metric_mean_squared_error()
metric_mean_squared_logarithmic_error()
metric_poisson()
metric_sparse_categorical_crossentropy()
metric_squared_hinge()

---

loss_categorical_focal_crossentropy

Computes the alpha balanced focal crossentropy loss.

Description

Use this crossentropy loss function when there are two or more label classes and if you want to handle class imbalance without using `class_weights`. We expect labels to be provided in a `one_hot` representation.

According to Lin et al., 2018, it helps to apply a focal factor to down-weight easy examples and focus more on hard examples. The general formula for the focal loss (FL) is as follows:

`FL(p_t) = (1 - p_t)^gamma * log(p_t)`

where `p_t` is defined as follows: `p_t = output if y_true == 1, else 1 - output`

`(1 - p_t)^gamma` is the `modulating_factor`, where `gamma` is a focusing parameter. When `gamma` = 0, there is no focal effect on the cross entropy. `gamma` reduces the importance given to simple examples in a smooth manner.

The authors use alpha-balanced variant of focal loss (FL) in the paper: `FL(p_t) = -alpha * (1 - p_t)^gamma * log(p_t)`

where `alpha` is the weight factor for the classes. If `alpha` = 1, the loss won't be able to handle class imbalance properly as all classes will have the same weight. This can be a constant or a list of constants. If alpha is a list, it must have the same length as the number of classes.

The formula above can be generalized to: `FL(p_t) = alpha * (1 - p_t)^gamma * CrossEntropy(y_true, y_pred)`

where minus comes from `CrossEntropy(y_true, y_pred)` (CE).

Extending this to multi-class case is straightforward: `FL(p_t) = alpha * (1 - p_t) ** gamma * CategoricalCE(y_true, y_pred)`

In the snippet below, there is `num_classes` floating pointing values per example. The shape of both `y_pred` and `y_true` are `(batch_size, num_classes)`.

Usage

```
loss_categorical_focal_crossentropy(
  y_true,
  y_pred,
  alpha = 0.25,
  gamma = 2,
  from_logits = FALSE,
  label_smoothing = 0,
  axis = -1L,
  ...,
  reduction = "sum_over_batch_size",
  name = "categorical_focal_crossentropy"
)
```

Arguments

`y_true`	Tensor of one-hot true targets.
`y_pred`	Tensor of predicted targets.
`alpha`	A weight balancing factor for all classes, default is `0.25` as mentioned in the reference. It can be a list of floats or a scalar. In the multi-class case, alpha may be set by inverse class frequency by using `compute_class_weight` from `sklearn.utils`.
`gamma`	A focusing parameter, default is `2.0` as mentioned in the reference. It helps to gradually reduce the importance given to simple examples in a smooth manner. When `gamma` = 0, there is no focal effect on the categorical crossentropy.
`from_logits`	Whether `output` is expected to be a logits tensor. By default, we consider that `output` encodes a probability distribution.
`label_smoothing`	Float in `[0, 1]`. When > 0, label values are smoothed, meaning the confidence on label values are relaxed. For example, if `0.1`, use `0.1` / `num_classes` for non-target labels and `0.9` + `0.1` / `num_classes` for target labels.
`axis`	The axis along which to compute crossentropy (the features axis). Defaults to `-1`.
`...`	For forward/backward compatability.
`reduction`	Type of reduction to apply to the loss. In almost all cases this should be `"sum_over_batch_size"`. Supported options are `"sum"`, `"sum_over_batch_size"` or `NULL`.
`name`	Optional name for the loss instance.

Value

Categorical focal crossentropy loss value.

Examples

```
y_true <- rbind(c(0, 1, 0), c(0, 0, 1))
y_pred <- rbind(c(0.05, 0.95, 0), c(0.1, 0.8, 0.1))
```

```
loss <- loss_categorical_focal_crossentropy(y_true, y_pred)
loss
```

```
## tf.Tensor([3.20583090e-05 4.66273481e-01], shape=(2), dtype=float64)
```

Standalone usage:

```
y_true <- rbind(c(0, 1, 0), c(0, 0, 1))
y_pred <- rbind(c(0.05, 0.95, 0), c(0.1, 0.8, 0.1))
# Using 'auto'/'sum_over_batch_size' reduction type.
cce <- loss_categorical_focal_crossentropy()
cce(y_true, y_pred)
```

```
## tf.Tensor(0.23315276, shape=(), dtype=float32)
```

```
# Calling with 'sample_weight'.
cce(y_true, y_pred, sample_weight = op_array(c(0.3, 0.7)))
```

```
## tf.Tensor(0.16320053, shape=(), dtype=float32)
```

```
# Using 'sum' reduction type.
cce <- loss_categorical_focal_crossentropy(reduction = "sum")
cce(y_true, y_pred)
```

```
## tf.Tensor(0.46630552, shape=(), dtype=float32)
```

```
# Using 'none' reduction type.
cce <- loss_categorical_focal_crossentropy(reduction = NULL)
cce(y_true, y_pred)
```

```
## tf.Tensor([3.2058331e-05 4.6627346e-01], shape=(2), dtype=float32)
```

Usage with the `compile()` API:

```
model %>% compile(
  optimizer = 'adam',
  loss = loss_categorical_focal_crossentropy())
```

See Also

Other losses:
Loss()
loss_binary_crossentropy()
loss_binary_focal_crossentropy()
loss_categorical_crossentropy()
loss_categorical_hinge()
loss_cosine_similarity()
loss_dice()
loss_hinge()
loss_huber()
loss_kl_divergence()
loss_log_cosh()
loss_mean_absolute_error()
loss_mean_absolute_percentage_error()
loss_mean_squared_error()
loss_mean_squared_logarithmic_error()
loss_poisson()
loss_sparse_categorical_crossentropy()
loss_squared_hinge()
metric_binary_crossentropy()
metric_binary_focal_crossentropy()
metric_categorical_crossentropy()
metric_categorical_focal_crossentropy()
metric_categorical_hinge()
metric_hinge()
metric_huber()
metric_kl_divergence()
metric_log_cosh()
metric_mean_absolute_error()
metric_mean_absolute_percentage_error()
metric_mean_squared_error()
metric_mean_squared_logarithmic_error()
metric_poisson()
metric_sparse_categorical_crossentropy()
metric_squared_hinge()

---

loss_categorical_hinge

Computes the categorical hinge loss between y_true *&* y_pred.

Description

Formula:

```
loss <- maximum(neg - pos + 1, 0)
```

where neg=maximum((1-y_true)*y_pred) and pos=sum(y_true*y_pred)

Usage

```
loss_categorical_hinge(
  y_true,
  y_pred,
  ...,
  reduction = "sum_over_batch_size",
  name = "categorical_hinge"
)
```

Arguments

y_true	The ground truth values. y_true values are expected to be either {-1, +1} or {0, 1} (i.e. a one-hot-encoded tensor) with shape <- [batch_size, d0, .. dN].
y_pred	The predicted values with shape = [batch_size, d0, .. dN].
...	For forward/backward compatability.
reduction	Type of reduction to apply to the loss. In almost all cases this should be "sum_over_batch_size". Supported options are "sum", "sum_over_batch_size" or NULL.
name	Optional name for the loss instance.

Value

Categorical hinge loss values with shape = [batch_size, d0, .. dN-1].

Examples

```
y_true <- rbind(c(0, 1), c(0, 0))
y_pred <- rbind(c(0.6, 0.4), c(0.4, 0.6))
loss <- loss_categorical_hinge(y_true, y_pred)
```

See Also

- https://keras.io/api/losses/hinge_losses#categoricalhinge-class

Other losses:
Loss()
loss_binary_crossentropy()
loss_binary_focal_crossentropy()
loss_categorical_crossentropy()
loss_categorical_focal_crossentropy()
loss_cosine_similarity()
loss_dice()
loss_hinge()
loss_huber()
loss_kl_divergence()
loss_log_cosh()
loss_mean_absolute_error()

loss_mean_absolute_percentage_error()
loss_mean_squared_error()
loss_mean_squared_logarithmic_error()
loss_poisson()
loss_sparse_categorical_crossentropy()
loss_squared_hinge()
metric_binary_crossentropy()
metric_binary_focal_crossentropy()
metric_categorical_crossentropy()
metric_categorical_focal_crossentropy()
metric_categorical_hinge()
metric_hinge()
metric_huber()
metric_kl_divergence()
metric_log_cosh()
metric_mean_absolute_error()
metric_mean_absolute_percentage_error()
metric_mean_squared_error()
metric_mean_squared_logarithmic_error()
metric_poisson()
metric_sparse_categorical_crossentropy()
metric_squared_hinge()

loss_cosine_similarity

Computes the cosine similarity between `y_true` *&* `y_pred`*.*

Description

Formula:

```
loss <- -sum(l2_norm(y_true) * l2_norm(y_pred))
```

Note that it is a number between -1 and 1. When it is a negative number between -1 and 0, 0 indicates orthogonality and values closer to -1 indicate greater similarity. This makes it usable as a loss function in a setting where you try to maximize the proximity between predictions and targets. If either `y_true` or `y_pred` is a zero vector, cosine similarity will be 0 regardless of the proximity between predictions and targets.

Usage

```
loss_cosine_similarity(
  y_true,
  y_pred,
  axis = -1L,
  ...,
  reduction = "sum_over_batch_size",
  name = "cosine_similarity"
)
```

Arguments

y_true	Tensor of true targets.
y_pred	Tensor of predicted targets.
axis	The axis along which the cosine similarity is computed (the features axis). Defaults to -1.
...	For forward/backward compatability.
reduction	Type of reduction to apply to the loss. In almost all cases this should be "sum_over_batch_size". Supported options are "sum", "sum_over_batch_size" or NULL.
name	Optional name for the loss instance.

Value

Cosine similarity tensor.

Examples

```
y_true <- rbind(c(0., 1.), c(1., 1.), c(1., 1.))
y_pred <- rbind(c(1., 0.), c(1., 1.), c(-1., -1.))
loss <- loss_cosine_similarity(y_true, y_pred, axis=-1)
loss

## tf.Tensor([-0. -1.  1.], shape=(3), dtype=float64)
```

See Also

- https://keras.io/api/losses/regression_losses#cosinesimilarity-class

Other losses:
Loss()
loss_binary_crossentropy()
loss_binary_focal_crossentropy()
loss_categorical_crossentropy()
loss_categorical_focal_crossentropy()
loss_categorical_hinge()
loss_dice()
loss_hinge()
loss_huber()
loss_kl_divergence()
loss_log_cosh()
loss_mean_absolute_error()
loss_mean_absolute_percentage_error()
loss_mean_squared_error()

loss_mean_squared_logarithmic_error()
loss_poisson()
loss_sparse_categorical_crossentropy()
loss_squared_hinge()
metric_binary_crossentropy()
metric_binary_focal_crossentropy()
metric_categorical_crossentropy()
metric_categorical_focal_crossentropy()
metric_categorical_hinge()
metric_hinge()
metric_huber()
metric_kl_divergence()
metric_log_cosh()
metric_mean_absolute_error()
metric_mean_absolute_percentage_error()
metric_mean_squared_error()
metric_mean_squared_logarithmic_error()
metric_poisson()
metric_sparse_categorical_crossentropy()
metric_squared_hinge()

---

loss_ctc *CTC (Connectionist Temporal Classification) loss.*

Description

CTC (Connectionist Temporal Classification) loss.

Usage

```
loss_ctc(
  y_true,
  y_pred,
  ...,
  reduction = "sum_over_batch_size",
  name = "sparse_categorical_crossentropy"
)
```

Arguments

y_true	A tensor of shape (batch_size, target_max_length) containing the true labels in integer format. 0 always represents the blank/mask index and should not be used for classes.
y_pred	A tensor of shape (batch_size, output_max_length, num_classes) containing logits (the output of your model). They should *not* be normalized via softmax.

`...`	For forward/backward compatability.
`reduction`	Type of reduction to apply to the loss. In almost all cases this should be `"sum_over_batch_size"`. Supported options are `"sum"`, `"sum_over_batch_size"` or `NULL`.
`name`	String, name for the object

Value

CTC loss value.

`loss_dice` *Computes the Dice loss value between* `y_true` *and* `y_pred`*.*

Description

Formula:

```
loss = 1 - (2 * sum(y_true * y_pred)) / (sum(y_true) + sum(y_pred))
```

Formula:

```
loss = 1 - (2 * sum(y_true * y_pred)) / (sum(y_true) + sum(y_pred))
```

Usage

```
loss_dice(
  y_true,
  y_pred,
  ...,
  reduction = "sum_over_batch_size",
  name = "dice"
)
```

Arguments

`y_true`	tensor of true targets.
`y_pred`	tensor of predicted targets.
`...`	For forward/backward compatability.
`reduction`	Type of reduction to apply to the loss. In almost all cases this should be `"sum_over_batch_size"`. Supported options are `"sum"`, `"sum_over_batch_size"` or `NULL`.
`name`	String, name for the object

Value

if `y_true` and `y_pred` are provided, Dice loss value. Otherwise, a `Loss()` instance.

See Also

Other losses:
Loss()
loss_binary_crossentropy()
loss_binary_focal_crossentropy()
loss_categorical_crossentropy()
loss_categorical_focal_crossentropy()
loss_categorical_hinge()
loss_cosine_similarity()
loss_hinge()
loss_huber()
loss_kl_divergence()
loss_log_cosh()
loss_mean_absolute_error()
loss_mean_absolute_percentage_error()
loss_mean_squared_error()
loss_mean_squared_logarithmic_error()
loss_poisson()
loss_sparse_categorical_crossentropy()
loss_squared_hinge()
metric_binary_crossentropy()
metric_binary_focal_crossentropy()
metric_categorical_crossentropy()
metric_categorical_focal_crossentropy()
metric_categorical_hinge()
metric_hinge()
metric_huber()
metric_kl_divergence()
metric_log_cosh()
metric_mean_absolute_error()
metric_mean_absolute_percentage_error()
metric_mean_squared_error()
metric_mean_squared_logarithmic_error()
metric_poisson()
metric_sparse_categorical_crossentropy()
metric_squared_hinge()

---

loss_hinge *Computes the hinge loss between* y_true *&* y_pred.

Description

Formula:

```
loss <- mean(maximum(1 - y_true * y_pred, 0), axis=-1)
```

y_true values are expected to be -1 or 1. If binary (0 or 1) labels are provided we will convert them to -1 or 1.

Usage

```
loss_hinge(
  y_true,
  y_pred,
  ...,
  reduction = "sum_over_batch_size",
  name = "hinge"
)
```

Arguments

y_true	The ground truth values. y_true values are expected to be -1 or 1. If binary (0 or 1) labels are provided they will be converted to -1 or 1 with shape = [batch_size, d0, .. dN].
y_pred	The predicted values with shape = [batch_size, d0, .. dN].
...	For forward/backward compatability.
reduction	Type of reduction to apply to the loss. In almost all cases this should be "sum_over_batch_size". Supported options are "sum", "sum_over_batch_size" or NULL.
name	Optional name for the loss instance.

Value

Hinge loss values with shape = [batch_size, d0, .. dN-1].

Examples

```
y_true <- array(sample(c(-1,1), 6, replace = TRUE), dim = c(2, 3))
y_pred <- random_uniform(c(2, 3))
loss <- loss_hinge(y_true, y_pred)
loss

## tf.Tensor([1.0610152  0.93285507], shape=(2), dtype=float32)
```

See Also

- https://keras.io/api/losses/hinge_losses#hinge-class

Other losses:
Loss()
loss_binary_crossentropy()
loss_binary_focal_crossentropy()
loss_categorical_crossentropy()
loss_categorical_focal_crossentropy()
loss_categorical_hinge()

loss_cosine_similarity()
loss_dice()
loss_huber()
loss_kl_divergence()
loss_log_cosh()
loss_mean_absolute_error()
loss_mean_absolute_percentage_error()
loss_mean_squared_error()
loss_mean_squared_logarithmic_error()
loss_poisson()
loss_sparse_categorical_crossentropy()
loss_squared_hinge()
metric_binary_crossentropy()
metric_binary_focal_crossentropy()
metric_categorical_crossentropy()
metric_categorical_focal_crossentropy()
metric_categorical_hinge()
metric_hinge()
metric_huber()
metric_kl_divergence()
metric_log_cosh()
metric_mean_absolute_error()
metric_mean_absolute_percentage_error()
metric_mean_squared_error()
metric_mean_squared_logarithmic_error()
metric_poisson()
metric_sparse_categorical_crossentropy()
metric_squared_hinge()

loss_huber *Computes the Huber loss between* y_true *&* y_pred.

Description

Formula:

```
for (x in error) {
  if (abs(x) <= delta){
    loss <- c(loss, (0.5 * x^2))
  } else if (abs(x) > delta) {
    loss <- c(loss, (delta * abs(x) - 0.5 * delta^2))
  }
}
loss <- mean(loss)
```

See: Huber loss.

Usage

```
loss_huber(
  y_true,
  y_pred,
  delta = 1,
  ...,
  reduction = "sum_over_batch_size",
  name = "huber_loss"
)
```

Arguments

y_true	tensor of true targets.
y_pred	tensor of predicted targets.
delta	A float, the point where the Huber loss function changes from a quadratic to linear. Defaults to 1.0.
...	For forward/backward compatability.
reduction	Type of reduction to apply to loss. Options are "sum", "sum_over_batch_size" or NULL. Defaults to "sum_over_batch_size".
name	Optional name for the instance.

Value

Tensor with one scalar loss entry per sample.

Examples

```
y_true <- rbind(c(0, 1), c(0, 0))
y_pred <- rbind(c(0.6, 0.4), c(0.4, 0.6))
loss <- loss_huber(y_true, y_pred)
```

See Also

- https://keras.io/api/losses/regression_losses#huber-class

Other losses:
Loss()
loss_binary_crossentropy()
loss_binary_focal_crossentropy()
loss_categorical_crossentropy()
loss_categorical_focal_crossentropy()
loss_categorical_hinge()
loss_cosine_similarity()
loss_dice()
loss_hinge()
loss_kl_divergence()
loss_log_cosh()
loss_mean_absolute_error()

loss_mean_absolute_percentage_error()
loss_mean_squared_error()
loss_mean_squared_logarithmic_error()
loss_poisson()
loss_sparse_categorical_crossentropy()
loss_squared_hinge()
metric_binary_crossentropy()
metric_binary_focal_crossentropy()
metric_categorical_crossentropy()
metric_categorical_focal_crossentropy()
metric_categorical_hinge()
metric_hinge()
metric_huber()
metric_kl_divergence()
metric_log_cosh()
metric_mean_absolute_error()
metric_mean_absolute_percentage_error()
metric_mean_squared_error()
metric_mean_squared_logarithmic_error()
metric_poisson()
metric_sparse_categorical_crossentropy()
metric_squared_hinge()

---

loss_kl_divergence   *Computes Kullback-Leibler divergence loss between* y_true *&* y_pred.

Description

Formula:

```
loss <- y_true * log(y_true / y_pred)
```

Usage

```
loss_kl_divergence(
  y_true,
  y_pred,
  ...,
  reduction = "sum_over_batch_size",
  name = "kl_divergence"
)
```

Arguments

y_true	Tensor of true targets.
y_pred	Tensor of predicted targets.
...	For forward/backward compatability.
reduction	Type of reduction to apply to the loss. In almost all cases this should be "sum_over_batch_size". Supported options are "sum", "sum_over_batch_size" or NULL.
name	Optional name for the loss instance.

Value

KL Divergence loss values with shape = `[batch_size, d0, .. dN-1]`.

Examples

```
y_true <- random_uniform(c(2, 3), 0, 2)
y_pred <- random_uniform(c(2,3))
loss <- loss_kl_divergence(y_true, y_pred)
loss

## tf.Tensor([3.5312676 0.2128672], shape=(2), dtype=float32)
```

See Also

- https://keras.io/api/losses/probabilistic_losses#kldivergence-class

Other losses:
Loss()
loss_binary_crossentropy()
loss_binary_focal_crossentropy()
loss_categorical_crossentropy()
loss_categorical_focal_crossentropy()
loss_categorical_hinge()
loss_cosine_similarity()
loss_dice()
loss_hinge()
loss_huber()
loss_log_cosh()
loss_mean_absolute_error()
loss_mean_absolute_percentage_error()
loss_mean_squared_error()
loss_mean_squared_logarithmic_error()
loss_poisson()
loss_sparse_categorical_crossentropy()
loss_squared_hinge()
metric_binary_crossentropy()
metric_binary_focal_crossentropy()
metric_categorical_crossentropy()

metric_categorical_focal_crossentropy()
metric_categorical_hinge()
metric_hinge()
metric_huber()
metric_kl_divergence()
metric_log_cosh()
metric_mean_absolute_error()
metric_mean_absolute_percentage_error()
metric_mean_squared_error()
metric_mean_squared_logarithmic_error()
metric_poisson()
metric_sparse_categorical_crossentropy()
metric_squared_hinge()

---

loss_log_cosh — *Computes the logarithm of the hyperbolic cosine of the prediction error.*

Description

Formula:

```
loss <- mean(log(cosh(y_pred - y_true)), axis=-1)
```

Note that `log(cosh(x))` is approximately equal to `(x ** 2) / 2` for small `x` and to `abs(x) - log(2)` for large `x`. This means that 'logcosh' works mostly like the mean squared error, but will not be so strongly affected by the occasional wildly incorrect prediction.

Usage

```
loss_log_cosh(
  y_true,
  y_pred,
  ...,
  reduction = "sum_over_batch_size",
  name = "log_cosh"
)
```

Arguments

`y_true`	Ground truth values with shape = `[batch_size, d0, .. dN]`.
`y_pred`	The predicted values with shape = `[batch_size, d0, .. dN]`.
`...`	For forward/backward compatability.
`reduction`	Type of reduction to apply to loss. Options are `"sum"`, `"sum_over_batch_size"` or `NULL`. Defaults to `"sum_over_batch_size"`.
`name`	Optional name for the instance.

Value

Logcosh error values with shape = `[batch_size, d0, .. dN-1]`.

Examples

```
y_true <- rbind(c(0., 1.), c(0., 0.))
y_pred <- rbind(c(1., 1.), c(0., 0.))
loss <- loss_log_cosh(y_true, y_pred)
# 0.108
```

See Also

- https://keras.io/api/losses/regression_losses#logcosh-class

Other losses:
`Loss()`
`loss_binary_crossentropy()`
`loss_binary_focal_crossentropy()`
`loss_categorical_crossentropy()`
`loss_categorical_focal_crossentropy()`
`loss_categorical_hinge()`
`loss_cosine_similarity()`
`loss_dice()`
`loss_hinge()`
`loss_huber()`
`loss_kl_divergence()`
`loss_mean_absolute_error()`
`loss_mean_absolute_percentage_error()`
`loss_mean_squared_error()`
`loss_mean_squared_logarithmic_error()`
`loss_poisson()`
`loss_sparse_categorical_crossentropy()`
`loss_squared_hinge()`
`metric_binary_crossentropy()`
`metric_binary_focal_crossentropy()`
`metric_categorical_crossentropy()`
`metric_categorical_focal_crossentropy()`
`metric_categorical_hinge()`
`metric_hinge()`
`metric_huber()`
`metric_kl_divergence()`
`metric_log_cosh()`
`metric_mean_absolute_error()`
`metric_mean_absolute_percentage_error()`
`metric_mean_squared_error()`
`metric_mean_squared_logarithmic_error()`
`metric_poisson()`
`metric_sparse_categorical_crossentropy()`

metric_squared_hinge()

loss_mean_absolute_error

Computes the mean of absolute difference between labels and predictions.

Description

Formula:

```
loss <- mean(abs(y_true - y_pred))
```

Usage

```
loss_mean_absolute_error(
  y_true,
  y_pred,
  ...,
  reduction = "sum_over_batch_size",
  name = "mean_absolute_error"
)
```

Arguments

y_true	Ground truth values with shape = [batch_size, d0, .. dN].
y_pred	The predicted values with shape = [batch_size, d0, .. dN].
...	For forward/backward compatability.
reduction	Type of reduction to apply to the loss. In almost all cases this should be "sum_over_batch_size". Supported options are "sum", "sum_over_batch_size" or NULL.
name	Optional name for the loss instance.

Value

Mean absolute error values with shape = [batch_size, d0, .. dN-1].

Examples

```
y_true <- random_uniform(c(2, 3), 0, 2)
y_pred <- random_uniform(c(2, 3))
loss <- loss_mean_absolute_error(y_true, y_pred)
```

See Also

- https://keras.io/api/losses/regression_losses#meanabsoluteerror-class

Other losses:
Loss()
loss_binary_crossentropy()
loss_binary_focal_crossentropy()
loss_categorical_crossentropy()
loss_categorical_focal_crossentropy()
loss_categorical_hinge()
loss_cosine_similarity()
loss_dice()
loss_hinge()
loss_huber()
loss_kl_divergence()
loss_log_cosh()
loss_mean_absolute_percentage_error()
loss_mean_squared_error()
loss_mean_squared_logarithmic_error()
loss_poisson()
loss_sparse_categorical_crossentropy()
loss_squared_hinge()
metric_binary_crossentropy()
metric_binary_focal_crossentropy()
metric_categorical_crossentropy()
metric_categorical_focal_crossentropy()
metric_categorical_hinge()
metric_hinge()
metric_huber()
metric_kl_divergence()
metric_log_cosh()
metric_mean_absolute_error()
metric_mean_absolute_percentage_error()
metric_mean_squared_error()
metric_mean_squared_logarithmic_error()
metric_poisson()
metric_sparse_categorical_crossentropy()
metric_squared_hinge()

---

loss_mean_absolute_percentage_error
Computes the mean absolute percentage error between y_true *and* y_pred.

Description

Formula:

```
loss <- 100 * op_mean(op_abs((y_true - y_pred) / y_true),
                      axis=-1)
```

Division by zero is prevented by dividing by `max(y_true, epsilon)` where `epsilon` = `config_epsilon()` (default to `1e-7`).

Usage

```
loss_mean_absolute_percentage_error(
  y_true,
  y_pred,
  ...,
  reduction = "sum_over_batch_size",
  name = "mean_absolute_percentage_error"
)
```

Arguments

`y_true`	Ground truth values with shape = `[batch_size, d0, .. dN]`.
`y_pred`	The predicted values with shape = `[batch_size, d0, .. dN]`.
`...`	For forward/backward compatability.
`reduction`	Type of reduction to apply to the loss. In almost all cases this should be `"sum_over_batch_size"`. Supported options are `"sum"`, `"sum_over_batch_size"` or `NULL`.
`name`	Optional name for the loss instance.

Value

Mean absolute percentage error values with shape = `[batch_size, d0, ..dN-1]`.

Examples

```
y_true <- random_uniform(c(2, 3))
y_pred <- random_uniform(c(2, 3))
loss <- loss_mean_absolute_percentage_error(y_true, y_pred)
```

See Also

- `https://keras.io/api/losses/regression_losses#meanabsolutepercentageerror-class`

Other losses:
`Loss()`
`loss_binary_crossentropy()`
`loss_binary_focal_crossentropy()`
`loss_categorical_crossentropy()`
`loss_categorical_focal_crossentropy()`

loss_categorical_hinge()
loss_cosine_similarity()
loss_dice()
loss_hinge()
loss_huber()
loss_kl_divergence()
loss_log_cosh()
loss_mean_absolute_error()
loss_mean_squared_error()
loss_mean_squared_logarithmic_error()
loss_poisson()
loss_sparse_categorical_crossentropy()
loss_squared_hinge()
metric_binary_crossentropy()
metric_binary_focal_crossentropy()
metric_categorical_crossentropy()
metric_categorical_focal_crossentropy()
metric_categorical_hinge()
metric_hinge()
metric_huber()
metric_kl_divergence()
metric_log_cosh()
metric_mean_absolute_error()
metric_mean_absolute_percentage_error()
metric_mean_squared_error()
metric_mean_squared_logarithmic_error()
metric_poisson()
metric_sparse_categorical_crossentropy()
metric_squared_hinge()

loss_mean_squared_error

Computes the mean of squares of errors between labels and predictions.

Description

Formula:

```
loss <- mean(square(y_true - y_pred))
```

Usage

```
loss_mean_squared_error(
  y_true,
  y_pred,
  ...,
  reduction = "sum_over_batch_size",
  name = "mean_squared_error"
)
```

Arguments

y_true	Ground truth values with shape = [batch_size, d0, .. dN].
y_pred	The predicted values with shape = [batch_size, d0, .. dN].
...	For forward/backward compatability.
reduction	Type of reduction to apply to the loss. In almost all cases this should be "sum_over_batch_size". Supported options are "sum", "sum_over_batch_size" or NULL.
name	Optional name for the loss instance.

Value

Mean squared error values with shape = [batch_size, d0, .. dN-1].

Examples

```
y_true <- random_uniform(c(2, 3), 0, 2)
y_pred <- random_uniform(c(2, 3))
loss <- loss_mean_squared_error(y_true, y_pred)
```

See Also

- https://keras.io/api/losses/regression_losses#meansquarederror-class

Other losses:
Loss()
loss_binary_crossentropy()
loss_binary_focal_crossentropy()
loss_categorical_crossentropy()
loss_categorical_focal_crossentropy()
loss_categorical_hinge()
loss_cosine_similarity()
loss_dice()
loss_hinge()
loss_huber()
loss_kl_divergence()
loss_log_cosh()
loss_mean_absolute_error()
loss_mean_absolute_percentage_error()
loss_mean_squared_logarithmic_error()
loss_poisson()
loss_sparse_categorical_crossentropy()
loss_squared_hinge()
metric_binary_crossentropy()
metric_binary_focal_crossentropy()

metric_categorical_crossentropy()
metric_categorical_focal_crossentropy()
metric_categorical_hinge()
metric_hinge()
metric_huber()
metric_kl_divergence()
metric_log_cosh()
metric_mean_absolute_error()
metric_mean_absolute_percentage_error()
metric_mean_squared_error()
metric_mean_squared_logarithmic_error()
metric_poisson()
metric_sparse_categorical_crossentropy()
metric_squared_hinge()

---

loss_mean_squared_logarithmic_error
Computes the mean squared logarithmic error between y_true *and* y_pred*.*

Description

Note that y_pred and y_true cannot be less or equal to 0. Negative values and 0 values will be replaced with config_epsilon() (default to 1e-7).

Formula:

```
loss <- mean(square(log(y_true + 1) - log(y_pred + 1)))
```

Usage

```
loss_mean_squared_logarithmic_error(
  y_true,
  y_pred,
  ...,
  reduction = "sum_over_batch_size",
  name = "mean_squared_logarithmic_error"
)
```

Arguments

y_true	Ground truth values with shape = [batch_size, d0, .. dN].
y_pred	The predicted values with shape = [batch_size, d0, .. dN].
...	For forward/backward compatability.
reduction	Type of reduction to apply to the loss. In almost all cases this should be "sum_over_batch_size". Supported options are "sum", "sum_over_batch_size" or NULL.
name	Optional name for the loss instance.

Value

Mean squared logarithmic error values with shape = `[batch_size, d0, .. dN-1]`.

Examples

```
y_true <- random_uniform(c(2, 3), 0, 2)
y_pred <- random_uniform(c(2, 3))
loss <- loss_mean_squared_logarithmic_error(y_true, y_pred)
```

See Also

- https://keras.io/api/losses/regression_losses#meansquaredlogarithmicerror-class

Other losses:
Loss()
loss_binary_crossentropy()
loss_binary_focal_crossentropy()
loss_categorical_crossentropy()
loss_categorical_focal_crossentropy()
loss_categorical_hinge()
loss_cosine_similarity()
loss_dice()
loss_hinge()
loss_huber()
loss_kl_divergence()
loss_log_cosh()
loss_mean_absolute_error()
loss_mean_absolute_percentage_error()
loss_mean_squared_error()
loss_poisson()
loss_sparse_categorical_crossentropy()
loss_squared_hinge()
metric_binary_crossentropy()
metric_binary_focal_crossentropy()
metric_categorical_crossentropy()
metric_categorical_focal_crossentropy()
metric_categorical_hinge()
metric_hinge()
metric_huber()
metric_kl_divergence()
metric_log_cosh()
metric_mean_absolute_error()
metric_mean_absolute_percentage_error()
metric_mean_squared_error()
metric_mean_squared_logarithmic_error()
metric_poisson()
metric_sparse_categorical_crossentropy()
metric_squared_hinge()

loss_poisson *Computes the Poisson loss between* `y_true` *&* `y_pred`*.*

Description

Formula:

```
loss <- y_pred - y_true * log(y_pred)
```

Usage

```
loss_poisson(
  y_true,
  y_pred,
  ...,
  reduction = "sum_over_batch_size",
  name = "poisson"
)
```

Arguments

`y_true`	Ground truth values. shape = `[batch_size, d0, .. dN]`.
`y_pred`	The predicted values. shape = `[batch_size, d0, .. dN]`.
`...`	For forward/backward compatability.
`reduction`	Type of reduction to apply to the loss. In almost all cases this should be `"sum_over_batch_size"`. Supported options are `"sum"`, `"sum_over_batch_size"` or `NULL`.
`name`	Optional name for the loss instance.

Value

Poisson loss values with shape = `[batch_size, d0, .. dN-1]`.

Examples

```
y_true <- random_uniform(c(2, 3), 0, 2)
y_pred <- random_uniform(c(2, 3))
loss <- loss_poisson(y_true, y_pred)
loss
```

```
## tf.Tensor([2.5907533  0.66836613], shape=(2), dtype=float32)
```

See Also

- https://keras.io/api/losses/probabilistic_losses#poisson-class

Other losses:
`Loss()`
`loss_binary_crossentropy()`
`loss_binary_focal_crossentropy()`
`loss_categorical_crossentropy()`
`loss_categorical_focal_crossentropy()`
`loss_categorical_hinge()`
`loss_cosine_similarity()`
`loss_dice()`
`loss_hinge()`
`loss_huber()`
`loss_kl_divergence()`
`loss_log_cosh()`
`loss_mean_absolute_error()`
`loss_mean_absolute_percentage_error()`
`loss_mean_squared_error()`
`loss_mean_squared_logarithmic_error()`
`loss_sparse_categorical_crossentropy()`
`loss_squared_hinge()`
`metric_binary_crossentropy()`
`metric_binary_focal_crossentropy()`
`metric_categorical_crossentropy()`
`metric_categorical_focal_crossentropy()`
`metric_categorical_hinge()`
`metric_hinge()`
`metric_huber()`
`metric_kl_divergence()`
`metric_log_cosh()`
`metric_mean_absolute_error()`
`metric_mean_absolute_percentage_error()`
`metric_mean_squared_error()`
`metric_mean_squared_logarithmic_error()`
`metric_poisson()`
`metric_sparse_categorical_crossentropy()`
`metric_squared_hinge()`

`loss_sparse_categorical_crossentropy`

Computes the crossentropy loss between the labels and predictions.

Description

Use this crossentropy loss function when there are two or more label classes. We expect labels to be provided as integers. If you want to provide labels using `one-hot` representation, please use

CategoricalCrossentropy loss. There should be # classes floating point values per feature for y_pred and a single floating point value per feature for y_true.

In the snippet below, there is a single floating point value per example for y_true and num_classes floating pointing values per example for y_pred. The shape of y_true is [batch_size] and the shape of y_pred is [batch_size, num_classes].

Usage

```
loss_sparse_categorical_crossentropy(
  y_true,
  y_pred,
  from_logits = FALSE,
  ignore_class = NULL,
  axis = -1L,
  ...,
  reduction = "sum_over_batch_size",
  name = "sparse_categorical_crossentropy"
)
```

Arguments

y_true	Ground truth values.
y_pred	The predicted values.
from_logits	Whether y_pred is expected to be a logits tensor. By default, we assume that y_pred encodes a probability distribution.
ignore_class	Optional integer. The ID of a class to be ignored during loss computation. This is useful, for example, in segmentation problems featuring a "void" class (commonly -1 or 255) in segmentation maps. By default (ignore_class=NULL), all classes are considered.
axis	Defaults to -1. The dimension along which the entropy is computed.
...	For forward/backward compatability.
reduction	Type of reduction to apply to the loss. In almost all cases this should be "sum_over_batch_size". Supported options are "sum", "sum_over_batch_size" or NULL.
name	Optional name for the loss instance.

Value

Sparse categorical crossentropy loss value.

Examples

```
y_true <- c(1, 2)
y_pred <- rbind(c(0.05, 0.95, 0), c(0.1, 0.8, 0.1))
loss <- loss_sparse_categorical_crossentropy(y_true, y_pred)
loss

## tf.Tensor([0.05129339 2.30258509], shape=(2), dtype=float64)
```

```
y_true <- c(1, 2)
y_pred <- rbind(c(0.05, 0.95, 0), c(0.1, 0.8, 0.1))
# Using 'auto'/'sum_over_batch_size' reduction type.
scce <- loss_sparse_categorical_crossentropy()
scce(op_array(y_true), op_array(y_pred))
```

```
## tf.Tensor(1.1769392, shape=(), dtype=float32)
```

```
# 1.177

# Calling with 'sample_weight'.
scce(op_array(y_true), op_array(y_pred), sample_weight = op_array(c(0.3, 0.7)))
```

```
## tf.Tensor(0.8135988, shape=(), dtype=float32)
```

```
# Using 'sum' reduction type.
scce <- loss_sparse_categorical_crossentropy(reduction="sum")
scce(op_array(y_true), op_array(y_pred))
```

```
## tf.Tensor(2.3538785, shape=(), dtype=float32)
```

```
# 2.354

# Using 'none' reduction type.
scce <- loss_sparse_categorical_crossentropy(reduction=NULL)
scce(op_array(y_true), op_array(y_pred))
```

```
## tf.Tensor([0.05129344 2.3025851 ], shape=(2), dtype=float32)
```

```
# array([0.0513, 2.303], dtype=float32)
```

Usage with the `compile()` API:

```
model %>% compile(optimizer = 'sgd',
                  loss = loss_sparse_categorical_crossentropy())
```

See Also

- https://keras.io/api/losses/probabilistic_losses#sparsecategoricalcrossentropy-class

Other losses:
`Loss()`
`loss_binary_crossentropy()`
`loss_binary_focal_crossentropy()`

loss_categorical_crossentropy()
loss_categorical_focal_crossentropy()
loss_categorical_hinge()
loss_cosine_similarity()
loss_dice()
loss_hinge()
loss_huber()
loss_kl_divergence()
loss_log_cosh()
loss_mean_absolute_error()
loss_mean_absolute_percentage_error()
loss_mean_squared_error()
loss_mean_squared_logarithmic_error()
loss_poisson()
loss_squared_hinge()
metric_binary_crossentropy()
metric_binary_focal_crossentropy()
metric_categorical_crossentropy()
metric_categorical_focal_crossentropy()
metric_categorical_hinge()
metric_hinge()
metric_huber()
metric_kl_divergence()
metric_log_cosh()
metric_mean_absolute_error()
metric_mean_absolute_percentage_error()
metric_mean_squared_error()
metric_mean_squared_logarithmic_error()
metric_poisson()
metric_sparse_categorical_crossentropy()
metric_squared_hinge()

---

loss_squared_hinge   *Computes the squared hinge loss between* y_true *&* y_pred*.*

Description

Formula:

```
loss <- square(maximum(1 - y_true * y_pred, 0))
```

y_true values are expected to be -1 or 1. If binary (0 or 1) labels are provided we will convert them to -1 or 1.

Usage

```
loss_squared_hinge(
  y_true,
  y_pred,
  ...,
  reduction = "sum_over_batch_size",
  name = "squared_hinge"
)
```

Arguments

`y_true`	The ground truth values. `y_true` values are expected to be -1 or 1. If binary (0 or 1) labels are provided we will convert them to -1 or 1 with shape = `[batch_size, d0, .. dN]`.
`y_pred`	The predicted values with shape = `[batch_size, d0, .. dN]`.
`...`	For forward/backward compatability.
`reduction`	Type of reduction to apply to the loss. In almost all cases this should be `"sum_over_batch_size"`. Supported options are `"sum"`, `"sum_over_batch_size"` or `NULL`.
`name`	Optional name for the loss instance.

Value

Squared hinge loss values with shape = `[batch_size, d0, .. dN-1]`.

Examples

```
y_true <- array(sample(c(-1,1), 6, replace = TRUE), dim = c(2, 3))
y_pred <- random_uniform(c(2, 3))
loss <- loss_squared_hinge(y_true, y_pred)
```

See Also

- https://keras.io/api/losses/hinge_losses#squaredhinge-class

Other losses:
`Loss()`
`loss_binary_crossentropy()`
`loss_binary_focal_crossentropy()`
`loss_categorical_crossentropy()`
`loss_categorical_focal_crossentropy()`
`loss_categorical_hinge()`
`loss_cosine_similarity()`
`loss_dice()`
`loss_hinge()`
`loss_huber()`
`loss_kl_divergence()`
`loss_log_cosh()`
`loss_mean_absolute_error()`

loss_mean_absolute_percentage_error()
loss_mean_squared_error()
loss_mean_squared_logarithmic_error()
loss_poisson()
loss_sparse_categorical_crossentropy()
metric_binary_crossentropy()
metric_binary_focal_crossentropy()
metric_categorical_crossentropy()
metric_categorical_focal_crossentropy()
metric_categorical_hinge()
metric_hinge()
metric_huber()
metric_kl_divergence()
metric_log_cosh()
metric_mean_absolute_error()
metric_mean_absolute_percentage_error()
metric_mean_squared_error()
metric_mean_squared_logarithmic_error()
metric_poisson()
metric_sparse_categorical_crossentropy()
metric_squared_hinge()

Metric *Subclass the base* Metric *class*

Description

A Metric object encapsulates metric logic and state that can be used to track model performance during training. It is what is returned by the family of metric functions that start with prefix metric_*, as well as what is returned by custom metrics defined with Metric().

Usage

```
Metric(
  classname,
  initialize = NULL,
  update_state = NULL,
  result = NULL,
  ...,
  public = list(),
  private = list(),
  inherit = NULL,
  parent_env = parent.frame()
)
```

Arguments

`classname`	String, the name of the custom class. (Conventionally, CamelCase).
`initialize, update_state, result`	Recommended methods to implement. See description section.
`..., public`	Additional methods or public members of the custom class.
`private`	Named list of R objects (typically, functions) to include in instance private environments. `private` methods will have all the same symbols in scope as public methods (See section "Symbols in Scope"). Each instance will have it's own `private` environment. Any objects in `private` will be invisible from the Keras framework and the Python runtime.
`inherit`	What the custom class will subclass. By default, the base keras class.
`parent_env`	The R environment that all class methods will have as a grandparent.

Value

A function that returns `Metric` instances, similar to the builtin metric functions.

Examples

Usage with `compile()`::

```
model |> compile(
  optimizer = 'sgd',
  loss = 'mse',
  metrics = c(metric_SOME_METRIC(), metric_SOME_OTHER_METRIC())
)
```

Standalone usage::

```
m <- metric_SOME_METRIC()
for (e in seq(epochs)) {
  for (i in seq(train_steps)) {
    c(y_true, y_pred, sample_weight = NULL) %<-% ...
    m$update_state(y_true, y_pred, sample_weight)
  }
  cat('Final epoch result: ', as.numeric(m$result()), "\n")
  m$reset_state()
}
```

Full Examples

Usage with `compile()`::

```
model <- keras_model_sequential()
model |>
  layer_dense(64, activation = "relu") |>
  layer_dense(64, activation = "relu") |>
  layer_dense(10, activation = "softmax")
```

```
model |>
  compile(optimizer = optimizer_rmsprop(0.01),
          loss = loss_categorical_crossentropy(),
          metrics = metric_categorical_accuracy())

data <- random_uniform(c(1000, 32))
labels <- random_uniform(c(1000, 10))

model |> fit(data, labels, verbose = 0)
```

To be implemented by subclasses (custom metrics):

- `initialize()`: All state variables should be created in this method by calling `self$add_variable()` like: `self$var <- self$add_variable(...)`.
- `update_state()`: Updates all the state variables like: `self$var$assign(...)`.
- `result()`: Computes and returns a scalar value or a named list of scalar values for the metric from the state variables.

Example subclass implementation:

```
metric_binary_true_positives <- Metric(
  classname = "BinaryTruePositives",

  initialize = function(name = 'binary_true_positives', ...) {
    super$initialize(name = name, ...)
    self$true_positives <-
      self$add_weight(shape = shape(),
                      initializer = 'zeros',
                      name = 'true_positives')
  },

  update_state = function(y_true, y_pred, sample_weight = NULL) {
    y_true <- op_cast(y_true, "bool")
    y_pred <- op_cast(y_pred, "bool")

    values <- y_true & y_pred # `&` calls op_logical_and()
    values <- op_cast(values, self$dtype)
    if (!is.null(sample_weight)) {
      sample_weight <- op_cast(sample_weight, self$dtype)
      sample_weight <- op_broadcast_to(sample_weight, shape(values))
      values <- values * sample_weight # `*` calls op_multiply()
    }
    self$true_positives$assign(self$true_positives + op_sum(values))
  },

  result = function() {
    self$true_positives
  }
)
model <- keras_model_sequential(input_shape = 32) |> layer_dense(10)
model |> compile(loss = loss_binary_crossentropy(),
                 metrics = list(metric_binary_true_positives()))
model |> fit(data, labels, verbose = 0)
```

Methods defined by the base `Metric` class:

- `__call__(...)`

  Calling a metric instance self like `m(...)` is equivalent to calling:

  ```
  function(...) {
    m$update_state(...)
    m$result()
  }
  ```

- `initialize(dtype=NULL, name=NULL)`

  Initialize self.

  Args:

  – `name`: (Optional) string name of the metric instance.
  – `dtype`: (Optional) data type of the metric result.

- `add_variable(shape, initializer, dtype=NULL, aggregation = 'sum', name=NULL)`
- `add_weight(shape=shape(), initializer=NULL, dtype=NULL, name=NULL)`
- `get_config()`

  Return the serializable config of the metric.

- `reset_state()`

  Reset all of the metric state variables.

  This function is called between epochs/steps, when a metric is evaluated during training.

- `result()`

  Compute the current metric value.

  Returns: A scalar tensor, or a named list of scalar tensors.

- `stateless_result(metric_variables)`
- `stateless_reset_state()`
- `stateless_update_state(metric_variables, ...)`
- `update_state(...)`

  Accumulate statistics for the metric.

Readonly properties

- `dtype`
- `variables`

Symbols in scope

All R function custom methods (public and private) will have the following symbols in scope:

- `self`: The custom class instance.
- `super`: The custom class superclass.
- `private`: An R environment specific to the class instance. Any objects assigned here are invisible to the Keras framework.
- `__class__` and `as.symbol(classname)`: the custom class type object.

See Also

Other metrics:
`custom_metric()`
`metric_auc()`
`metric_binary_accuracy()`
`metric_binary_crossentropy()`
`metric_binary_focal_crossentropy()`
`metric_binary_iou()`
`metric_categorical_accuracy()`
`metric_categorical_crossentropy()`
`metric_categorical_focal_crossentropy()`
`metric_categorical_hinge()`
`metric_cosine_similarity()`
`metric_f1_score()`
`metric_false_negatives()`
`metric_false_positives()`
`metric_fbeta_score()`
`metric_hinge()`
`metric_huber()`
`metric_iou()`
`metric_kl_divergence()`
`metric_log_cosh()`
`metric_log_cosh_error()`
`metric_mean()`
`metric_mean_absolute_error()`
`metric_mean_absolute_percentage_error()`
`metric_mean_iou()`
`metric_mean_squared_error()`
`metric_mean_squared_logarithmic_error()`
`metric_mean_wrapper()`
`metric_one_hot_iou()`
`metric_one_hot_mean_iou()`
`metric_poisson()`
`metric_precision()`
`metric_precision_at_recall()`
`metric_r2_score()`
`metric_recall()`
`metric_recall_at_precision()`

metric_root_mean_squared_error()
metric_sensitivity_at_specificity()
metric_sparse_categorical_accuracy()
metric_sparse_categorical_crossentropy()
metric_sparse_top_k_categorical_accuracy()
metric_specificity_at_sensitivity()
metric_squared_hinge()
metric_sum()
metric_top_k_categorical_accuracy()
metric_true_negatives()
metric_true_positives()

---

metric_auc *Approximates the AUC (Area under the curve) of the ROC or PR curves.*

Description

The AUC (Area under the curve) of the ROC (Receiver operating characteristic; default) or PR (Precision Recall) curves are quality measures of binary classifiers. Unlike the accuracy, and like cross-entropy losses, ROC-AUC and PR-AUC evaluate all the operational points of a model.

This class approximates AUCs using a Riemann sum. During the metric accumulation phrase, predictions are accumulated within predefined buckets by value. The AUC is then computed by interpolating per-bucket averages. These buckets define the evaluated operational points.

This metric creates four local variables, `true_positives`, `true_negatives`, `false_positives` and `false_negatives` that are used to compute the AUC. To discretize the AUC curve, a linearly spaced set of thresholds is used to compute pairs of recall and precision values. The area under the ROC-curve is therefore computed using the height of the recall values by the false positive rate, while the area under the PR-curve is the computed using the height of the precision values by the recall.

This value is ultimately returned as `auc`, an idempotent operation that computes the area under a discretized curve of precision versus recall values (computed using the aforementioned variables). The `num_thresholds` variable controls the degree of discretization with larger numbers of thresholds more closely approximating the true AUC. The quality of the approximation may vary dramatically depending on `num_thresholds`. The `thresholds` parameter can be used to manually specify thresholds which split the predictions more evenly.

For a best approximation of the real AUC, `predictions` should be distributed approximately uniformly in the range `[0, 1]` (if `from_logits=FALSE`). The quality of the AUC approximation may be poor if this is not the case. Setting `summation_method` to 'minoring' or 'majoring' can help quantify the error in the approximation by providing lower or upper bound estimate of the AUC.

If `sample_weight` is `NULL`, weights default to 1. Use `sample_weight` of 0 to mask values.

Usage

```
metric_auc(
  ...,
  num_thresholds = 200L,
  curve = "ROC",
  summation_method = "interpolation",
  name = NULL,
  dtype = NULL,
  thresholds = NULL,
  multi_label = FALSE,
  num_labels = NULL,
  label_weights = NULL,
  from_logits = FALSE
)
```

Arguments

`...`	For forward/backward compatability.
`num_thresholds`	(Optional) The number of thresholds to use when discretizing the roc curve. Values must be > 1. Defaults to `200`.
`curve`	(Optional) Specifies the name of the curve to be computed, `'ROC'` (default) or `'PR'` for the Precision-Recall-curve.
`summation_method`	(Optional) Specifies the Riemann summation method used. 'interpolation' (default) applies mid-point summation scheme for `ROC`. For PR-AUC, interpolates (true/false) positives but not the ratio that is precision (see Davis & Goadrich 2006 for details); 'minoring' applies left summation for increasing intervals and right summation for decreasing intervals; 'majoring' does the opposite.
`name`	(Optional) string name of the metric instance.
`dtype`	(Optional) data type of the metric result.
`thresholds`	(Optional) A list of floating point values to use as the thresholds for discretizing the curve. If set, the `num_thresholds` parameter is ignored. Values should be in `[0, 1]`. Endpoint thresholds equal to {`-epsilon`, `1+epsilon`} for a small positive epsilon value will be automatically included with these to correctly handle predictions equal to exactly 0 or 1.
`multi_label`	boolean indicating whether multilabel data should be treated as such, wherein AUC is computed separately for each label and then averaged across labels, or (when `FALSE`) if the data should be flattened into a single label before AUC computation. In the latter case, when multilabel data is passed to AUC, each label-prediction pair is treated as an individual data point. Should be set to 'FALSE" for multi-class data.
`num_labels`	(Optional) The number of labels, used when `multi_label` is TRUE. If `num_labels` is not specified, then state variables get created on the first call to `update_state`.
`label_weights`	(Optional) list, array, or tensor of non-negative weights used to compute AUCs for multilabel data. When `multi_label` is TRUE, the weights are applied to the individual label AUCs when they are averaged to produce the multi-label AUC.

When it's FALSE, they are used to weight the individual label predictions in computing the confusion matrix on the flattened data. Note that this is unlike class_weights in that class_weights weights the example depending on the value of its label, whereas label_weights depends only on the index of that label before flattening; therefore label_weights should not be used for multi-class data.

from_logits	boolean indicating whether the predictions (y_pred in update_state) are probabilities or sigmoid logits. As a rule of thumb, when using a keras loss, the from_logits constructor argument of the loss should match the AUC from_logits constructor argument.

Value

a Metric instance is returned. The Metric instance can be passed directly to compile(metrics =), or used as a standalone object. See ?Metric for example usage.

Usage

Standalone usage:

```
m <- metric_auc(num_thresholds = 3)
m$update_state(c(0,   0,   1,   1),
               c(0, 0.5, 0.3, 0.9))
# threshold values are [0 - 1e-7, 0.5, 1 + 1e-7]
# tp = [2, 1, 0], fp = [2, 0, 0], fn = [0, 1, 2], tn = [0, 2, 2]
# tp_rate = recall = [1, 0.5, 0], fp_rate = [1, 0, 0]
# auc = ((((1 + 0.5) / 2) * (1 - 0)) + (((0.5 + 0) / 2) * (0 - 0)))
#     = 0.75
m$result()
```

```
## tf.Tensor(0.75, shape=(), dtype=float32)
```

```
m$reset_state()
m$update_state(c(0,   0,   1,   1),
               c(0, 0.5, 0.3, 0.9),
               sample_weight=c(1, 0, 0, 1))
m$result()
```

```
## tf.Tensor(1.0, shape=(), dtype=float32)
```

Usage with compile() API:

```
# Reports the AUC of a model outputting a probability.
model |> compile(
  optimizer = 'sgd',
  loss = loss_binary_crossentropy(),
  metrics = list(metric_auc())
)

# Reports the AUC of a model outputting a logit.
model |> compile(
  optimizer = 'sgd',
  loss = loss_binary_crossentropy(from_logits = TRUE),
  metrics = list(metric_auc(from_logits = TRUE))
)
```

See Also

- https://keras.io/api/metrics/classification_metrics#auc-class

Other confusion metrics:
metric_false_negatives()
metric_false_positives()
metric_precision()
metric_precision_at_recall()
metric_recall()
metric_recall_at_precision()
metric_sensitivity_at_specificity()
metric_specificity_at_sensitivity()
metric_true_negatives()
metric_true_positives()

Other metrics:
Metric()
custom_metric()
metric_binary_accuracy()
metric_binary_crossentropy()
metric_binary_focal_crossentropy()
metric_binary_iou()
metric_categorical_accuracy()
metric_categorical_crossentropy()
metric_categorical_focal_crossentropy()
metric_categorical_hinge()
metric_cosine_similarity()
metric_f1_score()
metric_false_negatives()
metric_false_positives()
metric_fbeta_score()
metric_hinge()
metric_huber()
metric_iou()
metric_kl_divergence()
metric_log_cosh()
metric_log_cosh_error()
metric_mean()

metric_mean_absolute_error()
metric_mean_absolute_percentage_error()
metric_mean_iou()
metric_mean_squared_error()
metric_mean_squared_logarithmic_error()
metric_mean_wrapper()
metric_one_hot_iou()
metric_one_hot_mean_iou()
metric_poisson()
metric_precision()
metric_precision_at_recall()
metric_r2_score()
metric_recall()
metric_recall_at_precision()
metric_root_mean_squared_error()
metric_sensitivity_at_specificity()
metric_sparse_categorical_accuracy()
metric_sparse_categorical_crossentropy()
metric_sparse_top_k_categorical_accuracy()
metric_specificity_at_sensitivity()
metric_squared_hinge()
metric_sum()
metric_top_k_categorical_accuracy()
metric_true_negatives()
metric_true_positives()

---

metric_binary_accuracy

Calculates how often predictions match binary labels.

Description

This metric creates two local variables, `total` and `count` that are used to compute the frequency with which `y_pred` matches `y_true`. This frequency is ultimately returned as `binary accuracy`: an idempotent operation that simply divides `total` by `count`.

If `sample_weight` is `NULL`, weights default to 1. Use `sample_weight` of 0 to mask values.

Usage

```
metric_binary_accuracy(
  y_true,
  y_pred,
  threshold = 0.5,
  ...,
  name = "binary_accuracy",
  dtype = NULL
)
```

Arguments

`y_true`	Tensor of true targets.
`y_pred`	Tensor of predicted targets.
`threshold`	(Optional) Float representing the threshold for deciding whether prediction values are 1 or 0.
`...`	For forward/backward compatability.
`name`	(Optional) string name of the metric instance.
`dtype`	(Optional) data type of the metric result.

Value

If `y_true` and `y_pred` are missing, a `Metric` instance is returned. The `Metric` instance that can be passed directly to `compile(metrics = )`, or used as a standalone object. See `?Metric` for example usage. If `y_true` and `y_pred` are provided, then a tensor with the computed value is returned.

Usage

Standalone usage:

```
m <- metric_binary_accuracy()
m$update_state(rbind(1, 1, 0, 0), rbind(0.98, 1, 0, 0.6))
m$result()
```

```
## tf.Tensor(0.75, shape=(), dtype=float32)
```

```
# 0.75
```

```
m$reset_state()
m$update_state(rbind(1, 1, 0, 0), rbind(0.98, 1, 0, 0.6),
               sample_weight = c(1, 0, 0, 1))
m$result()
```

```
## tf.Tensor(0.5, shape=(), dtype=float32)
```

```
# 0.5
```

Usage with `compile()` API:

```
model %>% compile(optimizer='sgd',
                  loss='binary_crossentropy',
                  metrics=list(metric_binary_accuracy()))
```

See Also

- https://keras.io/api/metrics/accuracy_metrics#binaryaccuracy-class

Other accuracy metrics:
metric_categorical_accuracy()
metric_sparse_categorical_accuracy()
metric_sparse_top_k_categorical_accuracy()
metric_top_k_categorical_accuracy()

Other metrics:
Metric()
custom_metric()
metric_auc()
metric_binary_crossentropy()
metric_binary_focal_crossentropy()
metric_binary_iou()
metric_categorical_accuracy()
metric_categorical_crossentropy()
metric_categorical_focal_crossentropy()
metric_categorical_hinge()
metric_cosine_similarity()
metric_f1_score()
metric_false_negatives()
metric_false_positives()
metric_fbeta_score()
metric_hinge()
metric_huber()
metric_iou()
metric_kl_divergence()
metric_log_cosh()
metric_log_cosh_error()
metric_mean()
metric_mean_absolute_error()
metric_mean_absolute_percentage_error()
metric_mean_iou()
metric_mean_squared_error()
metric_mean_squared_logarithmic_error()
metric_mean_wrapper()
metric_one_hot_iou()
metric_one_hot_mean_iou()
metric_poisson()
metric_precision()
metric_precision_at_recall()
metric_r2_score()
metric_recall()
metric_recall_at_precision()
metric_root_mean_squared_error()
metric_sensitivity_at_specificity()

metric_sparse_categorical_accuracy()
metric_sparse_categorical_crossentropy()
metric_sparse_top_k_categorical_accuracy()
metric_specificity_at_sensitivity()
metric_squared_hinge()
metric_sum()
metric_top_k_categorical_accuracy()
metric_true_negatives()
metric_true_positives()

---

metric_binary_crossentropy
Computes the crossentropy metric between the labels and predictions.

Description

This is the crossentropy metric class to be used when there are only two label classes (0 and 1).

Usage

```
metric_binary_crossentropy(
  y_true,
  y_pred,
  from_logits = FALSE,
  label_smoothing = 0,
  axis = -1L,
  ...,
  name = "binary_crossentropy",
  dtype = NULL
)
```

Arguments

y_true	Ground truth values. shape = [batch_size, d0, .. dN].
y_pred	The predicted values. shape = [batch_size, d0, .. dN].
from_logits	(Optional) Whether output is expected to be a logits tensor. By default, we consider that output encodes a probability distribution.
label_smoothing	(Optional) Float in [0, 1]. When > 0, label values are smoothed, meaning the confidence on label values are relaxed. e.g. label_smoothing=0.2 means that we will use a value of 0.1 for label "0" and 0.9 for label "1".
axis	The axis along which the mean is computed. Defaults to -1.
...	For forward/backward compatability.
name	(Optional) string name of the metric instance.
dtype	(Optional) data type of the metric result.

Value

If y_true and y_pred are missing, a Metric instance is returned. The Metric instance that can be passed directly to compile(metrics =), or used as a standalone object. See ?Metric for example usage. If y_true and y_pred are provided, then a tensor with the computed value is returned.

Examples

Standalone usage:

```
m <- metric_binary_crossentropy()
m$update_state(rbind(c(0, 1), c(0, 0)), rbind(c(0.6, 0.4), c(0.4, 0.6)))
m$result()

## tf.Tensor(0.8149245, shape=(), dtype=float32)


m$reset_state()
m$update_state(rbind(c(0, 1), c(0, 0)), rbind(c(0.6, 0.4), c(0.4, 0.6)),
               sample_weight=c(1, 0))
m$result()

## tf.Tensor(0.91629076, shape=(), dtype=float32)
```

Usage with compile() API:

```
model %>% compile(
    optimizer = 'sgd',
    loss = 'mse',
    metrics = list(metric_binary_crossentropy()))
```

See Also

- https://keras.io/api/metrics/probabilistic_metrics#binarycrossentropy-class

Other losses:
Loss()
loss_binary_crossentropy()
loss_binary_focal_crossentropy()
loss_categorical_crossentropy()
loss_categorical_focal_crossentropy()
loss_categorical_hinge()
loss_cosine_similarity()
loss_dice()
loss_hinge()
loss_huber()
loss_kl_divergence()
loss_log_cosh()

loss_mean_absolute_error()
loss_mean_absolute_percentage_error()
loss_mean_squared_error()
loss_mean_squared_logarithmic_error()
loss_poisson()
loss_sparse_categorical_crossentropy()
loss_squared_hinge()
metric_binary_focal_crossentropy()
metric_categorical_crossentropy()
metric_categorical_focal_crossentropy()
metric_categorical_hinge()
metric_hinge()
metric_huber()
metric_kl_divergence()
metric_log_cosh()
metric_mean_absolute_error()
metric_mean_absolute_percentage_error()
metric_mean_squared_error()
metric_mean_squared_logarithmic_error()
metric_poisson()
metric_sparse_categorical_crossentropy()
metric_squared_hinge()

Other metrics:
Metric()
custom_metric()
metric_auc()
metric_binary_accuracy()
metric_binary_focal_crossentropy()
metric_binary_iou()
metric_categorical_accuracy()
metric_categorical_crossentropy()
metric_categorical_focal_crossentropy()
metric_categorical_hinge()
metric_cosine_similarity()
metric_f1_score()
metric_false_negatives()
metric_false_positives()
metric_fbeta_score()
metric_hinge()
metric_huber()
metric_iou()
metric_kl_divergence()
metric_log_cosh()
metric_log_cosh_error()
metric_mean()
metric_mean_absolute_error()
metric_mean_absolute_percentage_error()

metric_mean_iou()
metric_mean_squared_error()
metric_mean_squared_logarithmic_error()
metric_mean_wrapper()
metric_one_hot_iou()
metric_one_hot_mean_iou()
metric_poisson()
metric_precision()
metric_precision_at_recall()
metric_r2_score()
metric_recall()
metric_recall_at_precision()
metric_root_mean_squared_error()
metric_sensitivity_at_specificity()
metric_sparse_categorical_accuracy()
metric_sparse_categorical_crossentropy()
metric_sparse_top_k_categorical_accuracy()
metric_specificity_at_sensitivity()
metric_squared_hinge()
metric_sum()
metric_top_k_categorical_accuracy()
metric_true_negatives()
metric_true_positives()

Other probabilistic metrics:
metric_categorical_crossentropy()
metric_kl_divergence()
metric_poisson()
metric_sparse_categorical_crossentropy()

`metric_binary_focal_crossentropy`
Computes the binary focal crossentropy loss.

Description

According to Lin et al., 2018, it helps to apply a focal factor to down-weight easy examples and focus more on hard examples. By default, the focal tensor is computed as follows:

`focal_factor = (1 - output)^gamma` for class 1 `focal_factor = output^gamma` for class 0 where `gamma` is a focusing parameter. When `gamma` = 0, there is no focal effect on the binary crossentropy loss.

If `apply_class_balancing == TRUE`, this function also takes into account a weight balancing factor for the binary classes 0 and 1 as follows:

`weight = alpha` for class 1 (`target == 1`) `weight = 1 - alpha` for class 0 where `alpha` is a float in the range of `[0, 1]`.

Usage

```
metric_binary_focal_crossentropy(
  y_true,
  y_pred,
  apply_class_balancing = FALSE,
  alpha = 0.25,
  gamma = 2,
  from_logits = FALSE,
  label_smoothing = 0,
  axis = -1L
)
```

Arguments

`y_true`	Ground truth values, of shape `(batch_size, d0, .. dN)`.
`y_pred`	The predicted values, of shape `(batch_size, d0, .. dN)`.
`apply_class_balancing`	A bool, whether to apply weight balancing on the binary classes 0 and 1.
`alpha`	A weight balancing factor for class 1, default is `0.25` as mentioned in the reference. The weight for class 0 is `1.0 - alpha`.
`gamma`	A focusing parameter, default is `2.0` as mentioned in the reference.
`from_logits`	Whether `y_pred` is expected to be a logits tensor. By default, we assume that `y_pred` encodes a probability distribution.
`label_smoothing`	Float in `[0, 1]`. If `> 0` then smooth the labels by squeezing them towards 0.5, that is, using `1. - 0.5 * label_smoothing` for the target class and `0.5 * label_smoothing` for the non-target class.
`axis`	The axis along which the mean is computed. Defaults to `-1`.

Value

Binary focal crossentropy loss value with shape = `[batch_size, d0, .. dN-1]`.

Examples

```
y_true <- rbind(c(0, 1), c(0, 0))
y_pred <- rbind(c(0.6, 0.4), c(0.4, 0.6))
loss <- loss_binary_focal_crossentropy(y_true, y_pred, gamma=2)
loss


## tf.Tensor([0.32986466 0.20579838], shape=(2), dtype=float64)
```

See Also

Other losses:
Loss()
loss_binary_crossentropy()
loss_binary_focal_crossentropy()
loss_categorical_crossentropy()
loss_categorical_focal_crossentropy()
loss_categorical_hinge()
loss_cosine_similarity()
loss_dice()
loss_hinge()
loss_huber()
loss_kl_divergence()
loss_log_cosh()
loss_mean_absolute_error()
loss_mean_absolute_percentage_error()
loss_mean_squared_error()
loss_mean_squared_logarithmic_error()
loss_poisson()
loss_sparse_categorical_crossentropy()
loss_squared_hinge()
metric_binary_crossentropy()
metric_categorical_crossentropy()
metric_categorical_focal_crossentropy()
metric_categorical_hinge()
metric_hinge()
metric_huber()
metric_kl_divergence()
metric_log_cosh()
metric_mean_absolute_error()
metric_mean_absolute_percentage_error()
metric_mean_squared_error()
metric_mean_squared_logarithmic_error()
metric_poisson()
metric_sparse_categorical_crossentropy()
metric_squared_hinge()

Other metrics:
Metric()
custom_metric()
metric_auc()
metric_binary_accuracy()
metric_binary_crossentropy()
metric_binary_iou()
metric_categorical_accuracy()
metric_categorical_crossentropy()
metric_categorical_focal_crossentropy()
metric_categorical_hinge()

metric_cosine_similarity()
metric_f1_score()
metric_false_negatives()
metric_false_positives()
metric_fbeta_score()
metric_hinge()
metric_huber()
metric_iou()
metric_kl_divergence()
metric_log_cosh()
metric_log_cosh_error()
metric_mean()
metric_mean_absolute_error()
metric_mean_absolute_percentage_error()
metric_mean_iou()
metric_mean_squared_error()
metric_mean_squared_logarithmic_error()
metric_mean_wrapper()
metric_one_hot_iou()
metric_one_hot_mean_iou()
metric_poisson()
metric_precision()
metric_precision_at_recall()
metric_r2_score()
metric_recall()
metric_recall_at_precision()
metric_root_mean_squared_error()
metric_sensitivity_at_specificity()
metric_sparse_categorical_accuracy()
metric_sparse_categorical_crossentropy()
metric_sparse_top_k_categorical_accuracy()
metric_specificity_at_sensitivity()
metric_squared_hinge()
metric_sum()
metric_top_k_categorical_accuracy()
metric_true_negatives()
metric_true_positives()

---

metric_binary_iou *Computes the Intersection-Over-Union metric for class 0 and/or 1.*

Description

Formula:

```
iou <- true_positives / (true_positives + false_positives + false_negatives)
```

Intersection-Over-Union is a common evaluation metric for semantic image segmentation.

To compute IoUs, the predictions are accumulated in a confusion matrix, weighted by `sample_weight` and the metric is then calculated from it.

If `sample_weight` is `NULL`, weights default to 1. Use `sample_weight` of 0 to mask values.

This class can be used to compute IoUs for a binary classification task where the predictions are provided as logits. First a `threshold` is applied to the predicted values such that those that are below the `threshold` are converted to class 0 and those that are above the `threshold` are converted to class 1.

IoUs for classes 0 and 1 are then computed, the mean of IoUs for the classes that are specified by `target_class_ids` is returned.

Usage

```
metric_binary_iou(
  ...,
  target_class_ids = list(0L, 1L),
  threshold = 0.5,
  name = NULL,
  dtype = NULL
)
```

Arguments

`...`	For forward/backward compatability.
`target_class_ids`	A list or list of target class ids for which the metric is returned. Options are `0`, `1`, or `c(0, 1)`. With `0` (or `1`), the IoU metric for class 0 (or class 1, respectively) is returned. With `c(0, 1)`, the mean of IoUs for the two classes is returned.
`threshold`	A threshold that applies to the prediction logits to convert them to either predicted class 0 if the logit is below `threshold` or predicted class 1 if the logit is above `threshold`.
`name`	(Optional) string name of the metric instance.
`dtype`	(Optional) data type of the metric result.

Value

a `Metric` instance is returned. The `Metric` instance can be passed directly to `compile(metrics = )`, or used as a standalone object. See `?Metric` for example usage.

Note

with `threshold=0`, this metric has the same behavior as `IoU`.

Examples

Standalone usage:

```
m <- metric_binary_iou(target_class_ids=c(0L, 1L), threshold = 0.3)
m$update_state(c(0, 1, 0, 1), c(0.1, 0.2, 0.4, 0.7))
m$result()

## tf.Tensor(0.33333334, shape=(), dtype=float32)
```

```
m$reset_state()
m$update_state(c(0, 1, 0, 1), c(0.1, 0.2, 0.4, 0.7),
               sample_weight = c(0.2, 0.3, 0.4, 0.1))
m$result()

## tf.Tensor(0.17361109, shape=(), dtype=float32)
```

Usage with `compile()` API:

```
model %>% compile(
    optimizer = 'sgd',
    loss = 'mse',
    metrics = list(metric_binary_iou(
        target_class_ids = 0L,
        threshold = 0.5
    ))
)
```

See Also

Other iou metrics:
`metric_iou()`
`metric_mean_iou()`
`metric_one_hot_iou()`
`metric_one_hot_mean_iou()`

Other metrics:
`Metric()`
`custom_metric()`
`metric_auc()`
`metric_binary_accuracy()`
`metric_binary_crossentropy()`
`metric_binary_focal_crossentropy()`
`metric_categorical_accuracy()`
`metric_categorical_crossentropy()`
`metric_categorical_focal_crossentropy()`

metric_categorical_hinge()
metric_cosine_similarity()
metric_f1_score()
metric_false_negatives()
metric_false_positives()
metric_fbeta_score()
metric_hinge()
metric_huber()
metric_iou()
metric_kl_divergence()
metric_log_cosh()
metric_log_cosh_error()
metric_mean()
metric_mean_absolute_error()
metric_mean_absolute_percentage_error()
metric_mean_iou()
metric_mean_squared_error()
metric_mean_squared_logarithmic_error()
metric_mean_wrapper()
metric_one_hot_iou()
metric_one_hot_mean_iou()
metric_poisson()
metric_precision()
metric_precision_at_recall()
metric_r2_score()
metric_recall()
metric_recall_at_precision()
metric_root_mean_squared_error()
metric_sensitivity_at_specificity()
metric_sparse_categorical_accuracy()
metric_sparse_categorical_crossentropy()
metric_sparse_top_k_categorical_accuracy()
metric_specificity_at_sensitivity()
metric_squared_hinge()
metric_sum()
metric_top_k_categorical_accuracy()
metric_true_negatives()
metric_true_positives()

---

metric_categorical_accuracy
Calculates how often predictions match one-hot labels.

Description

You can provide logits of classes as y_pred, since argmax of logits and probabilities are same.

This metric creates two local variables, `total` and `count` that are used to compute the frequency with which `y_pred` matches `y_true`. This frequency is ultimately returned as `categorical accuracy`: an idempotent operation that simply divides `total` by `count`.

`y_pred` and `y_true` should be passed in as vectors of probabilities, rather than as labels. If necessary, use `op_one_hot` to expand `y_true` as a vector.

If `sample_weight` is `NULL`, weights default to 1. Use `sample_weight` of 0 to mask values.

Usage

```
metric_categorical_accuracy(
  y_true,
  y_pred,
  ...,
  name = "categorical_accuracy",
  dtype = NULL
)
```

Arguments

`y_true`	Tensor of true targets.
`y_pred`	Tensor of predicted targets.
`...`	For forward/backward compatability.
`name`	(Optional) string name of the metric instance.
`dtype`	(Optional) data type of the metric result.

Value

If `y_true` and `y_pred` are missing, a `Metric` instance is returned. The `Metric` instance that can be passed directly to `compile(metrics = )`, or used as a standalone object. See `?Metric` for example usage. If `y_true` and `y_pred` are provided, then a tensor with the computed value is returned.

Usage

Standalone usage:

```
m <- metric_categorical_accuracy()
m$update_state(rbind(c(0, 0, 1), c(0, 1, 0)), rbind(c(0.1, 0.9, 0.8),
                c(0.05, 0.95, 0)))
m$result()

## tf.Tensor(0.5, shape=(), dtype=float32)


m$reset_state()
m$update_state(rbind(c(0, 0, 1), c(0, 1, 0)), rbind(c(0.1, 0.9, 0.8),
               c(0.05, 0.95, 0)),
               sample_weight = c(0.7, 0.3))
m$result()
```

```
## tf.Tensor(0.3, shape=(), dtype=float32)


# 0.3
```

Usage with `compile()` API:

```
model %>% compile(optimizer = 'sgd',
                  loss = 'categorical_crossentropy',
                  metrics = list(metric_categorical_accuracy()))
```

See Also

- `https://keras.io/api/metrics/accuracy_metrics#categoricalaccuracy-class`

Other accuracy metrics:
`metric_binary_accuracy()`
`metric_sparse_categorical_accuracy()`
`metric_sparse_top_k_categorical_accuracy()`
`metric_top_k_categorical_accuracy()`

Other metrics:
`Metric()`
`custom_metric()`
`metric_auc()`
`metric_binary_accuracy()`
`metric_binary_crossentropy()`
`metric_binary_focal_crossentropy()`
`metric_binary_iou()`
`metric_categorical_crossentropy()`
`metric_categorical_focal_crossentropy()`
`metric_categorical_hinge()`
`metric_cosine_similarity()`
`metric_f1_score()`
`metric_false_negatives()`
`metric_false_positives()`
`metric_fbeta_score()`
`metric_hinge()`
`metric_huber()`
`metric_iou()`
`metric_kl_divergence()`
`metric_log_cosh()`
`metric_log_cosh_error()`
`metric_mean()`
`metric_mean_absolute_error()`
`metric_mean_absolute_percentage_error()`
`metric_mean_iou()`
`metric_mean_squared_error()`
`metric_mean_squared_logarithmic_error()`

metric_mean_wrapper()
metric_one_hot_iou()
metric_one_hot_mean_iou()
metric_poisson()
metric_precision()
metric_precision_at_recall()
metric_r2_score()
metric_recall()
metric_recall_at_precision()
metric_root_mean_squared_error()
metric_sensitivity_at_specificity()
metric_sparse_categorical_accuracy()
metric_sparse_categorical_crossentropy()
metric_sparse_top_k_categorical_accuracy()
metric_specificity_at_sensitivity()
metric_squared_hinge()
metric_sum()
metric_top_k_categorical_accuracy()
metric_true_negatives()
metric_true_positives()

metric_categorical_crossentropy

Computes the crossentropy metric between the labels and predictions.

Description

This is the crossentropy metric class to be used when there are multiple label classes (2 or more). It assumes that labels are one-hot encoded, e.g., when labels values are `c(2, 0, 1)`, then `y_true` is `rbind(c([0, 0, 1), c(1, 0, 0), c(0, 1, 0))`.

Usage

```
metric_categorical_crossentropy(
  y_true,
  y_pred,
  from_logits = FALSE,
  label_smoothing = 0,
  axis = -1L,
  ...,
  name = "categorical_crossentropy",
  dtype = NULL
)
```

Arguments

y_true	Tensor of one-hot true targets.
y_pred	Tensor of predicted targets.
from_logits	(Optional) Whether output is expected to be a logits tensor. By default, we consider that output encodes a probability distribution.
label_smoothing	(Optional) Float in [0, 1]. When > 0, label values are smoothed, meaning the confidence on label values are relaxed. e.g. label_smoothing=0.2 means that we will use a value of 0.1 for label "0" and 0.9 for label "1".
axis	(Optional) Defaults to -1. The dimension along which entropy is computed.
...	For forward/backward compatability.
name	(Optional) string name of the metric instance.
dtype	(Optional) data type of the metric result.

Value

If y_true and y_pred are missing, a Metric instance is returned. The Metric instance that can be passed directly to compile(metrics =), or used as a standalone object. See ?Metric for example usage. If y_true and y_pred are provided, then a tensor with the computed value is returned.

Examples

Standalone usage:

```
# EPSILON = 1e-7, y = y_true, y` = y_pred
# y` = clip_op_clip_by_value(output, EPSILON, 1. - EPSILON)
# y` = rbind(c(0.05, 0.95, EPSILON), c(0.1, 0.8, 0.1))
# xent = -sum(y * log(y'), axis = -1)
#      = -((log 0.95), (log 0.1))
#      = [0.051, 2.302]
# Reduced xent = (0.051 + 2.302) / 2
m <- metric_categorical_crossentropy()
m$update_state(rbind(c(0, 1, 0), c(0, 0, 1)),
               rbind(c(0.05, 0.95, 0), c(0.1, 0.8, 0.1)))
m$result()

## tf.Tensor(1.1769392, shape=(), dtype=float32)


# 1.1769392

m$reset_state()
m$update_state(rbind(c(0, 1, 0), c(0, 0, 1)),
               rbind(c(0.05, 0.95, 0), c(0.1, 0.8, 0.1)),
               sample_weight = c(0.3, 0.7))
m$result()
```

```
## tf.Tensor(1.6271976, shape=(), dtype=float32)
```

Usage with `compile()` API:

```
model %>% compile(
  optimizer = 'sgd',
  loss = 'mse',
  metrics = list(metric_categorical_crossentropy()))
```

See Also

- https://keras.io/api/metrics/probabilistic_metrics#categoricalcrossentropy-class

Other losses:
Loss()
loss_binary_crossentropy()
loss_binary_focal_crossentropy()
loss_categorical_crossentropy()
loss_categorical_focal_crossentropy()
loss_categorical_hinge()
loss_cosine_similarity()
loss_dice()
loss_hinge()
loss_huber()
loss_kl_divergence()
loss_log_cosh()
loss_mean_absolute_error()
loss_mean_absolute_percentage_error()
loss_mean_squared_error()
loss_mean_squared_logarithmic_error()
loss_poisson()
loss_sparse_categorical_crossentropy()
loss_squared_hinge()
metric_binary_crossentropy()
metric_binary_focal_crossentropy()
metric_categorical_focal_crossentropy()
metric_categorical_hinge()
metric_hinge()
metric_huber()
metric_kl_divergence()
metric_log_cosh()
metric_mean_absolute_error()
metric_mean_absolute_percentage_error()
metric_mean_squared_error()
metric_mean_squared_logarithmic_error()
metric_poisson()
metric_sparse_categorical_crossentropy()
metric_squared_hinge()

Other metrics:
Metric()
custom_metric()
metric_auc()
metric_binary_accuracy()
metric_binary_crossentropy()
metric_binary_focal_crossentropy()
metric_binary_iou()
metric_categorical_accuracy()
metric_categorical_focal_crossentropy()
metric_categorical_hinge()
metric_cosine_similarity()
metric_f1_score()
metric_false_negatives()
metric_false_positives()
metric_fbeta_score()
metric_hinge()
metric_huber()
metric_iou()
metric_kl_divergence()
metric_log_cosh()
metric_log_cosh_error()
metric_mean()
metric_mean_absolute_error()
metric_mean_absolute_percentage_error()
metric_mean_iou()
metric_mean_squared_error()
metric_mean_squared_logarithmic_error()
metric_mean_wrapper()
metric_one_hot_iou()
metric_one_hot_mean_iou()
metric_poisson()
metric_precision()
metric_precision_at_recall()
metric_r2_score()
metric_recall()
metric_recall_at_precision()
metric_root_mean_squared_error()
metric_sensitivity_at_specificity()
metric_sparse_categorical_accuracy()
metric_sparse_categorical_crossentropy()
metric_sparse_top_k_categorical_accuracy()
metric_specificity_at_sensitivity()
metric_squared_hinge()
metric_sum()
metric_top_k_categorical_accuracy()
metric_true_negatives()
metric_true_positives()

Other probabilistic metrics:
`metric_binary_crossentropy()`
`metric_kl_divergence()`
`metric_poisson()`
`metric_sparse_categorical_crossentropy()`

`metric_categorical_focal_crossentropy`
Computes the categorical focal crossentropy loss.

Description

Computes the categorical focal crossentropy loss.

Usage

```
metric_categorical_focal_crossentropy(
  y_true,
  y_pred,
  alpha = 0.25,
  gamma = 2,
  from_logits = FALSE,
  label_smoothing = 0,
  axis = -1L
)
```

Arguments

`y_true`	Tensor of one-hot true targets.
`y_pred`	Tensor of predicted targets.
`alpha`	A weight balancing factor for all classes, default is `0.25` as mentioned in the reference. It can be a list of floats or a scalar. In the multi-class case, alpha may be set by inverse class frequency by using `compute_class_weight` from `sklearn.utils`.
`gamma`	A focusing parameter, default is `2.0` as mentioned in the reference. It helps to gradually reduce the importance given to simple examples in a smooth manner. When `gamma` = 0, there is no focal effect on the categorical crossentropy.
`from_logits`	Whether `y_pred` is expected to be a logits tensor. By default, we assume that `y_pred` encodes a probability distribution.
`label_smoothing`	Float in `[0, 1]`. If > `0` then smooth the labels. For example, if `0.1`, use `0.1` / `num_classes` for non-target labels and `0.9` + `0.1` / `num_classes` for target labels.
`axis`	Defaults to `-1`. The dimension along which the entropy is computed.

Value

Categorical focal crossentropy loss value.

Examples

```
y_true <- rbind(c(0, 1, 0), c(0, 0, 1))
y_pred <- rbind(c(0.05, 0.9, 0.05), c(0.1, 0.85, 0.05))
loss <- loss_categorical_focal_crossentropy(y_true, y_pred)
loss

## tf.Tensor([2.63401289e-04 6.75912094e-01], shape=(2), dtype=float64)
```

See Also

Other losses:
Loss()
loss_binary_crossentropy()
loss_binary_focal_crossentropy()
loss_categorical_crossentropy()
loss_categorical_focal_crossentropy()
loss_categorical_hinge()
loss_cosine_similarity()
loss_dice()
loss_hinge()
loss_huber()
loss_kl_divergence()
loss_log_cosh()
loss_mean_absolute_error()
loss_mean_absolute_percentage_error()
loss_mean_squared_error()
loss_mean_squared_logarithmic_error()
loss_poisson()
loss_sparse_categorical_crossentropy()
loss_squared_hinge()
metric_binary_crossentropy()
metric_binary_focal_crossentropy()
metric_categorical_crossentropy()
metric_categorical_hinge()
metric_hinge()
metric_huber()
metric_kl_divergence()
metric_log_cosh()
metric_mean_absolute_error()
metric_mean_absolute_percentage_error()
metric_mean_squared_error()
metric_mean_squared_logarithmic_error()
metric_poisson()

metric_sparse_categorical_crossentropy()
metric_squared_hinge()

Other metrics:
Metric()
custom_metric()
metric_auc()
metric_binary_accuracy()
metric_binary_crossentropy()
metric_binary_focal_crossentropy()
metric_binary_iou()
metric_categorical_accuracy()
metric_categorical_crossentropy()
metric_categorical_hinge()
metric_cosine_similarity()
metric_f1_score()
metric_false_negatives()
metric_false_positives()
metric_fbeta_score()
metric_hinge()
metric_huber()
metric_iou()
metric_kl_divergence()
metric_log_cosh()
metric_log_cosh_error()
metric_mean()
metric_mean_absolute_error()
metric_mean_absolute_percentage_error()
metric_mean_iou()
metric_mean_squared_error()
metric_mean_squared_logarithmic_error()
metric_mean_wrapper()
metric_one_hot_iou()
metric_one_hot_mean_iou()
metric_poisson()
metric_precision()
metric_precision_at_recall()
metric_r2_score()
metric_recall()
metric_recall_at_precision()
metric_root_mean_squared_error()
metric_sensitivity_at_specificity()
metric_sparse_categorical_accuracy()
metric_sparse_categorical_crossentropy()
metric_sparse_top_k_categorical_accuracy()
metric_specificity_at_sensitivity()
metric_squared_hinge()
metric_sum()

metric_top_k_categorical_accuracy()
metric_true_negatives()
metric_true_positives()

---

metric_categorical_hinge

Computes the categorical hinge metric between y_true *and* y_pred*.*

Description

Formula:

```
loss <- maximum(neg - pos + 1, 0)
```

where neg=maximum((1-y_true)*y_pred) and pos=sum(y_true*y_pred)

Usage

```
metric_categorical_hinge(
  y_true,
  y_pred,
  ...,
  name = "categorical_hinge",
  dtype = NULL
)
```

Arguments

y_true	The ground truth values. y_true values are expected to be either {-1, +1} or {0, 1} (i.e. a one-hot-encoded tensor) with shape = [batch_size, d0, .. dN].
y_pred	The predicted values with shape = [batch_size, d0, .. dN].
...	For forward/backward compatability.
name	(Optional) string name of the metric instance.
dtype	(Optional) data type of the metric result.

Value

Categorical hinge loss values with shape = [batch_size, d0, .. dN-1].

Usage

Standalone usage:

```
m <- metric_categorical_hinge()
m$update_state(rbind(c(0, 1), c(0, 0)), rbind(c(0.6, 0.4), c(0.4, 0.6)))
m$result()

## tf.Tensor(1.4000001, shape=(), dtype=float32)


m$reset_state()
m$update_state(rbind(c(0, 1), c(0, 0)), rbind(c(0.6, 0.4), c(0.4, 0.6)),
               sample_weight = c(1, 0))
m$result()

## tf.Tensor(1.2, shape=(), dtype=float32)
```

See Also

- https://keras.io/api/metrics/hinge_metrics#categoricalhinge-class

Other losses:
Loss()
loss_binary_crossentropy()
loss_binary_focal_crossentropy()
loss_categorical_crossentropy()
loss_categorical_focal_crossentropy()
loss_categorical_hinge()
loss_cosine_similarity()
loss_dice()
loss_hinge()
loss_huber()
loss_kl_divergence()
loss_log_cosh()
loss_mean_absolute_error()
loss_mean_absolute_percentage_error()
loss_mean_squared_error()
loss_mean_squared_logarithmic_error()
loss_poisson()
loss_sparse_categorical_crossentropy()
loss_squared_hinge()
metric_binary_crossentropy()
metric_binary_focal_crossentropy()
metric_categorical_crossentropy()
metric_categorical_focal_crossentropy()
metric_hinge()
metric_huber()
metric_kl_divergence()

metric_log_cosh()
metric_mean_absolute_error()
metric_mean_absolute_percentage_error()
metric_mean_squared_error()
metric_mean_squared_logarithmic_error()
metric_poisson()
metric_sparse_categorical_crossentropy()
metric_squared_hinge()

Other metrics:
Metric()
custom_metric()
metric_auc()
metric_binary_accuracy()
metric_binary_crossentropy()
metric_binary_focal_crossentropy()
metric_binary_iou()
metric_categorical_accuracy()
metric_categorical_crossentropy()
metric_categorical_focal_crossentropy()
metric_cosine_similarity()
metric_f1_score()
metric_false_negatives()
metric_false_positives()
metric_fbeta_score()
metric_hinge()
metric_huber()
metric_iou()
metric_kl_divergence()
metric_log_cosh()
metric_log_cosh_error()
metric_mean()
metric_mean_absolute_error()
metric_mean_absolute_percentage_error()
metric_mean_iou()
metric_mean_squared_error()
metric_mean_squared_logarithmic_error()
metric_mean_wrapper()
metric_one_hot_iou()
metric_one_hot_mean_iou()
metric_poisson()
metric_precision()
metric_precision_at_recall()
metric_r2_score()
metric_recall()
metric_recall_at_precision()
metric_root_mean_squared_error()
metric_sensitivity_at_specificity()

metric_sparse_categorical_accuracy()
metric_sparse_categorical_crossentropy()
metric_sparse_top_k_categorical_accuracy()
metric_specificity_at_sensitivity()
metric_squared_hinge()
metric_sum()
metric_top_k_categorical_accuracy()
metric_true_negatives()
metric_true_positives()

Other hinge metrics:
metric_hinge()
metric_squared_hinge()

---

metric_cosine_similarity

Computes the cosine similarity between the labels and predictions.

Description

Formula:

```
loss <- sum(l2_norm(y_true) * l2_norm(y_pred))
```

See: Cosine Similarity. This metric keeps the average cosine similarity between `predictions` and `labels` over a stream of data.

Usage

```
metric_cosine_similarity(
  ...,
  name = "cosine_similarity",
  dtype = NULL,
  axis = -1L
)
```

Arguments

`...`	For forward/backward compatability.
`name`	(Optional) string name of the metric instance.
`dtype`	(Optional) data type of the metric result.
`axis`	(Optional) Defaults to `-1`. The dimension along which the cosine similarity is computed.

Value

a Metric instance is returned. The Metric instance can be passed directly to compile(metrics =), or used as a standalone object. See ?Metric for example usage.

Examples

Standalone usage:

```
m <- metric_cosine_similarity(axis=2)
m$update_state(rbind(c(0., 1.), c(1., 1.)), rbind(c(1., 0.), c(1., 1.)))
m$result()

## tf.Tensor(0.5, shape=(), dtype=float32)


m$reset_state()
m$update_state(rbind(c(0., 1.), c(1., 1.)), rbind(c(1., 0.), c(1., 1.)),
               sample_weight = c(0.3, 0.7))
m$result()

## tf.Tensor(0.7, shape=(), dtype=float32)
```

Usage with compile() API:

```
model %>% compile(
  optimizer = 'sgd',
  loss = 'mse',
  metrics = list(metric_cosine_similarity(axis=2)))
```

See Also

- https://keras.io/api/metrics/regression_metrics#cosinesimilarity-class

Other regression metrics:
metric_log_cosh_error()
metric_mean_absolute_error()
metric_mean_absolute_percentage_error()
metric_mean_squared_error()
metric_mean_squared_logarithmic_error()
metric_r2_score()
metric_root_mean_squared_error()

Other metrics:
Metric()
custom_metric()
metric_auc()
metric_binary_accuracy()

metric_binary_crossentropy()
metric_binary_focal_crossentropy()
metric_binary_iou()
metric_categorical_accuracy()
metric_categorical_crossentropy()
metric_categorical_focal_crossentropy()
metric_categorical_hinge()
metric_f1_score()
metric_false_negatives()
metric_false_positives()
metric_fbeta_score()
metric_hinge()
metric_huber()
metric_iou()
metric_kl_divergence()
metric_log_cosh()
metric_log_cosh_error()
metric_mean()
metric_mean_absolute_error()
metric_mean_absolute_percentage_error()
metric_mean_iou()
metric_mean_squared_error()
metric_mean_squared_logarithmic_error()
metric_mean_wrapper()
metric_one_hot_iou()
metric_one_hot_mean_iou()
metric_poisson()
metric_precision()
metric_precision_at_recall()
metric_r2_score()
metric_recall()
metric_recall_at_precision()
metric_root_mean_squared_error()
metric_sensitivity_at_specificity()
metric_sparse_categorical_accuracy()
metric_sparse_categorical_crossentropy()
metric_sparse_top_k_categorical_accuracy()
metric_specificity_at_sensitivity()
metric_squared_hinge()
metric_sum()
metric_top_k_categorical_accuracy()
metric_true_negatives()
metric_true_positives()

metric_f1_score *Computes F-1 Score.*

Description

Formula:

```
f1_score <- 2 * (precision * recall) / (precision + recall)
```

This is the harmonic mean of precision and recall. Its output range is `[0, 1]`. It works for both multi-class and multi-label classification.

Usage

```
metric_f1_score(
  ...,
  average = NULL,
  threshold = NULL,
  name = "f1_score",
  dtype = NULL
)
```

Arguments

`...`	For forward/backward compatability.
`average`	Type of averaging to be performed on data. Acceptable values are `NULL`, `"micro"`, `"macro"` and `"weighted"`. Defaults to `NULL`. If `NULL`, no averaging is performed and `result()` will return the score for each class. If `"micro"`, compute metrics globally by counting the total true positives, false negatives and false positives. If `"macro"`, compute metrics for each label, and return their unweighted mean. This does not take label imbalance into account. If `"weighted"`, compute metrics for each label, and return their average weighted by support (the number of true instances for each label). This alters `"macro"` to account for label imbalance. It can result in an score that is not between precision and recall.
`threshold`	Elements of `y_pred` greater than `threshold` are converted to be 1, and the rest 0. If `threshold` is `NULL`, the argmax of `y_pred` is converted to 1, and the rest to 0.
`name`	Optional. String name of the metric instance.
`dtype`	Optional. Data type of the metric result.

Value

a `Metric` instance is returned. The `Metric` instance can be passed directly to `compile(metrics = )`, or used as a standalone object. See `?Metric` for example usage.

Examples

```
metric <- metric_f1_score(threshold = 0.5)
y_true <- rbind(c(1, 1, 1),
                c(1, 0, 0),
                c(1, 1, 0))
```

```
y_pred <- rbind(c(0.2, 0.6, 0.7),
                c(0.2, 0.6, 0.6),
                c(0.6, 0.8, 0.0))
metric$update_state(y_true, y_pred)
result <- metric$result()
result

## tf.Tensor([0.49999997 0.79999995 0.66666657], shape=(3), dtype=float32)
```

Returns

F-1 Score: float.

See Also

Other f score metrics:
metric_fbeta_score()

Other metrics:
Metric()
custom_metric()
metric_auc()
metric_binary_accuracy()
metric_binary_crossentropy()
metric_binary_focal_crossentropy()
metric_binary_iou()
metric_categorical_accuracy()
metric_categorical_crossentropy()
metric_categorical_focal_crossentropy()
metric_categorical_hinge()
metric_cosine_similarity()
metric_false_negatives()
metric_false_positives()
metric_fbeta_score()
metric_hinge()
metric_huber()
metric_iou()
metric_kl_divergence()
metric_log_cosh()
metric_log_cosh_error()
metric_mean()
metric_mean_absolute_error()
metric_mean_absolute_percentage_error()
metric_mean_iou()
metric_mean_squared_error()
metric_mean_squared_logarithmic_error()
metric_mean_wrapper()
metric_one_hot_iou()

metric_one_hot_mean_iou()
metric_poisson()
metric_precision()
metric_precision_at_recall()
metric_r2_score()
metric_recall()
metric_recall_at_precision()
metric_root_mean_squared_error()
metric_sensitivity_at_specificity()
metric_sparse_categorical_accuracy()
metric_sparse_categorical_crossentropy()
metric_sparse_top_k_categorical_accuracy()
metric_specificity_at_sensitivity()
metric_squared_hinge()
metric_sum()
metric_top_k_categorical_accuracy()
metric_true_negatives()
metric_true_positives()

---

metric_false_negatives

Calculates the number of false negatives.

Description

If sample_weight is given, calculates the sum of the weights of false negatives. This metric creates one local variable, accumulator that is used to keep track of the number of false negatives.

If sample_weight is NULL, weights default to 1. Use sample_weight of 0 to mask values.

Usage

```
metric_false_negatives(..., thresholds = NULL, name = NULL, dtype = NULL)
```

Arguments

...	For forward/backward compatability.
thresholds	(Optional) Defaults to 0.5. A float value, or a Python list of float threshold values in [0, 1]. A threshold is compared with prediction values to determine the truth value of predictions (i.e., above the threshold is TRUE, below is FALSE). If used with a loss function that sets from_logits=TRUE (i.e. no sigmoid applied to predictions), thresholds should be set to 0. One metric value is generated for each threshold value.
name	(Optional) string name of the metric instance.
dtype	(Optional) data type of the metric result.

Value

a `Metric` instance is returned. The `Metric` instance can be passed directly to `compile(metrics = )`, or used as a standalone object. See `?Metric` for example usage.

Usage

Standalone usage:

```
m <- metric_false_negatives()
m$update_state(c(0, 1, 1, 1), c(0, 1, 0, 0))
m$result()

## tf.Tensor(2.0, shape=(), dtype=float32)


m$reset_state()
m$update_state(c(0, 1, 1, 1), c(0, 1, 0, 0), sample_weight=c(0, 0, 1, 0))
m$result()

## tf.Tensor(1.0, shape=(), dtype=float32)


# 1.0
```

See Also

- https://keras.io/api/metrics/classification_metrics#falsenegatives-class

Other confusion metrics:
`metric_auc()`
`metric_false_positives()`
`metric_precision()`
`metric_precision_at_recall()`
`metric_recall()`
`metric_recall_at_precision()`
`metric_sensitivity_at_specificity()`
`metric_specificity_at_sensitivity()`
`metric_true_negatives()`
`metric_true_positives()`

Other metrics:
`Metric()`
`custom_metric()`
`metric_auc()`
`metric_binary_accuracy()`
`metric_binary_crossentropy()`
`metric_binary_focal_crossentropy()`
`metric_binary_iou()`

metric_categorical_accuracy()
metric_categorical_crossentropy()
metric_categorical_focal_crossentropy()
metric_categorical_hinge()
metric_cosine_similarity()
metric_f1_score()
metric_false_positives()
metric_fbeta_score()
metric_hinge()
metric_huber()
metric_iou()
metric_kl_divergence()
metric_log_cosh()
metric_log_cosh_error()
metric_mean()
metric_mean_absolute_error()
metric_mean_absolute_percentage_error()
metric_mean_iou()
metric_mean_squared_error()
metric_mean_squared_logarithmic_error()
metric_mean_wrapper()
metric_one_hot_iou()
metric_one_hot_mean_iou()
metric_poisson()
metric_precision()
metric_precision_at_recall()
metric_r2_score()
metric_recall()
metric_recall_at_precision()
metric_root_mean_squared_error()
metric_sensitivity_at_specificity()
metric_sparse_categorical_accuracy()
metric_sparse_categorical_crossentropy()
metric_sparse_top_k_categorical_accuracy()
metric_specificity_at_sensitivity()
metric_squared_hinge()
metric_sum()
metric_top_k_categorical_accuracy()
metric_true_negatives()
metric_true_positives()

metric_false_positives

Calculates the number of false positives.

Description

If `sample_weight` is given, calculates the sum of the weights of false positives. This metric creates one local variable, `accumulator` that is used to keep track of the number of false positives.

If `sample_weight` is `NULL`, weights default to 1. Use `sample_weight` of 0 to mask values.

Usage

```
metric_false_positives(..., thresholds = NULL, name = NULL, dtype = NULL)
```

Arguments

`...`	For forward/backward compatability.
`thresholds`	(Optional) Defaults to `0.5`. A float value, or a Python list of float threshold values in `[0, 1]`. A threshold is compared with prediction values to determine the truth value of predictions (i.e., above the threshold is `TRUE`, below is `FALSE`). If used with a loss function that sets `from_logits=TRUE` (i.e. no sigmoid applied to predictions), `thresholds` should be set to 0. One metric value is generated for each threshold value.
`name`	(Optional) string name of the metric instance.
`dtype`	(Optional) data type of the metric result.

Value

a `Metric` instance is returned. The `Metric` instance can be passed directly to `compile(metrics = )`, or used as a standalone object. See `?Metric` for example usage.

Usage

Standalone usage:

```
m <- metric_false_positives()
m$update_state(c(0, 1, 0, 0), c(0, 0, 1, 1))
m$result()
```

```
## tf.Tensor(2.0, shape=(), dtype=float32)
```

```
m$reset_state()
m$update_state(c(0, 1, 0, 0), c(0, 0, 1, 1), sample_weight = c(0, 0, 1, 0))
m$result()
```

```
## tf.Tensor(1.0, shape=(), dtype=float32)
```

See Also

- https://keras.io/api/metrics/classification_metrics#falsepositives-class

Other confusion metrics:
metric_auc()
metric_false_negatives()
metric_precision()
metric_precision_at_recall()
metric_recall()
metric_recall_at_precision()
metric_sensitivity_at_specificity()
metric_specificity_at_sensitivity()
metric_true_negatives()
metric_true_positives()

Other metrics:
Metric()
custom_metric()
metric_auc()
metric_binary_accuracy()
metric_binary_crossentropy()
metric_binary_focal_crossentropy()
metric_binary_iou()
metric_categorical_accuracy()
metric_categorical_crossentropy()
metric_categorical_focal_crossentropy()
metric_categorical_hinge()
metric_cosine_similarity()
metric_f1_score()
metric_false_negatives()
metric_fbeta_score()
metric_hinge()
metric_huber()
metric_iou()
metric_kl_divergence()
metric_log_cosh()
metric_log_cosh_error()
metric_mean()
metric_mean_absolute_error()
metric_mean_absolute_percentage_error()
metric_mean_iou()
metric_mean_squared_error()
metric_mean_squared_logarithmic_error()
metric_mean_wrapper()
metric_one_hot_iou()
metric_one_hot_mean_iou()
metric_poisson()
metric_precision()

metric_precision_at_recall()
metric_r2_score()
metric_recall()
metric_recall_at_precision()
metric_root_mean_squared_error()
metric_sensitivity_at_specificity()
metric_sparse_categorical_accuracy()
metric_sparse_categorical_crossentropy()
metric_sparse_top_k_categorical_accuracy()
metric_specificity_at_sensitivity()
metric_squared_hinge()
metric_sum()
metric_top_k_categorical_accuracy()
metric_true_negatives()
metric_true_positives()

---

metric_fbeta_score *Computes F-Beta score.*

Description

Formula:

```
b2 <- beta^2
f_beta_score <- (1 + b2) * (precision * recall) / (precision * b2 + recall)
```

This is the weighted harmonic mean of precision and recall. Its output range is `[0, 1]`. It works for both multi-class and multi-label classification.

Usage

```
metric_fbeta_score(
  ...,
  average = NULL,
  beta = 1,
  threshold = NULL,
  name = "fbeta_score",
  dtype = NULL
)
```

Arguments

...	For forward/backward compatability.

average	Type of averaging to be performed across per-class results in the multi-class case. Acceptable values are NULL, "micro", "macro" and "weighted". Defaults to NULL. If NULL, no averaging is performed and result() will return the score for each class. If "micro", compute metrics globally by counting the total true positives, false negatives and false positives. If "macro", compute metrics for each label, and return their unweighted mean. This does not take label imbalance into account. If "weighted", compute metrics for each label, and return their average weighted by support (the number of true instances for each label). This alters "macro" to account for label imbalance. It can result in an score that is not between precision and recall.
beta	Determines the weight of given to recall in the harmonic mean between precision and recall (see pseudocode equation above). Defaults to 1.
threshold	Elements of y_pred greater than threshold are converted to be 1, and the rest 0. If threshold is NULL, the argmax of y_pred is converted to 1, and the rest to 0.
name	Optional. String name of the metric instance.
dtype	Optional. Data type of the metric result.

Value

a Metric instance is returned. The Metric instance can be passed directly to compile(metrics =), or used as a standalone object. See ?Metric for example usage.

Examples

```
metric <- metric_fbeta_score(beta = 2.0, threshold = 0.5)
y_true <- rbind(c(1, 1, 1),
                c(1, 0, 0),
                c(1, 1, 0))
y_pred <- rbind(c(0.2, 0.6, 0.7),
                c(0.2, 0.6, 0.6),
                c(0.6, 0.8, 0.0))
metric$update_state(y_true, y_pred)
metric$result()

## tf.Tensor([0.3846154  0.90909094 0.8333332 ], shape=(3), dtype=float32)
```

Returns

F-Beta Score: float.

See Also

Other f score metrics:
metric_f1_score()

Other metrics:
`Metric()`
`custom_metric()`
`metric_auc()`
`metric_binary_accuracy()`
`metric_binary_crossentropy()`
`metric_binary_focal_crossentropy()`
`metric_binary_iou()`
`metric_categorical_accuracy()`
`metric_categorical_crossentropy()`
`metric_categorical_focal_crossentropy()`
`metric_categorical_hinge()`
`metric_cosine_similarity()`
`metric_f1_score()`
`metric_false_negatives()`
`metric_false_positives()`
`metric_hinge()`
`metric_huber()`
`metric_iou()`
`metric_kl_divergence()`
`metric_log_cosh()`
`metric_log_cosh_error()`
`metric_mean()`
`metric_mean_absolute_error()`
`metric_mean_absolute_percentage_error()`
`metric_mean_iou()`
`metric_mean_squared_error()`
`metric_mean_squared_logarithmic_error()`
`metric_mean_wrapper()`
`metric_one_hot_iou()`
`metric_one_hot_mean_iou()`
`metric_poisson()`
`metric_precision()`
`metric_precision_at_recall()`
`metric_r2_score()`
`metric_recall()`
`metric_recall_at_precision()`
`metric_root_mean_squared_error()`
`metric_sensitivity_at_specificity()`
`metric_sparse_categorical_accuracy()`
`metric_sparse_categorical_crossentropy()`
`metric_sparse_top_k_categorical_accuracy()`
`metric_specificity_at_sensitivity()`
`metric_squared_hinge()`
`metric_sum()`
`metric_top_k_categorical_accuracy()`
`metric_true_negatives()`
`metric_true_positives()`

metric_hinge — *Computes the hinge metric between* `y_true` *and* `y_pred`.

Description

Formula:

```
loss <- mean(maximum(1 - y_true * y_pred, 0), axis=-1)
```

`y_true` values are expected to be -1 or 1. If binary (0 or 1) labels are provided we will convert them to -1 or 1.

Usage

```
metric_hinge(y_true, y_pred, ..., name = "hinge", dtype = NULL)
```

Arguments

`y_true`	The ground truth values. `y_true` values are expected to be -1 or 1. If binary (0 or 1) labels are provided they will be converted to -1 or 1 with shape = `[batch_size, d0, .. dN]`.
`y_pred`	The predicted values with shape = `[batch_size, d0, .. dN]`.
`...`	For forward/backward compatability.
`name`	(Optional) string name of the metric instance.
`dtype`	(Optional) data type of the metric result.

Value

If `y_true` and `y_pred` are missing, a `Metric` instance is returned. The `Metric` instance that can be passed directly to `compile(metrics = )`, or used as a standalone object. See `?Metric` for example usage. If `y_true` and `y_pred` are provided, then a tensor with the computed value is returned.

Usage

Standalone usage:

```
m <- metric_hinge()
m$update_state(rbind(c(0, 1), c(0, 0)), rbind(c(0.6, 0.4), c(0.4, 0.6)))
m$result()
```

```
## tf.Tensor(1.3, shape=(), dtype=float32)
```

```
m$reset_state()
m$update_state(rbind(c(0, 1), c(0, 0)), rbind(c(0.6, 0.4), c(0.4, 0.6)),
               sample_weight = c(1, 0))
m$result()

## tf.Tensor(1.1, shape=(), dtype=float32)
```

See Also

- https://keras.io/api/metrics/hinge_metrics#hinge-class

Other losses:
Loss()
loss_binary_crossentropy()
loss_binary_focal_crossentropy()
loss_categorical_crossentropy()
loss_categorical_focal_crossentropy()
loss_categorical_hinge()
loss_cosine_similarity()
loss_dice()
loss_hinge()
loss_huber()
loss_kl_divergence()
loss_log_cosh()
loss_mean_absolute_error()
loss_mean_absolute_percentage_error()
loss_mean_squared_error()
loss_mean_squared_logarithmic_error()
loss_poisson()
loss_sparse_categorical_crossentropy()
loss_squared_hinge()
metric_binary_crossentropy()
metric_binary_focal_crossentropy()
metric_categorical_crossentropy()
metric_categorical_focal_crossentropy()
metric_categorical_hinge()
metric_huber()
metric_kl_divergence()
metric_log_cosh()
metric_mean_absolute_error()
metric_mean_absolute_percentage_error()
metric_mean_squared_error()
metric_mean_squared_logarithmic_error()
metric_poisson()
metric_sparse_categorical_crossentropy()
metric_squared_hinge()

Other metrics:

Metric()
custom_metric()
metric_auc()
metric_binary_accuracy()
metric_binary_crossentropy()
metric_binary_focal_crossentropy()
metric_binary_iou()
metric_categorical_accuracy()
metric_categorical_crossentropy()
metric_categorical_focal_crossentropy()
metric_categorical_hinge()
metric_cosine_similarity()
metric_f1_score()
metric_false_negatives()
metric_false_positives()
metric_fbeta_score()
metric_huber()
metric_iou()
metric_kl_divergence()
metric_log_cosh()
metric_log_cosh_error()
metric_mean()
metric_mean_absolute_error()
metric_mean_absolute_percentage_error()
metric_mean_iou()
metric_mean_squared_error()
metric_mean_squared_logarithmic_error()
metric_mean_wrapper()
metric_one_hot_iou()
metric_one_hot_mean_iou()
metric_poisson()
metric_precision()
metric_precision_at_recall()
metric_r2_score()
metric_recall()
metric_recall_at_precision()
metric_root_mean_squared_error()
metric_sensitivity_at_specificity()
metric_sparse_categorical_accuracy()
metric_sparse_categorical_crossentropy()
metric_sparse_top_k_categorical_accuracy()
metric_specificity_at_sensitivity()
metric_squared_hinge()
metric_sum()
metric_top_k_categorical_accuracy()
metric_true_negatives()
metric_true_positives()

Other hinge metrics:
metric_categorical_hinge()
metric_squared_hinge()

metric_huber *Computes Huber loss value.*

Description

Formula:

```
for (x in error) {
  if (abs(x) <= delta){
    loss <- c(loss, (0.5 * x^2))
  } else if (abs(x) > delta) {
    loss <- c(loss, (delta * abs(x) - 0.5 * delta^2))
  }
}
loss <- mean(loss)
```

See: Huber loss.

Usage

```
metric_huber(y_true, y_pred, delta = 1)
```

Arguments

y_true	tensor of true targets.
y_pred	tensor of predicted targets.
delta	A float, the point where the Huber loss function changes from a quadratic to linear. Defaults to 1.0.

Value

Tensor with one scalar loss entry per sample.

Examples

```
y_true <- rbind(c(0, 1), c(0, 0))
y_pred <- rbind(c(0.6, 0.4), c(0.4, 0.6))
loss <- loss_huber(y_true, y_pred)
```

See Also

Other losses:
`Loss()`
`loss_binary_crossentropy()`
`loss_binary_focal_crossentropy()`
`loss_categorical_crossentropy()`
`loss_categorical_focal_crossentropy()`
`loss_categorical_hinge()`
`loss_cosine_similarity()`
`loss_dice()`
`loss_hinge()`
`loss_huber()`
`loss_kl_divergence()`
`loss_log_cosh()`
`loss_mean_absolute_error()`
`loss_mean_absolute_percentage_error()`
`loss_mean_squared_error()`
`loss_mean_squared_logarithmic_error()`
`loss_poisson()`
`loss_sparse_categorical_crossentropy()`
`loss_squared_hinge()`
`metric_binary_crossentropy()`
`metric_binary_focal_crossentropy()`
`metric_categorical_crossentropy()`
`metric_categorical_focal_crossentropy()`
`metric_categorical_hinge()`
`metric_hinge()`
`metric_kl_divergence()`
`metric_log_cosh()`
`metric_mean_absolute_error()`
`metric_mean_absolute_percentage_error()`
`metric_mean_squared_error()`
`metric_mean_squared_logarithmic_error()`
`metric_poisson()`
`metric_sparse_categorical_crossentropy()`
`metric_squared_hinge()`

Other metrics:
`Metric()`
`custom_metric()`
`metric_auc()`
`metric_binary_accuracy()`
`metric_binary_crossentropy()`
`metric_binary_focal_crossentropy()`
`metric_binary_iou()`
`metric_categorical_accuracy()`
`metric_categorical_crossentropy()`
`metric_categorical_focal_crossentropy()`

metric_categorical_hinge()
metric_cosine_similarity()
metric_f1_score()
metric_false_negatives()
metric_false_positives()
metric_fbeta_score()
metric_hinge()
metric_iou()
metric_kl_divergence()
metric_log_cosh()
metric_log_cosh_error()
metric_mean()
metric_mean_absolute_error()
metric_mean_absolute_percentage_error()
metric_mean_iou()
metric_mean_squared_error()
metric_mean_squared_logarithmic_error()
metric_mean_wrapper()
metric_one_hot_iou()
metric_one_hot_mean_iou()
metric_poisson()
metric_precision()
metric_precision_at_recall()
metric_r2_score()
metric_recall()
metric_recall_at_precision()
metric_root_mean_squared_error()
metric_sensitivity_at_specificity()
metric_sparse_categorical_accuracy()
metric_sparse_categorical_crossentropy()
metric_sparse_top_k_categorical_accuracy()
metric_specificity_at_sensitivity()
metric_squared_hinge()
metric_sum()
metric_top_k_categorical_accuracy()
metric_true_negatives()
metric_true_positives()

---

metric_iou — *Computes the Intersection-Over-Union metric for specific target classes.*

Description

Formula:

```
iou <- true_positives / (true_positives + false_positives + false_negatives)
```

Intersection-Over-Union is a common evaluation metric for semantic image segmentation.

To compute IoUs, the predictions are accumulated in a confusion matrix, weighted by `sample_weight` and the metric is then calculated from it.

If `sample_weight` is `NULL`, weights default to 1. Use `sample_weight` of 0 to mask values.

Note, this class first computes IoUs for all individual classes, then returns the mean of IoUs for the classes that are specified by `target_class_ids`. If `target_class_ids` has only one id value, the IoU of that specific class is returned.

Usage

```
metric_iou(
  ...,
  num_classes,
  target_class_ids,
  name = NULL,
  dtype = NULL,
  ignore_class = NULL,
  sparse_y_true = TRUE,
  sparse_y_pred = TRUE,
  axis = -1L
)
```

Arguments

`...`	For forward/backward compatability.
`num_classes`	The possible number of labels the prediction task can have.
`target_class_ids`	A list of target class ids for which the metric is returned. To compute IoU for a specific class, a list of a single id value should be provided.
`name`	(Optional) string name of the metric instance.
`dtype`	(Optional) data type of the metric result.
`ignore_class`	Optional integer. The ID of a class to be ignored during metric computation. This is useful, for example, in segmentation problems featuring a "void" class (commonly -1 or 255) in segmentation maps. By default (`ignore_class=NULL`), all classes are considered.
`sparse_y_true`	Whether labels are encoded using integers or dense floating point vectors. If `FALSE`, the `argmax` function is used to determine each sample's most likely associated label.
`sparse_y_pred`	Whether predictions are encoded using integers or dense floating point vectors. If `FALSE`, the `argmax` function is used to determine each sample's most likely associated label.
`axis`	(Optional) -1 is the dimension containing the logits. Defaults to `-1`.

Value

a `Metric` instance is returned. The `Metric` instance can be passed directly to `compile(metrics = )`, or used as a standalone object. See `?Metric` for example usage.

Examples

Standalone usage:

```
m <- metric_iou(num_classes = 2L, target_class_ids = list(0L))
m$update_state(c(0, 0, 1, 1), c(0, 1, 0, 1))
m$result()

## tf.Tensor(0.3333333, shape=(), dtype=float32)


m$reset_state()
m$update_state(c(0, 0, 1, 1), c(0, 1, 0, 1),
               sample_weight = c(0.3, 0.3, 0.3, 0.1))
m$result()

## tf.Tensor(0.33333325, shape=(), dtype=float32)
```

Usage with `compile()` API:

```
model %>% compile(
  optimizer = 'sgd',
  loss = 'mse',
  metrics = list(metric_iou(num_classes = 2L, target_class_ids = list(0L))))
```

See Also

Other iou metrics:
`metric_binary_iou()`
`metric_mean_iou()`
`metric_one_hot_iou()`
`metric_one_hot_mean_iou()`

Other metrics:
`Metric()`
`custom_metric()`
`metric_auc()`
`metric_binary_accuracy()`
`metric_binary_crossentropy()`
`metric_binary_focal_crossentropy()`
`metric_binary_iou()`
`metric_categorical_accuracy()`

metric_categorical_crossentropy()
metric_categorical_focal_crossentropy()
metric_categorical_hinge()
metric_cosine_similarity()
metric_f1_score()
metric_false_negatives()
metric_false_positives()
metric_fbeta_score()
metric_hinge()
metric_huber()
metric_kl_divergence()
metric_log_cosh()
metric_log_cosh_error()
metric_mean()
metric_mean_absolute_error()
metric_mean_absolute_percentage_error()
metric_mean_iou()
metric_mean_squared_error()
metric_mean_squared_logarithmic_error()
metric_mean_wrapper()
metric_one_hot_iou()
metric_one_hot_mean_iou()
metric_poisson()
metric_precision()
metric_precision_at_recall()
metric_r2_score()
metric_recall()
metric_recall_at_precision()
metric_root_mean_squared_error()
metric_sensitivity_at_specificity()
metric_sparse_categorical_accuracy()
metric_sparse_categorical_crossentropy()
metric_sparse_top_k_categorical_accuracy()
metric_specificity_at_sensitivity()
metric_squared_hinge()
metric_sum()
metric_top_k_categorical_accuracy()
metric_true_negatives()
metric_true_positives()

---

metric_kl_divergence *Computes Kullback-Leibler divergence metric between* y_true *and*

Description

Formula:

```
loss <- y_true * log(y_true / y_pred)
```

Usage

```
metric_kl_divergence(y_true, y_pred, ..., name = "kl_divergence", dtype = NULL)
```

Arguments

`y_true`	Tensor of true targets.
`y_pred`	Tensor of predicted targets.
`...`	For forward/backward compatability.
`name`	(Optional) string name of the metric instance.
`dtype`	(Optional) data type of the metric result.

Value

If `y_true` and `y_pred` are missing, a `Metric` instance is returned. The `Metric` instance that can be passed directly to `compile(metrics = )`, or used as a standalone object. See `?Metric` for example usage. If `y_true` and `y_pred` are provided, then a tensor with the computed value is returned.

Usage

Standalone usage:

```
m <- metric_kl_divergence()
m$update_state(rbind(c(0, 1), c(0, 0)), rbind(c(0.6, 0.4), c(0.4, 0.6)))
m$result()
```

```
## tf.Tensor(0.45814303, shape=(), dtype=float32)
```

```
m$reset_state()
m$update_state(rbind(c(0, 1), c(0, 0)), rbind(c(0.6, 0.4), c(0.4, 0.6)),
               sample_weight = c(1, 0))
m$result()
```

```
## tf.Tensor(0.91628915, shape=(), dtype=float32)
```

Usage with `compile()` API:

```
model %>% compile(optimizer = 'sgd',
                  loss = 'mse',
                  metrics = list(metric_kl_divergence()))
```

See Also

- https://keras.io/api/metrics/probabilistic_metrics#kldivergence-class

Other losses:
Loss()
loss_binary_crossentropy()
loss_binary_focal_crossentropy()
loss_categorical_crossentropy()
loss_categorical_focal_crossentropy()
loss_categorical_hinge()
loss_cosine_similarity()
loss_dice()
loss_hinge()
loss_huber()
loss_kl_divergence()
loss_log_cosh()
loss_mean_absolute_error()
loss_mean_absolute_percentage_error()
loss_mean_squared_error()
loss_mean_squared_logarithmic_error()
loss_poisson()
loss_sparse_categorical_crossentropy()
loss_squared_hinge()
metric_binary_crossentropy()
metric_binary_focal_crossentropy()
metric_categorical_crossentropy()
metric_categorical_focal_crossentropy()
metric_categorical_hinge()
metric_hinge()
metric_huber()
metric_log_cosh()
metric_mean_absolute_error()
metric_mean_absolute_percentage_error()
metric_mean_squared_error()
metric_mean_squared_logarithmic_error()
metric_poisson()
metric_sparse_categorical_crossentropy()
metric_squared_hinge()

Other metrics:
Metric()
custom_metric()
metric_auc()
metric_binary_accuracy()
metric_binary_crossentropy()
metric_binary_focal_crossentropy()
metric_binary_iou()
metric_categorical_accuracy()

metric_categorical_crossentropy()
metric_categorical_focal_crossentropy()
metric_categorical_hinge()
metric_cosine_similarity()
metric_f1_score()
metric_false_negatives()
metric_false_positives()
metric_fbeta_score()
metric_hinge()
metric_huber()
metric_iou()
metric_log_cosh()
metric_log_cosh_error()
metric_mean()
metric_mean_absolute_error()
metric_mean_absolute_percentage_error()
metric_mean_iou()
metric_mean_squared_error()
metric_mean_squared_logarithmic_error()
metric_mean_wrapper()
metric_one_hot_iou()
metric_one_hot_mean_iou()
metric_poisson()
metric_precision()
metric_precision_at_recall()
metric_r2_score()
metric_recall()
metric_recall_at_precision()
metric_root_mean_squared_error()
metric_sensitivity_at_specificity()
metric_sparse_categorical_accuracy()
metric_sparse_categorical_crossentropy()
metric_sparse_top_k_categorical_accuracy()
metric_specificity_at_sensitivity()
metric_squared_hinge()
metric_sum()
metric_top_k_categorical_accuracy()
metric_true_negatives()
metric_true_positives()

Other probabilistic metrics:
metric_binary_crossentropy()
metric_categorical_crossentropy()
metric_poisson()
metric_sparse_categorical_crossentropy()

metric_log_cosh *Logarithm of the hyperbolic cosine of the prediction error.*

Description

Formula:

```
loss <- mean(log(cosh(y_pred - y_true)), axis=-1)
```

Note that `log(cosh(x))` is approximately equal to `(x ** 2) / 2` for small `x` and to `abs(x) - log(2)` for large `x`. This means that 'logcosh' works mostly like the mean squared error, but will not be so strongly affected by the occasional wildly incorrect prediction.

Usage

```
metric_log_cosh(y_true, y_pred)
```

Arguments

y_true	Ground truth values with shape = `[batch_size, d0, .. dN]`.
y_pred	The predicted values with shape = `[batch_size, d0, .. dN]`.

Value

Logcosh error values with shape = `[batch_size, d0, .. dN-1]`.

Examples

```
y_true <- rbind(c(0., 1.), c(0., 0.))
y_pred <- rbind(c(1., 1.), c(0., 0.))
loss <- metric_log_cosh(y_true, y_pred)
loss

## tf.Tensor([ 2.16890413e-01 -1.90465432e-09], shape=(2), dtype=float64)
```

See Also

Other losses:
`Loss()`
`loss_binary_crossentropy()`
`loss_binary_focal_crossentropy()`
`loss_categorical_crossentropy()`
`loss_categorical_focal_crossentropy()`
`loss_categorical_hinge()`
`loss_cosine_similarity()`
`loss_dice()`

loss_hinge()
loss_huber()
loss_kl_divergence()
loss_log_cosh()
loss_mean_absolute_error()
loss_mean_absolute_percentage_error()
loss_mean_squared_error()
loss_mean_squared_logarithmic_error()
loss_poisson()
loss_sparse_categorical_crossentropy()
loss_squared_hinge()
metric_binary_crossentropy()
metric_binary_focal_crossentropy()
metric_categorical_crossentropy()
metric_categorical_focal_crossentropy()
metric_categorical_hinge()
metric_hinge()
metric_huber()
metric_kl_divergence()
metric_mean_absolute_error()
metric_mean_absolute_percentage_error()
metric_mean_squared_error()
metric_mean_squared_logarithmic_error()
metric_poisson()
metric_sparse_categorical_crossentropy()
metric_squared_hinge()

Other metrics:
Metric()
custom_metric()
metric_auc()
metric_binary_accuracy()
metric_binary_crossentropy()
metric_binary_focal_crossentropy()
metric_binary_iou()
metric_categorical_accuracy()
metric_categorical_crossentropy()
metric_categorical_focal_crossentropy()
metric_categorical_hinge()
metric_cosine_similarity()
metric_f1_score()
metric_false_negatives()
metric_false_positives()
metric_fbeta_score()
metric_hinge()
metric_huber()
metric_iou()
metric_kl_divergence()

metric_log_cosh_error()
metric_mean()
metric_mean_absolute_error()
metric_mean_absolute_percentage_error()
metric_mean_iou()
metric_mean_squared_error()
metric_mean_squared_logarithmic_error()
metric_mean_wrapper()
metric_one_hot_iou()
metric_one_hot_mean_iou()
metric_poisson()
metric_precision()
metric_precision_at_recall()
metric_r2_score()
metric_recall()
metric_recall_at_precision()
metric_root_mean_squared_error()
metric_sensitivity_at_specificity()
metric_sparse_categorical_accuracy()
metric_sparse_categorical_crossentropy()
metric_sparse_top_k_categorical_accuracy()
metric_specificity_at_sensitivity()
metric_squared_hinge()
metric_sum()
metric_top_k_categorical_accuracy()
metric_true_negatives()
metric_true_positives()

---

metric_log_cosh_error *Computes the logarithm of the hyperbolic cosine of the prediction error.*

Description

Formula:

```
error <- y_pred - y_true
logcosh <- mean(log((exp(error) + exp(-error))/2), axis=-1)
```

Usage

```
metric_log_cosh_error(..., name = "logcosh", dtype = NULL)
```

Arguments

...	For forward/backward compatability.
name	(Optional) string name of the metric instance.
dtype	(Optional) data type of the metric result.

Value

a `Metric` instance is returned. The `Metric` instance can be passed directly to `compile(metrics = )`, or used as a standalone object. See `?Metric` for example usage.

Examples

Standalone usage:

```
m <- metric_log_cosh_error()
m$update_state(rbind(c(0, 1), c(0, 0)), rbind(c(1, 1), c(0, 0)))
m$result()

## tf.Tensor(0.108445205, shape=(), dtype=float32)


m$reset_state()
m$update_state(rbind(c(0, 1), c(0, 0)), rbind(c(1, 1), c(0, 0)),
               sample_weight = c(1, 0))
m$result()

## tf.Tensor(0.21689041, shape=(), dtype=float32)
```

Usage with `compile()` API:

```
model %>% compile(optimizer = 'sgd',
                  loss = 'mse',
                  metrics = list(metric_log_cosh_error()))
```

See Also

- https://keras.io/api/metrics/regression_metrics#logcosherror-class

Other regression metrics:
`metric_cosine_similarity()`
`metric_mean_absolute_error()`
`metric_mean_absolute_percentage_error()`
`metric_mean_squared_error()`
`metric_mean_squared_logarithmic_error()`
`metric_r2_score()`
`metric_root_mean_squared_error()`

Other metrics:
`Metric()`
`custom_metric()`
`metric_auc()`
`metric_binary_accuracy()`
`metric_binary_crossentropy()`

metric_binary_focal_crossentropy()
metric_binary_iou()
metric_categorical_accuracy()
metric_categorical_crossentropy()
metric_categorical_focal_crossentropy()
metric_categorical_hinge()
metric_cosine_similarity()
metric_f1_score()
metric_false_negatives()
metric_false_positives()
metric_fbeta_score()
metric_hinge()
metric_huber()
metric_iou()
metric_kl_divergence()
metric_log_cosh()
metric_mean()
metric_mean_absolute_error()
metric_mean_absolute_percentage_error()
metric_mean_iou()
metric_mean_squared_error()
metric_mean_squared_logarithmic_error()
metric_mean_wrapper()
metric_one_hot_iou()
metric_one_hot_mean_iou()
metric_poisson()
metric_precision()
metric_precision_at_recall()
metric_r2_score()
metric_recall()
metric_recall_at_precision()
metric_root_mean_squared_error()
metric_sensitivity_at_specificity()
metric_sparse_categorical_accuracy()
metric_sparse_categorical_crossentropy()
metric_sparse_top_k_categorical_accuracy()
metric_specificity_at_sensitivity()
metric_squared_hinge()
metric_sum()
metric_top_k_categorical_accuracy()
metric_true_negatives()
metric_true_positives()

metric_mean *Compute the (weighted) mean of the given values.*

Description

For example, if values is `c(1, 3, 5, 7)` then the mean is 4. If `sample_weight` was specified as `c(1, 1, 0, 0)` then the mean would be 2.

This metric creates two variables, `total` and `count`. The mean value returned is simply `total` divided by `count`.

Usage

```
metric_mean(..., name = "mean", dtype = NULL)
```

Arguments

`...`	For forward/backward compatability.
`name`	(Optional) string name of the metric instance.
`dtype`	(Optional) data type of the metric result.

Value

a `Metric` instance is returned. The `Metric` instance can be passed directly to `compile(metrics = )`, or used as a standalone object. See `?Metric` for example usage.

Examples

```
m <- metric_mean()
m$update_state(c(1, 3, 5, 7))
m$result()

## tf.Tensor(4.0, shape=(), dtype=float32)


m$reset_state()
m$update_state(c(1, 3, 5, 7), sample_weight = c(1, 1, 0, 0))
m$result()

## tf.Tensor(2.0, shape=(), dtype=float32)
```

See Also

Other reduction metrics:
`metric_mean_wrapper()`
`metric_sum()`

Other metrics:
`Metric()`

custom_metric()
metric_auc()
metric_binary_accuracy()
metric_binary_crossentropy()
metric_binary_focal_crossentropy()
metric_binary_iou()
metric_categorical_accuracy()
metric_categorical_crossentropy()
metric_categorical_focal_crossentropy()
metric_categorical_hinge()
metric_cosine_similarity()
metric_f1_score()
metric_false_negatives()
metric_false_positives()
metric_fbeta_score()
metric_hinge()
metric_huber()
metric_iou()
metric_kl_divergence()
metric_log_cosh()
metric_log_cosh_error()
metric_mean_absolute_error()
metric_mean_absolute_percentage_error()
metric_mean_iou()
metric_mean_squared_error()
metric_mean_squared_logarithmic_error()
metric_mean_wrapper()
metric_one_hot_iou()
metric_one_hot_mean_iou()
metric_poisson()
metric_precision()
metric_precision_at_recall()
metric_r2_score()
metric_recall()
metric_recall_at_precision()
metric_root_mean_squared_error()
metric_sensitivity_at_specificity()
metric_sparse_categorical_accuracy()
metric_sparse_categorical_crossentropy()
metric_sparse_top_k_categorical_accuracy()
metric_specificity_at_sensitivity()
metric_squared_hinge()
metric_sum()
metric_top_k_categorical_accuracy()
metric_true_negatives()
metric_true_positives()

`metric_mean_absolute_error` *Computes the mean absolute error between the labels and predictions.*

Description

Formula:

```
loss <- mean(abs(y_true - y_pred))
```

Usage

```
metric_mean_absolute_error(
  y_true,
  y_pred,
  ...,
  name = "mean_absolute_error",
  dtype = NULL
)
```

Arguments

`y_true`	Ground truth values with shape = `[batch_size, d0, .. dN]`.
`y_pred`	The predicted values with shape = `[batch_size, d0, .. dN]`.
`...`	For forward/backward compatability.
`name`	(Optional) string name of the metric instance.
`dtype`	(Optional) data type of the metric result.

Value

If `y_true` and `y_pred` are missing, a `Metric` instance is returned. The `Metric` instance that can be passed directly to `compile(metrics = )`, or used as a standalone object. See `?Metric` for example usage. If `y_true` and `y_pred` are provided, then a tensor with the computed value is returned.

Examples

Standalone usage:

```
m <- metric_mean_absolute_error()
m$update_state(rbind(c(0, 1), c(0, 0)), rbind(c(1, 1), c(0, 0)))
m$result()
```

```
## tf.Tensor(0.25, shape=(), dtype=float32)
```

```
m$reset_state()
m$update_state(rbind(c(0, 1), c(0, 0)), rbind(c(1, 1), c(0, 0)),
               sample_weight = c(1, 0))
m$result()

## tf.Tensor(0.5, shape=(), dtype=float32)
```

Usage with `compile()` API:

```
model %>% compile(
    optimizer = 'sgd',
    loss = 'mse',
    metrics = list(metric_mean_absolute_error()))
```

See Also

- https://keras.io/api/metrics/regression_metrics#meanabsoluteerror-class

Other losses:
Loss()
loss_binary_crossentropy()
loss_binary_focal_crossentropy()
loss_categorical_crossentropy()
loss_categorical_focal_crossentropy()
loss_categorical_hinge()
loss_cosine_similarity()
loss_dice()
loss_hinge()
loss_huber()
loss_kl_divergence()
loss_log_cosh()
loss_mean_absolute_error()
loss_mean_absolute_percentage_error()
loss_mean_squared_error()
loss_mean_squared_logarithmic_error()
loss_poisson()
loss_sparse_categorical_crossentropy()
loss_squared_hinge()
metric_binary_crossentropy()
metric_binary_focal_crossentropy()
metric_categorical_crossentropy()
metric_categorical_focal_crossentropy()
metric_categorical_hinge()
metric_hinge()
metric_huber()
metric_kl_divergence()
metric_log_cosh()
metric_mean_absolute_percentage_error()

metric_mean_squared_error()
metric_mean_squared_logarithmic_error()
metric_poisson()
metric_sparse_categorical_crossentropy()
metric_squared_hinge()

Other metrics:
Metric()
custom_metric()
metric_auc()
metric_binary_accuracy()
metric_binary_crossentropy()
metric_binary_focal_crossentropy()
metric_binary_iou()
metric_categorical_accuracy()
metric_categorical_crossentropy()
metric_categorical_focal_crossentropy()
metric_categorical_hinge()
metric_cosine_similarity()
metric_f1_score()
metric_false_negatives()
metric_false_positives()
metric_fbeta_score()
metric_hinge()
metric_huber()
metric_iou()
metric_kl_divergence()
metric_log_cosh()
metric_log_cosh_error()
metric_mean()
metric_mean_absolute_percentage_error()
metric_mean_iou()
metric_mean_squared_error()
metric_mean_squared_logarithmic_error()
metric_mean_wrapper()
metric_one_hot_iou()
metric_one_hot_mean_iou()
metric_poisson()
metric_precision()
metric_precision_at_recall()
metric_r2_score()
metric_recall()
metric_recall_at_precision()
metric_root_mean_squared_error()
metric_sensitivity_at_specificity()
metric_sparse_categorical_accuracy()
metric_sparse_categorical_crossentropy()
metric_sparse_top_k_categorical_accuracy()

metric_specificity_at_sensitivity()
metric_squared_hinge()
metric_sum()
metric_top_k_categorical_accuracy()
metric_true_negatives()
metric_true_positives()

Other regression metrics:
metric_cosine_similarity()
metric_log_cosh_error()
metric_mean_absolute_percentage_error()
metric_mean_squared_error()
metric_mean_squared_logarithmic_error()
metric_r2_score()
metric_root_mean_squared_error()

---

metric_mean_absolute_percentage_error

Computes mean absolute percentage error between `y_true` *and* `y_pred`.

Description

Formula:

```
loss <- 100 * mean(abs((y_true - y_pred) / y_true), axis=-1)
```

Division by zero is prevented by dividing by `maximum(y_true, epsilon)` where `epsilon` = `keras$backend$epsilon()` (default to `1e-7`).

Usage

```
metric_mean_absolute_percentage_error(
  y_true,
  y_pred,
  ...,
  name = "mean_absolute_percentage_error",
  dtype = NULL
)
```

Arguments

y_true	Ground truth values with shape = `[batch_size, d0, .. dN]`.
y_pred	The predicted values with shape = `[batch_size, d0, .. dN]`.
...	For forward/backward compatability.
name	(Optional) string name of the metric instance.
dtype	(Optional) data type of the metric result.

Value

If y_true and y_pred are missing, a Metric instance is returned. The Metric instance that can be passed directly to compile(metrics =), or used as a standalone object. See ?Metric for example usage. If y_true and y_pred are provided, then a tensor with the computed value is returned.

Examples

Standalone usage:

```
m <- metric_mean_absolute_percentage_error()
m$update_state(rbind(c(0, 1), c(0, 0)), rbind(c(1, 1), c(0, 0)))
m$result()

## tf.Tensor(250000000.0, shape=(), dtype=float32)



m$reset_state()
m$update_state(rbind(c(0, 1), c(0, 0)), rbind(c(1, 1), c(0, 0)),
               sample_weight = c(1, 0))
m$result()

## tf.Tensor(500000000.0, shape=(), dtype=float32)
```

Usage with compile() API:

```
model %>% compile(
    optimizer = 'sgd',
    loss = 'mse',
    metrics = list(metric_mean_absolute_percentage_error()))
```

See Also

- https://keras.io/api/metrics/regression_metrics#meanabsolutepercentageerror-class

Other losses:
Loss()
loss_binary_crossentropy()
loss_binary_focal_crossentropy()
loss_categorical_crossentropy()
loss_categorical_focal_crossentropy()
loss_categorical_hinge()
loss_cosine_similarity()
loss_dice()
loss_hinge()
loss_huber()
loss_kl_divergence()
loss_log_cosh()

loss_mean_absolute_error()
loss_mean_absolute_percentage_error()
loss_mean_squared_error()
loss_mean_squared_logarithmic_error()
loss_poisson()
loss_sparse_categorical_crossentropy()
loss_squared_hinge()
metric_binary_crossentropy()
metric_binary_focal_crossentropy()
metric_categorical_crossentropy()
metric_categorical_focal_crossentropy()
metric_categorical_hinge()
metric_hinge()
metric_huber()
metric_kl_divergence()
metric_log_cosh()
metric_mean_absolute_error()
metric_mean_squared_error()
metric_mean_squared_logarithmic_error()
metric_poisson()
metric_sparse_categorical_crossentropy()
metric_squared_hinge()

Other metrics:
Metric()
custom_metric()
metric_auc()
metric_binary_accuracy()
metric_binary_crossentropy()
metric_binary_focal_crossentropy()
metric_binary_iou()
metric_categorical_accuracy()
metric_categorical_crossentropy()
metric_categorical_focal_crossentropy()
metric_categorical_hinge()
metric_cosine_similarity()
metric_f1_score()
metric_false_negatives()
metric_false_positives()
metric_fbeta_score()
metric_hinge()
metric_huber()
metric_iou()
metric_kl_divergence()
metric_log_cosh()
metric_log_cosh_error()
metric_mean()
metric_mean_absolute_error()

metric_mean_iou()
metric_mean_squared_error()
metric_mean_squared_logarithmic_error()
metric_mean_wrapper()
metric_one_hot_iou()
metric_one_hot_mean_iou()
metric_poisson()
metric_precision()
metric_precision_at_recall()
metric_r2_score()
metric_recall()
metric_recall_at_precision()
metric_root_mean_squared_error()
metric_sensitivity_at_specificity()
metric_sparse_categorical_accuracy()
metric_sparse_categorical_crossentropy()
metric_sparse_top_k_categorical_accuracy()
metric_specificity_at_sensitivity()
metric_squared_hinge()
metric_sum()
metric_top_k_categorical_accuracy()
metric_true_negatives()
metric_true_positives()

Other regression metrics:
metric_cosine_similarity()
metric_log_cosh_error()
metric_mean_absolute_error()
metric_mean_squared_error()
metric_mean_squared_logarithmic_error()
metric_r2_score()
metric_root_mean_squared_error()

---

metric_mean_iou *Computes the mean Intersection-Over-Union metric.*

Description

Formula:

```
iou <- true_positives / (true_positives + false_positives + false_negatives)
```

Intersection-Over-Union is a common evaluation metric for semantic image segmentation.

To compute IoUs, the predictions are accumulated in a confusion matrix, weighted by `sample_weight` and the metric is then calculated from it.

If `sample_weight` is `NULL`, weights default to 1. Use `sample_weight` of 0 to mask values.

Note that this class first computes IoUs for all individual classes, then returns the mean of these values.

Usage

```
metric_mean_iou(
  ...,
  num_classes,
  name = NULL,
  dtype = NULL,
  ignore_class = NULL,
  sparse_y_true = TRUE,
  sparse_y_pred = TRUE,
  axis = -1L
)
```

Arguments

`...`	For forward/backward compatability.
`num_classes`	The possible number of labels the prediction task can have. This value must be provided, since a confusion matrix of dimension = `[num_classes, num_classes]` will be allocated.
`name`	(Optional) string name of the metric instance.
`dtype`	(Optional) data type of the metric result.
`ignore_class`	Optional integer. The ID of a class to be ignored during metric computation. This is useful, for example, in segmentation problems featuring a "void" class (commonly -1 or 255) in segmentation maps. By default (`ignore_class=NULL`), all classes are considered.
`sparse_y_true`	Whether labels are encoded using integers or dense floating point vectors. If `FALSE`, the `argmax` function is used to determine each sample's most likely associated label.
`sparse_y_pred`	Whether predictions are encoded using integers or dense floating point vectors. If `FALSE`, the `argmax` function is used to determine each sample's most likely associated label.
`axis`	(Optional) The dimension containing the logits. Defaults to `-1`.

Value

a `Metric` instance is returned. The `Metric` instance can be passed directly to `compile(metrics = )`, or used as a standalone object. See `?Metric` for example usage.

Examples

Standalone usage:

```
# cm = [[1, 1],
#        [1, 1]]
# sum_row = [2, 2], sum_col = [2, 2], true_positives = [1, 1]
# iou = true_positives / (sum_row + sum_col - true_positives))
# result = (1 / (2 + 2 - 1) + 1 / (2 + 2 - 1)) / 2 = 0.33
m <- metric_mean_iou(num_classes = 2)
m$update_state(c(0, 0, 1, 1), c(0, 1, 0, 1))
m$result()
```

```
## tf.Tensor(0.33333334, shape=(), dtype=float32)
```

```
m$reset_state()
m$update_state(c(0, 0, 1, 1), c(0, 1, 0, 1),
               sample_weight=c(0.3, 0.3, 0.3, 0.1))
m$result()
```

```
## tf.Tensor(0.2380952, shape=(), dtype=float32)
```

Usage with `compile()` API:

```
model %>% compile(
  optimizer = 'sgd',
  loss = 'mse',
  metrics = list(metric_mean_iou(num_classes=2)))
```

See Also

- https://keras.io/api/metrics/segmentation_metrics#meaniou-class

Other iou metrics:
metric_binary_iou()
metric_iou()
metric_one_hot_iou()
metric_one_hot_mean_iou()

Other metrics:
Metric()
custom_metric()
metric_auc()
metric_binary_accuracy()
metric_binary_crossentropy()
metric_binary_focal_crossentropy()
metric_binary_iou()
metric_categorical_accuracy()
metric_categorical_crossentropy()
metric_categorical_focal_crossentropy()
metric_categorical_hinge()

metric_cosine_similarity()
metric_f1_score()
metric_false_negatives()
metric_false_positives()
metric_fbeta_score()
metric_hinge()
metric_huber()
metric_iou()
metric_kl_divergence()
metric_log_cosh()
metric_log_cosh_error()
metric_mean()
metric_mean_absolute_error()
metric_mean_absolute_percentage_error()
metric_mean_squared_error()
metric_mean_squared_logarithmic_error()
metric_mean_wrapper()
metric_one_hot_iou()
metric_one_hot_mean_iou()
metric_poisson()
metric_precision()
metric_precision_at_recall()
metric_r2_score()
metric_recall()
metric_recall_at_precision()
metric_root_mean_squared_error()
metric_sensitivity_at_specificity()
metric_sparse_categorical_accuracy()
metric_sparse_categorical_crossentropy()
metric_sparse_top_k_categorical_accuracy()
metric_specificity_at_sensitivity()
metric_squared_hinge()
metric_sum()
metric_top_k_categorical_accuracy()
metric_true_negatives()
metric_true_positives()

---

metric_mean_squared_error   *Computes the mean squared error between* `y_true` *and* `y_pred`*.*

Description

Formula:

```
loss <- mean(square(y_true - y_pred))
```

Usage

```
metric_mean_squared_error(
  y_true,
  y_pred,
  ...,
  name = "mean_squared_error",
  dtype = NULL
)
```

Arguments

`y_true`	Ground truth values with shape = `[batch_size, d0, .. dN]`.
`y_pred`	The predicted values with shape = `[batch_size, d0, .. dN]`.
`...`	For forward/backward compatability.
`name`	(Optional) string name of the metric instance.
`dtype`	(Optional) data type of the metric result.

Value

If `y_true` and `y_pred` are missing, a `Metric` instance is returned. The `Metric` instance that can be passed directly to `compile(metrics = )`, or used as a standalone object. See `?Metric` for example usage. If `y_true` and `y_pred` are provided, then a tensor with the computed value is returned.

Examples

```
m <- metric_mean_squared_error()
m$update_state(rbind(c(0, 1), c(0, 0)), rbind(c(1, 1), c(0, 0)))
m$result()

## tf.Tensor(0.25, shape=(), dtype=float32)
```

See Also

- https://keras.io/api/metrics/regression_metrics#meansquarederror-class

Other losses:
`Loss()`
`loss_binary_crossentropy()`
`loss_binary_focal_crossentropy()`
`loss_categorical_crossentropy()`
`loss_categorical_focal_crossentropy()`
`loss_categorical_hinge()`
`loss_cosine_similarity()`
`loss_dice()`
`loss_hinge()`
`loss_huber()`
`loss_kl_divergence()`

loss_log_cosh()
loss_mean_absolute_error()
loss_mean_absolute_percentage_error()
loss_mean_squared_error()
loss_mean_squared_logarithmic_error()
loss_poisson()
loss_sparse_categorical_crossentropy()
loss_squared_hinge()
metric_binary_crossentropy()
metric_binary_focal_crossentropy()
metric_categorical_crossentropy()
metric_categorical_focal_crossentropy()
metric_categorical_hinge()
metric_hinge()
metric_huber()
metric_kl_divergence()
metric_log_cosh()
metric_mean_absolute_error()
metric_mean_absolute_percentage_error()
metric_mean_squared_logarithmic_error()
metric_poisson()
metric_sparse_categorical_crossentropy()
metric_squared_hinge()

Other metrics:
Metric()
custom_metric()
metric_auc()
metric_binary_accuracy()
metric_binary_crossentropy()
metric_binary_focal_crossentropy()
metric_binary_iou()
metric_categorical_accuracy()
metric_categorical_crossentropy()
metric_categorical_focal_crossentropy()
metric_categorical_hinge()
metric_cosine_similarity()
metric_f1_score()
metric_false_negatives()
metric_false_positives()
metric_fbeta_score()
metric_hinge()
metric_huber()
metric_iou()
metric_kl_divergence()
metric_log_cosh()
metric_log_cosh_error()
metric_mean()

metric_mean_absolute_error()
metric_mean_absolute_percentage_error()
metric_mean_iou()
metric_mean_squared_logarithmic_error()
metric_mean_wrapper()
metric_one_hot_iou()
metric_one_hot_mean_iou()
metric_poisson()
metric_precision()
metric_precision_at_recall()
metric_r2_score()
metric_recall()
metric_recall_at_precision()
metric_root_mean_squared_error()
metric_sensitivity_at_specificity()
metric_sparse_categorical_accuracy()
metric_sparse_categorical_crossentropy()
metric_sparse_top_k_categorical_accuracy()
metric_specificity_at_sensitivity()
metric_squared_hinge()
metric_sum()
metric_top_k_categorical_accuracy()
metric_true_negatives()
metric_true_positives()

Other regression metrics:
metric_cosine_similarity()
metric_log_cosh_error()
metric_mean_absolute_error()
metric_mean_absolute_percentage_error()
metric_mean_squared_logarithmic_error()
metric_r2_score()
metric_root_mean_squared_error()

---

metric_mean_squared_logarithmic_error

Computes mean squared logarithmic error between y_true *and* y_pred*.*

Description

Formula:

```
loss <- mean(square(log(y_true + 1) - log(y_pred + 1)), axis=-1)
```

Note that y_pred and y_true cannot be less or equal to 0. Negative values and 0 values will be replaced with keras$backend$epsilon() (default to 1e-7).

Usage

```
metric_mean_squared_logarithmic_error(
  y_true,
  y_pred,
  ...,
  name = "mean_squared_logarithmic_error",
  dtype = NULL
)
```

Arguments

y_true	Ground truth values with shape = [batch_size, d0, .. dN].
y_pred	The predicted values with shape = [batch_size, d0, .. dN].
...	For forward/backward compatability.
name	(Optional) string name of the metric instance.
dtype	(Optional) data type of the metric result.

Value

If y_true and y_pred are missing, a Metric instance is returned. The Metric instance that can be passed directly to compile(metrics =), or used as a standalone object. See ?Metric for example usage. If y_true and y_pred are provided, then a tensor with the computed value is returned.

Examples

Standalone usage:

```
m <- metric_mean_squared_logarithmic_error()
m$update_state(rbind(c(0, 1), c(0, 0)), rbind(c(1, 1), c(0, 0)))
m$result()

## tf.Tensor(0.12011322, shape=(), dtype=float32)


m$reset_state()
m$update_state(rbind(c(0, 1), c(0, 0)), rbind(c(1, 1), c(0, 0)),
               sample_weight = c(1, 0))
m$result()

## tf.Tensor(0.24022643, shape=(), dtype=float32)
```

Usage with compile() API:

```
model %>% compile(
  optimizer = 'sgd',
  loss = 'mse',
  metrics = list(metric_mean_squared_logarithmic_error()))
```

See Also

- https://keras.io/api/metrics/regression_metrics#meansquaredlogarithmicerror-class

Other losses:
Loss()
loss_binary_crossentropy()
loss_binary_focal_crossentropy()
loss_categorical_crossentropy()
loss_categorical_focal_crossentropy()
loss_categorical_hinge()
loss_cosine_similarity()
loss_dice()
loss_hinge()
loss_huber()
loss_kl_divergence()
loss_log_cosh()
loss_mean_absolute_error()
loss_mean_absolute_percentage_error()
loss_mean_squared_error()
loss_mean_squared_logarithmic_error()
loss_poisson()
loss_sparse_categorical_crossentropy()
loss_squared_hinge()
metric_binary_crossentropy()
metric_binary_focal_crossentropy()
metric_categorical_crossentropy()
metric_categorical_focal_crossentropy()
metric_categorical_hinge()
metric_hinge()
metric_huber()
metric_kl_divergence()
metric_log_cosh()
metric_mean_absolute_error()
metric_mean_absolute_percentage_error()
metric_mean_squared_error()
metric_poisson()
metric_sparse_categorical_crossentropy()
metric_squared_hinge()

Other metrics:
Metric()
custom_metric()
metric_auc()
metric_binary_accuracy()
metric_binary_crossentropy()
metric_binary_focal_crossentropy()
metric_binary_iou()
metric_categorical_accuracy()

metric_categorical_crossentropy()
metric_categorical_focal_crossentropy()
metric_categorical_hinge()
metric_cosine_similarity()
metric_f1_score()
metric_false_negatives()
metric_false_positives()
metric_fbeta_score()
metric_hinge()
metric_huber()
metric_iou()
metric_kl_divergence()
metric_log_cosh()
metric_log_cosh_error()
metric_mean()
metric_mean_absolute_error()
metric_mean_absolute_percentage_error()
metric_mean_iou()
metric_mean_squared_error()
metric_mean_wrapper()
metric_one_hot_iou()
metric_one_hot_mean_iou()
metric_poisson()
metric_precision()
metric_precision_at_recall()
metric_r2_score()
metric_recall()
metric_recall_at_precision()
metric_root_mean_squared_error()
metric_sensitivity_at_specificity()
metric_sparse_categorical_accuracy()
metric_sparse_categorical_crossentropy()
metric_sparse_top_k_categorical_accuracy()
metric_specificity_at_sensitivity()
metric_squared_hinge()
metric_sum()
metric_top_k_categorical_accuracy()
metric_true_negatives()
metric_true_positives()

Other regression metrics:
metric_cosine_similarity()
metric_log_cosh_error()
metric_mean_absolute_error()
metric_mean_absolute_percentage_error()
metric_mean_squared_error()
metric_r2_score()
metric_root_mean_squared_error()

metric_mean_wrapper *Wrap a stateless metric function with the* Mean *metric.*

Description

You could use this class to quickly build a mean metric from a function. The function needs to have the signature `fn(y_true, y_pred)` and return a per-sample loss array. `metric_mean_wrapper$result()` will return the average metric value across all samples seen so far.

For example:

```
mse <- function(y_true, y_pred) {
  (y_true - y_pred)^2
}

mse_metric <- metric_mean_wrapper(fn = mse)
mse_metric$update_state(c(0, 1), c(1, 1))
mse_metric$result()
```

```
## tf.Tensor(0.5, shape=(), dtype=float32)
```

Usage

```
metric_mean_wrapper(..., fn, name = NULL, dtype = NULL)
```

Arguments

`...`	Keyword arguments to pass on to `fn`.
`fn`	The metric function to wrap, with signature `fn(y_true, y_pred)`.
`name`	(Optional) string name of the metric instance.
`dtype`	(Optional) data type of the metric result.

Value

a `Metric` instance is returned. The `Metric` instance can be passed directly to `compile(metrics = )`, or used as a standalone object. See `?Metric` for example usage.

See Also

Other reduction metrics:
metric_mean()
metric_sum()

Other metrics:
Metric()
custom_metric()
metric_auc()
metric_binary_accuracy()
metric_binary_crossentropy()
metric_binary_focal_crossentropy()
metric_binary_iou()
metric_categorical_accuracy()
metric_categorical_crossentropy()
metric_categorical_focal_crossentropy()
metric_categorical_hinge()
metric_cosine_similarity()
metric_f1_score()
metric_false_negatives()
metric_false_positives()
metric_fbeta_score()
metric_hinge()
metric_huber()
metric_iou()
metric_kl_divergence()
metric_log_cosh()
metric_log_cosh_error()
metric_mean()
metric_mean_absolute_error()
metric_mean_absolute_percentage_error()
metric_mean_iou()
metric_mean_squared_error()
metric_mean_squared_logarithmic_error()
metric_one_hot_iou()
metric_one_hot_mean_iou()
metric_poisson()
metric_precision()
metric_precision_at_recall()
metric_r2_score()
metric_recall()
metric_recall_at_precision()
metric_root_mean_squared_error()
metric_sensitivity_at_specificity()
metric_sparse_categorical_accuracy()
metric_sparse_categorical_crossentropy()
metric_sparse_top_k_categorical_accuracy()
metric_specificity_at_sensitivity()

```
metric_squared_hinge()
metric_sum()
metric_top_k_categorical_accuracy()
metric_true_negatives()
metric_true_positives()
```

`metric_one_hot_iou` *Computes the Intersection-Over-Union metric for one-hot encoded labels.*

Description

Formula:

```
iou <- true_positives / (true_positives + false_positives + false_negatives)
```

Intersection-Over-Union is a common evaluation metric for semantic image segmentation.

To compute IoUs, the predictions are accumulated in a confusion matrix, weighted by `sample_weight` and the metric is then calculated from it.

If `sample_weight` is `NULL`, weights default to 1. Use `sample_weight` of 0 to mask values.

This class can be used to compute IoU for multi-class classification tasks where the labels are one-hot encoded (the last axis should have one dimension per class). Note that the predictions should also have the same shape. To compute the IoU, first the labels and predictions are converted back into integer format by taking the argmax over the class axis. Then the same computation steps as for the base `IoU` class apply.

Note, if there is only one channel in the labels and predictions, this class is the same as class `IoU`. In this case, use `IoU` instead.

Also, make sure that `num_classes` is equal to the number of classes in the data, to avoid a "labels out of bound" error when the confusion matrix is computed.

Usage

```
metric_one_hot_iou(
  ...,
  num_classes,
  target_class_ids,
  name = NULL,
  dtype = NULL,
  ignore_class = NULL,
  sparse_y_pred = FALSE,
  axis = -1L
)
```

Arguments

`...`	For forward/backward compatability.
`num_classes`	The possible number of labels the prediction task can have.
`target_class_ids`	A list or list of target class ids for which the metric is returned. To compute IoU for a specific class, a list (or list) of a single id value should be provided.
`name`	(Optional) string name of the metric instance.
`dtype`	(Optional) data type of the metric result.
`ignore_class`	Optional integer. The ID of a class to be ignored during metric computation. This is useful, for example, in segmentation problems featuring a "void" class (commonly -1 or 255) in segmentation maps. By default (`ignore_class=NULL`), all classes are considered.
`sparse_y_pred`	Whether predictions are encoded using integers or dense floating point vectors. If `FALSE`, the `argmax` function is used to determine each sample's most likely associated label.
`axis`	(Optional) The dimension containing the logits. Defaults to `-1`.

Value

a `Metric` instance is returned. The `Metric` instance can be passed directly to `compile(metrics = )`, or used as a standalone object. See `?Metric` for example usage.

Examples

Standalone usage:

```
y_true <- rbind(c(0, 0, 1), c(1, 0, 0), c(0, 1, 0), c(1, 0, 0))
y_pred <- rbind(c(0.2, 0.3, 0.5), c(0.1, 0.2, 0.7), c(0.5, 0.3, 0.1),
                c(0.1, 0.4, 0.5))
sample_weight <- c(0.1, 0.2, 0.3, 0.4)
m <- metric_one_hot_iou(num_classes = 3, target_class_ids = c(0, 2))
m$update_state(
    y_true = y_true, y_pred = y_pred, sample_weight = sample_weight)
m$result()
```

```
## tf.Tensor(0.07142855, shape=(), dtype=float32)
```

Usage with `compile()` API:

```
model %>% compile(
  optimizer = 'sgd',
  loss = 'mse',
  metrics = list(metric_one_hot_iou(
    num_classes = 3L,
    target_class_id = list(1L)
  ))
)
```

See Also

Other iou metrics:
metric_binary_iou()
metric_iou()
metric_mean_iou()
metric_one_hot_mean_iou()

Other metrics:
Metric()
custom_metric()
metric_auc()
metric_binary_accuracy()
metric_binary_crossentropy()
metric_binary_focal_crossentropy()
metric_binary_iou()
metric_categorical_accuracy()
metric_categorical_crossentropy()
metric_categorical_focal_crossentropy()
metric_categorical_hinge()
metric_cosine_similarity()
metric_f1_score()
metric_false_negatives()
metric_false_positives()
metric_fbeta_score()
metric_hinge()
metric_huber()
metric_iou()
metric_kl_divergence()
metric_log_cosh()
metric_log_cosh_error()
metric_mean()
metric_mean_absolute_error()
metric_mean_absolute_percentage_error()
metric_mean_iou()
metric_mean_squared_error()
metric_mean_squared_logarithmic_error()
metric_mean_wrapper()
metric_one_hot_mean_iou()
metric_poisson()
metric_precision()
metric_precision_at_recall()
metric_r2_score()
metric_recall()
metric_recall_at_precision()
metric_root_mean_squared_error()
metric_sensitivity_at_specificity()
metric_sparse_categorical_accuracy()
metric_sparse_categorical_crossentropy()

metric_sparse_top_k_categorical_accuracy()
metric_specificity_at_sensitivity()
metric_squared_hinge()
metric_sum()
metric_top_k_categorical_accuracy()
metric_true_negatives()
metric_true_positives()

---

metric_one_hot_mean_iou *Computes mean Intersection-Over-Union metric for one-hot encoded labels.*

Description

Formula:

```
iou <- true_positives / (true_positives + false_positives + false_negatives)
```

Intersection-Over-Union is a common evaluation metric for semantic image segmentation.

To compute IoUs, the predictions are accumulated in a confusion matrix, weighted by `sample_weight` and the metric is then calculated from it.

If `sample_weight` is `NULL`, weights default to 1. Use `sample_weight` of 0 to mask values.

This class can be used to compute the mean IoU for multi-class classification tasks where the labels are one-hot encoded (the last axis should have one dimension per class). Note that the predictions should also have the same shape. To compute the mean IoU, first the labels and predictions are converted back into integer format by taking the argmax over the class axis. Then the same computation steps as for the base `MeanIoU` class apply.

Note, if there is only one channel in the labels and predictions, this class is the same as class `metric_mean_iou`. In this case, use `metric_mean_iou` instead.

Also, make sure that `num_classes` is equal to the number of classes in the data, to avoid a "labels out of bound" error when the confusion matrix is computed.

Usage

```
metric_one_hot_mean_iou(
  ...,
  num_classes,
  name = NULL,
  dtype = NULL,
  ignore_class = NULL,
  sparse_y_pred = FALSE,
  axis = -1L
)
```

Arguments

...	For forward/backward compatability.
num_classes	The possible number of labels the prediction task can have.
name	(Optional) string name of the metric instance.
dtype	(Optional) data type of the metric result.
ignore_class	Optional integer. The ID of a class to be ignored during metric computation. This is useful, for example, in segmentation problems featuring a "void" class (commonly -1 or 255) in segmentation maps. By default (ignore_class=NULL), all classes are considered.
sparse_y_pred	Whether predictions are encoded using natural numbers or probability distribution vectors. If FALSE, the argmax function will be used to determine each sample's most likely associated label.
axis	(Optional) The dimension containing the logits. Defaults to -1.

Value

a Metric instance is returned. The Metric instance can be passed directly to compile(metrics =), or used as a standalone object. See ?Metric for example usage.

Examples

Standalone usage:

```
y_true <- rbind(c(0, 0, 1), c(1, 0, 0), c(0, 1, 0), c(1, 0, 0))
y_pred <- rbind(c(0.2, 0.3, 0.5), c(0.1, 0.2, 0.7), c(0.5, 0.3, 0.1),
                c(0.1, 0.4, 0.5))
sample_weight <- c(0.1, 0.2, 0.3, 0.4)
m <- metric_one_hot_mean_iou(num_classes = 3L)
m$update_state(
    y_true = y_true, y_pred = y_pred, sample_weight = sample_weight)
m$result()
```

```
## tf.Tensor(0.047619034, shape=(), dtype=float32)
```

Usage with compile() API:

```
model %>% compile(
    optimizer = 'sgd',
    loss = 'mse',
    metrics = list(metric_one_hot_mean_iou(num_classes = 3L)))
```

See Also

Other iou metrics:
metric_binary_iou()
metric_iou()
metric_mean_iou()
metric_one_hot_iou()

Other metrics:
Metric()
custom_metric()
metric_auc()
metric_binary_accuracy()
metric_binary_crossentropy()
metric_binary_focal_crossentropy()
metric_binary_iou()
metric_categorical_accuracy()
metric_categorical_crossentropy()
metric_categorical_focal_crossentropy()
metric_categorical_hinge()
metric_cosine_similarity()
metric_f1_score()
metric_false_negatives()
metric_false_positives()
metric_fbeta_score()
metric_hinge()
metric_huber()
metric_iou()
metric_kl_divergence()
metric_log_cosh()
metric_log_cosh_error()
metric_mean()
metric_mean_absolute_error()
metric_mean_absolute_percentage_error()
metric_mean_iou()
metric_mean_squared_error()
metric_mean_squared_logarithmic_error()
metric_mean_wrapper()
metric_one_hot_iou()
metric_poisson()
metric_precision()
metric_precision_at_recall()
metric_r2_score()
metric_recall()
metric_recall_at_precision()
metric_root_mean_squared_error()
metric_sensitivity_at_specificity()
metric_sparse_categorical_accuracy()
metric_sparse_categorical_crossentropy()

metric_sparse_top_k_categorical_accuracy()
metric_specificity_at_sensitivity()
metric_squared_hinge()
metric_sum()
metric_top_k_categorical_accuracy()
metric_true_negatives()
metric_true_positives()

---

metric_poisson *Computes the Poisson metric between* y_true *and* y_pred.

Description

Formula:

```
metric <- y_pred - y_true * log(y_pred)
```

Usage

```
metric_poisson(y_true, y_pred, ..., name = "poisson", dtype = NULL)
```

Arguments

y_true	Ground truth values. shape = [batch_size, d0, .. dN].
y_pred	The predicted values. shape = [batch_size, d0, .. dN].
...	For forward/backward compatability.
name	(Optional) string name of the metric instance.
dtype	(Optional) data type of the metric result.

Value

If y_true and y_pred are missing, a Metric instance is returned. The Metric instance that can be passed directly to compile(metrics =), or used as a standalone object. See ?Metric for example usage. If y_true and y_pred are provided, then a tensor with the computed value is returned.

Examples

Standalone usage:

```
m <- metric_poisson()
m$update_state(rbind(c(0, 1), c(0, 0)), rbind(c(1, 1), c(0, 0)))
m$result()
```

```
## tf.Tensor(0.49999997, shape=(), dtype=float32)
```

```
m$reset_state()
m$update_state(rbind(c(0, 1), c(0, 0)), rbind(c(1, 1), c(0, 0)),
               sample_weight = c(1, 0))
m$result()

## tf.Tensor(0.99999994, shape=(), dtype=float32)
```

Usage with `compile()` API:

```
model %>% compile(
  optimizer = 'sgd',
  loss = 'mse',
  metrics = list(metric_poisson())
)
```

See Also

- https://keras.io/api/metrics/probabilistic_metrics#poisson-class

Other losses:
`Loss()`
`loss_binary_crossentropy()`
`loss_binary_focal_crossentropy()`
`loss_categorical_crossentropy()`
`loss_categorical_focal_crossentropy()`
`loss_categorical_hinge()`
`loss_cosine_similarity()`
`loss_dice()`
`loss_hinge()`
`loss_huber()`
`loss_kl_divergence()`
`loss_log_cosh()`
`loss_mean_absolute_error()`
`loss_mean_absolute_percentage_error()`
`loss_mean_squared_error()`
`loss_mean_squared_logarithmic_error()`
`loss_poisson()`
`loss_sparse_categorical_crossentropy()`
`loss_squared_hinge()`
`metric_binary_crossentropy()`
`metric_binary_focal_crossentropy()`
`metric_categorical_crossentropy()`
`metric_categorical_focal_crossentropy()`
`metric_categorical_hinge()`
`metric_hinge()`
`metric_huber()`
`metric_kl_divergence()`
`metric_log_cosh()`

metric_mean_absolute_error()
metric_mean_absolute_percentage_error()
metric_mean_squared_error()
metric_mean_squared_logarithmic_error()
metric_sparse_categorical_crossentropy()
metric_squared_hinge()

Other metrics:
Metric()
custom_metric()
metric_auc()
metric_binary_accuracy()
metric_binary_crossentropy()
metric_binary_focal_crossentropy()
metric_binary_iou()
metric_categorical_accuracy()
metric_categorical_crossentropy()
metric_categorical_focal_crossentropy()
metric_categorical_hinge()
metric_cosine_similarity()
metric_f1_score()
metric_false_negatives()
metric_false_positives()
metric_fbeta_score()
metric_hinge()
metric_huber()
metric_iou()
metric_kl_divergence()
metric_log_cosh()
metric_log_cosh_error()
metric_mean()
metric_mean_absolute_error()
metric_mean_absolute_percentage_error()
metric_mean_iou()
metric_mean_squared_error()
metric_mean_squared_logarithmic_error()
metric_mean_wrapper()
metric_one_hot_iou()
metric_one_hot_mean_iou()
metric_precision()
metric_precision_at_recall()
metric_r2_score()
metric_recall()
metric_recall_at_precision()
metric_root_mean_squared_error()
metric_sensitivity_at_specificity()
metric_sparse_categorical_accuracy()
metric_sparse_categorical_crossentropy()

metric_sparse_top_k_categorical_accuracy()
metric_specificity_at_sensitivity()
metric_squared_hinge()
metric_sum()
metric_top_k_categorical_accuracy()
metric_true_negatives()
metric_true_positives()

Other probabilistic metrics:
metric_binary_crossentropy()
metric_categorical_crossentropy()
metric_kl_divergence()
metric_sparse_categorical_crossentropy()

metric_precision — *Computes the precision of the predictions with respect to the labels.*

Description

The metric creates two local variables, `true_positives` and `false_positives` that are used to compute the precision. This value is ultimately returned as `precision`, an idempotent operation that simply divides `true_positives` by the sum of `true_positives` and `false_positives`.

If `sample_weight` is `NULL`, weights default to 1. Use `sample_weight` of 0 to mask values.

If `top_k` is set, we'll calculate precision as how often on average a class among the top-k classes with the highest predicted values of a batch entry is correct and can be found in the label for that entry.

If `class_id` is specified, we calculate precision by considering only the entries in the batch for which `class_id` is above the threshold and/or in the top-k highest predictions, and computing the fraction of them for which `class_id` is indeed a correct label.

Usage

```
metric_precision(
  ...,
  thresholds = NULL,
  top_k = NULL,
  class_id = NULL,
  name = NULL,
  dtype = NULL
)
```

Arguments

`...`	For forward/backward compatability.
`thresholds`	(Optional) A float value, or a Python list of float threshold values in `[0, 1]`. A threshold is compared with prediction values to determine the truth value of predictions (i.e., above the threshold is `TRUE`, below is `FALSE`). If used with a loss function that sets `from_logits=TRUE` (i.e. no sigmoid applied to predictions), `thresholds` should be set to 0. One metric value is generated for each threshold value. If neither `thresholds` nor `top_k` are set, the default is to calculate precision with `thresholds=0.5`.
`top_k`	(Optional) Unset by default. An int value specifying the top-k predictions to consider when calculating precision.
`class_id`	(Optional) Integer class ID for which we want binary metrics. This must be in the half-open interval `[0, num_classes)`, where `num_classes` is the last dimension of predictions.
`name`	(Optional) string name of the metric instance.
`dtype`	(Optional) data type of the metric result.

Value

a `Metric` instance is returned. The `Metric` instance can be passed directly to `compile(metrics = )`, or used as a standalone object. See `?Metric` for example usage.

Usage

Standalone usage:

```
m <- metric_precision()
m$update_state(c(0, 1, 1, 1),
               c(1, 0, 1, 1))
m$result() |> as.double() |> signif()

## [1] 0.666667


m$reset_state()
m$update_state(c(0, 1, 1, 1),
               c(1, 0, 1, 1),
               sample_weight = c(0, 0, 1, 0))
m$result() |> as.double() |> signif()

## [1] 1


# With top_k=2, it will calculate precision over y_true[1:2]
# and y_pred[1:2]
m <- metric_precision(top_k = 2)
m$update_state(c(0, 0, 1, 1), c(1, 1, 1, 1))
m$result()
```

```
## tf.Tensor(0.0, shape=(), dtype=float32)


# With top_k=4, it will calculate precision over y_true[1:4]
# and y_pred[1:4]
m <- metric_precision(top_k = 4)
m$update_state(c(0, 0, 1, 1), c(1, 1, 1, 1))
m$result()

## tf.Tensor(0.5, shape=(), dtype=float32)
```

Usage with `compile()` API:

```
model |> compile(
  optimizer = 'sgd',
  loss = 'binary_crossentropy',
  metrics = list(metric_precision())
)
```

Usage with a loss with `from_logits=TRUE`:

```
model |> compile(
  optimizer = 'adam',
  loss = loss_binary_crossentropy(from_logits = TRUE),
  metrics = list(metric_precision(thresholds = 0))
)
```

See Also

- https://keras.io/api/metrics/classification_metrics#precision-class

Other confusion metrics:
`metric_auc()`
`metric_false_negatives()`
`metric_false_positives()`
`metric_precision_at_recall()`
`metric_recall()`
`metric_recall_at_precision()`
`metric_sensitivity_at_specificity()`
`metric_specificity_at_sensitivity()`
`metric_true_negatives()`
`metric_true_positives()`

Other metrics:
`Metric()`
`custom_metric()`
`metric_auc()`

metric_binary_accuracy()
metric_binary_crossentropy()
metric_binary_focal_crossentropy()
metric_binary_iou()
metric_categorical_accuracy()
metric_categorical_crossentropy()
metric_categorical_focal_crossentropy()
metric_categorical_hinge()
metric_cosine_similarity()
metric_f1_score()
metric_false_negatives()
metric_false_positives()
metric_fbeta_score()
metric_hinge()
metric_huber()
metric_iou()
metric_kl_divergence()
metric_log_cosh()
metric_log_cosh_error()
metric_mean()
metric_mean_absolute_error()
metric_mean_absolute_percentage_error()
metric_mean_iou()
metric_mean_squared_error()
metric_mean_squared_logarithmic_error()
metric_mean_wrapper()
metric_one_hot_iou()
metric_one_hot_mean_iou()
metric_poisson()
metric_precision_at_recall()
metric_r2_score()
metric_recall()
metric_recall_at_precision()
metric_root_mean_squared_error()
metric_sensitivity_at_specificity()
metric_sparse_categorical_accuracy()
metric_sparse_categorical_crossentropy()
metric_sparse_top_k_categorical_accuracy()
metric_specificity_at_sensitivity()
metric_squared_hinge()
metric_sum()
metric_top_k_categorical_accuracy()
metric_true_negatives()
metric_true_positives()

metric_precision_at_recall

Computes best precision where recall is >= specified value.

Description

This metric creates four local variables, `true_positives`, `true_negatives`, `false_positives` and `false_negatives` that are used to compute the precision at the given recall. The threshold for the given recall value is computed and used to evaluate the corresponding precision.

If `sample_weight` is `NULL`, weights default to 1. Use `sample_weight` of 0 to mask values.

If `class_id` is specified, we calculate precision by considering only the entries in the batch for which `class_id` is above the threshold predictions, and computing the fraction of them for which `class_id` is indeed a correct label.

Usage

```
metric_precision_at_recall(
  ...,
  recall,
  num_thresholds = 200L,
  class_id = NULL,
  name = NULL,
  dtype = NULL
)
```

Arguments

`...`	For forward/backward compatability.
`recall`	A scalar value in range `[0, 1]`.
`num_thresholds`	(Optional) Defaults to 200. The number of thresholds to use for matching the given recall.
`class_id`	(Optional) Integer class ID for which we want binary metrics. This must be in the half-open interval `[0, num_classes)`, where `num_classes` is the last dimension of predictions.
`name`	(Optional) string name of the metric instance.
`dtype`	(Optional) data type of the metric result.

Value

a `Metric` instance is returned. The `Metric` instance can be passed directly to `compile(metrics = )`, or used as a standalone object. See `?Metric` for example usage.

Usage

Standalone usage:

```
m <- metric_precision_at_recall(recall = 0.5)
m$update_state(c(0,   0,   0,   1,   1),
               c(0, 0.3, 0.8, 0.3, 0.8))
m$result()

## tf.Tensor(0.5, shape=(), dtype=float32)


m$reset_state()
m$update_state(c(0,   0,   0,   1,   1),
               c(0, 0.3, 0.8, 0.3, 0.8),
               sample_weight = c(2, 2, 2, 1, 1))
m$result()

## tf.Tensor(0.33333334, shape=(), dtype=float32)
```

Usage with `compile()` API:

```
model |> compile(
  optimizer = 'sgd',
  loss = 'binary_crossentropy',
  metrics = list(metric_precision_at_recall(recall = 0.8))
)
```

See Also

- `https://keras.io/api/metrics/classification_metrics#precisionatrecall-class`

Other confusion metrics:
`metric_auc()`
`metric_false_negatives()`
`metric_false_positives()`
`metric_precision()`
`metric_recall()`
`metric_recall_at_precision()`
`metric_sensitivity_at_specificity()`
`metric_specificity_at_sensitivity()`
`metric_true_negatives()`
`metric_true_positives()`

Other metrics:
`Metric()`
`custom_metric()`
`metric_auc()`
`metric_binary_accuracy()`
`metric_binary_crossentropy()`
`metric_binary_focal_crossentropy()`
`metric_binary_iou()`

metric_categorical_accuracy()
metric_categorical_crossentropy()
metric_categorical_focal_crossentropy()
metric_categorical_hinge()
metric_cosine_similarity()
metric_f1_score()
metric_false_negatives()
metric_false_positives()
metric_fbeta_score()
metric_hinge()
metric_huber()
metric_iou()
metric_kl_divergence()
metric_log_cosh()
metric_log_cosh_error()
metric_mean()
metric_mean_absolute_error()
metric_mean_absolute_percentage_error()
metric_mean_iou()
metric_mean_squared_error()
metric_mean_squared_logarithmic_error()
metric_mean_wrapper()
metric_one_hot_iou()
metric_one_hot_mean_iou()
metric_poisson()
metric_precision()
metric_r2_score()
metric_recall()
metric_recall_at_precision()
metric_root_mean_squared_error()
metric_sensitivity_at_specificity()
metric_sparse_categorical_accuracy()
metric_sparse_categorical_crossentropy()
metric_sparse_top_k_categorical_accuracy()
metric_specificity_at_sensitivity()
metric_squared_hinge()
metric_sum()
metric_top_k_categorical_accuracy()
metric_true_negatives()
metric_true_positives()

metric_r2_score *Computes R2 score.*

Description

Formula:

```
sum_squares_residuals <- sum((y_true - y_pred) ** 2)
sum_squares <- sum((y_true - mean(y_true)) ** 2)
R2 <- 1 - sum_squares_residuals / sum_squares
```

This is also called the coefficient of determination.

It indicates how close the fitted regression line is to ground-truth data.

- The highest score possible is 1.0. It indicates that the predictors perfectly accounts for variation in the target.
- A score of 0.0 indicates that the predictors do not account for variation in the target.
- It can also be negative if the model is worse than random.

This metric can also compute the "Adjusted R2" score.

Usage

```
metric_r2_score(
  ...,
  class_aggregation = "uniform_average",
  num_regressors = 0L,
  name = "r2_score",
  dtype = NULL
)
```

Arguments

`...`	For forward/backward compatability.
`class_aggregation`	Specifies how to aggregate scores corresponding to different output classes (or target dimensions), i.e. different dimensions on the last axis of the predictions. Equivalent to `multioutput` argument in Scikit-Learn. Should be one of `NULL` (no aggregation), `"uniform_average"`, `"variance_weighted_average"`.
`num_regressors`	Number of independent regressors used ("Adjusted R2" score). 0 is the standard R2 score. Defaults to `0`.
`name`	Optional. string name of the metric instance.
`dtype`	Optional. data type of the metric result.

Value

a `Metric` instance is returned. The `Metric` instance can be passed directly to `compile(metrics = )`, or used as a standalone object. See `?Metric` for example usage.

Examples

```
y_true <- rbind(1, 4, 3)
y_pred <- rbind(2, 4, 4)
metric <- metric_r2_score()
metric$update_state(y_true, y_pred)
metric$result()

## tf.Tensor(0.57142854, shape=(), dtype=float32)
```

See Also

Other regression metrics:
`metric_cosine_similarity()`
`metric_log_cosh_error()`
`metric_mean_absolute_error()`
`metric_mean_absolute_percentage_error()`
`metric_mean_squared_error()`
`metric_mean_squared_logarithmic_error()`
`metric_root_mean_squared_error()`

Other metrics:
`Metric()`
`custom_metric()`
`metric_auc()`
`metric_binary_accuracy()`
`metric_binary_crossentropy()`
`metric_binary_focal_crossentropy()`
`metric_binary_iou()`
`metric_categorical_accuracy()`
`metric_categorical_crossentropy()`
`metric_categorical_focal_crossentropy()`
`metric_categorical_hinge()`
`metric_cosine_similarity()`
`metric_f1_score()`
`metric_false_negatives()`
`metric_false_positives()`
`metric_fbeta_score()`
`metric_hinge()`
`metric_huber()`
`metric_iou()`
`metric_kl_divergence()`
`metric_log_cosh()`
`metric_log_cosh_error()`
`metric_mean()`
`metric_mean_absolute_error()`
`metric_mean_absolute_percentage_error()`
`metric_mean_iou()`

metric_mean_squared_error()
metric_mean_squared_logarithmic_error()
metric_mean_wrapper()
metric_one_hot_iou()
metric_one_hot_mean_iou()
metric_poisson()
metric_precision()
metric_precision_at_recall()
metric_recall()
metric_recall_at_precision()
metric_root_mean_squared_error()
metric_sensitivity_at_specificity()
metric_sparse_categorical_accuracy()
metric_sparse_categorical_crossentropy()
metric_sparse_top_k_categorical_accuracy()
metric_specificity_at_sensitivity()
metric_squared_hinge()
metric_sum()
metric_top_k_categorical_accuracy()
metric_true_negatives()
metric_true_positives()

---

metric_recall *Computes the recall of the predictions with respect to the labels.*

Description

This metric creates two local variables, `true_positives` and `false_negatives`, that are used to compute the recall. This value is ultimately returned as `recall`, an idempotent operation that simply divides `true_positives` by the sum of `true_positives` and `false_negatives`.

If `sample_weight` is `NULL`, weights default to 1. Use `sample_weight` of 0 to mask values.

If `top_k` is set, recall will be computed as how often on average a class among the labels of a batch entry is in the top-k predictions.

If `class_id` is specified, we calculate recall by considering only the entries in the batch for which `class_id` is in the label, and computing the fraction of them for which `class_id` is above the threshold and/or in the top-k predictions.

Usage

```
metric_recall(
  ...,
  thresholds = NULL,
  top_k = NULL,
  class_id = NULL,
  name = NULL,
  dtype = NULL
)
```

Arguments

`...`	For forward/backward compatability.
`thresholds`	(Optional) A float value, or a Python list of float threshold values in `[0, 1]`. A threshold is compared with prediction values to determine the truth value of predictions (i.e., above the threshold is `TRUE`, below is `FALSE`). If used with a loss function that sets `from_logits=TRUE` (i.e. no sigmoid applied to predictions), `thresholds` should be set to 0. One metric value is generated for each threshold value. If neither `thresholds` nor `top_k` are set, the default is to calculate recall with `thresholds=0.5`.
`top_k`	(Optional) Unset by default. An int value specifying the top-k predictions to consider when calculating recall.
`class_id`	(Optional) Integer class ID for which we want binary metrics. This must be in the half-open interval `[0, num_classes)`, where `num_classes` is the last dimension of predictions.
`name`	(Optional) string name of the metric instance.
`dtype`	(Optional) data type of the metric result.

Value

a `Metric` instance is returned. The `Metric` instance can be passed directly to `compile(metrics = )`, or used as a standalone object. See `?Metric` for example usage.

Usage

Standalone usage:

```
m <- metric_recall()
m$update_state(c(0, 1, 1, 1),
               c(1, 0, 1, 1))
m$result()

## tf.Tensor(0.6666667, shape=(), dtype=float32)


m$reset_state()
m$update_state(c(0, 1, 1, 1),
               c(1, 0, 1, 1),
               sample_weight = c(0, 0, 1, 0))
m$result()

## tf.Tensor(1.0, shape=(), dtype=float32)
```

Usage with `compile()` API:

```
model |> compile(
  optimizer = 'sgd',
  loss = 'binary_crossentropy',
  metrics = list(metric_recall())
)
```

Usage with a loss with `from_logits=TRUE`:

```
model |> compile(
  optimizer = 'adam',
  loss = loss_binary_crossentropy(from_logits = TRUE),
  metrics = list(metric_recall(thresholds = 0))
)
```

See Also

- https://keras.io/api/metrics/classification_metrics#recall-class

Other confusion metrics:
metric_auc()
metric_false_negatives()
metric_false_positives()
metric_precision()
metric_precision_at_recall()
metric_recall_at_precision()
metric_sensitivity_at_specificity()
metric_specificity_at_sensitivity()
metric_true_negatives()
metric_true_positives()

Other metrics:
Metric()
custom_metric()
metric_auc()
metric_binary_accuracy()
metric_binary_crossentropy()
metric_binary_focal_crossentropy()
metric_binary_iou()
metric_categorical_accuracy()
metric_categorical_crossentropy()
metric_categorical_focal_crossentropy()
metric_categorical_hinge()
metric_cosine_similarity()
metric_f1_score()
metric_false_negatives()
metric_false_positives()
metric_fbeta_score()
metric_hinge()
metric_huber()

metric_iou()
metric_kl_divergence()
metric_log_cosh()
metric_log_cosh_error()
metric_mean()
metric_mean_absolute_error()
metric_mean_absolute_percentage_error()
metric_mean_iou()
metric_mean_squared_error()
metric_mean_squared_logarithmic_error()
metric_mean_wrapper()
metric_one_hot_iou()
metric_one_hot_mean_iou()
metric_poisson()
metric_precision()
metric_precision_at_recall()
metric_r2_score()
metric_recall_at_precision()
metric_root_mean_squared_error()
metric_sensitivity_at_specificity()
metric_sparse_categorical_accuracy()
metric_sparse_categorical_crossentropy()
metric_sparse_top_k_categorical_accuracy()
metric_specificity_at_sensitivity()
metric_squared_hinge()
metric_sum()
metric_top_k_categorical_accuracy()
metric_true_negatives()
metric_true_positives()

---

metric_recall_at_precision
Computes best recall where precision is >= specified value.

Description

For a given score-label-distribution the required precision might not be achievable, in this case 0.0 is returned as recall.

This metric creates four local variables, `true_positives`, `true_negatives`, `false_positives` and `false_negatives` that are used to compute the recall at the given precision. The threshold for the given precision value is computed and used to evaluate the corresponding recall.

If `sample_weight` is `NULL`, weights default to 1. Use `sample_weight` of 0 to mask values.

If `class_id` is specified, we calculate precision by considering only the entries in the batch for which `class_id` is above the threshold predictions, and computing the fraction of them for which `class_id` is indeed a correct label.

Usage

```
metric_recall_at_precision(
  ...,
  precision,
  num_thresholds = 200L,
  class_id = NULL,
  name = NULL,
  dtype = NULL
)
```

Arguments

`...`	For forward/backward compatability.
`precision`	A scalar value in range `[0, 1]`.
`num_thresholds`	(Optional) Defaults to 200. The number of thresholds to use for matching the given precision.
`class_id`	(Optional) Integer class ID for which we want binary metrics. This must be in the half-open interval `[0, num_classes)`, where `num_classes` is the last dimension of predictions.
`name`	(Optional) string name of the metric instance.
`dtype`	(Optional) data type of the metric result.

Value

a `Metric` instance is returned. The `Metric` instance can be passed directly to `compile(metrics = )`, or used as a standalone object. See `?Metric` for example usage.

Usage

Standalone usage:

```
m <- metric_recall_at_precision(precision = 0.8)
m$update_state(c(0,   0,   1,   1),
               c(0, 0.5, 0.3, 0.9))
m$result()

## tf.Tensor(0.5, shape=(), dtype=float32)


m$reset_state()
m$update_state(c(0,   0,   1,   1),
               c(0, 0.5, 0.3, 0.9),
               sample_weight = c(1, 0, 0, 1))
m$result()

## tf.Tensor(1.0, shape=(), dtype=float32)
```

Usage with `compile()` API:

```
model |> compile(
  optimizer = 'sgd',
  loss = 'binary_crossentropy',
  metrics = list(metric_recall_at_precision(precision = 0.8))
)
```

See Also

Other confusion metrics:
`metric_auc()`
`metric_false_negatives()`
`metric_false_positives()`
`metric_precision()`
`metric_precision_at_recall()`
`metric_recall()`
`metric_sensitivity_at_specificity()`
`metric_specificity_at_sensitivity()`
`metric_true_negatives()`
`metric_true_positives()`

Other metrics:
`Metric()`
`custom_metric()`
`metric_auc()`
`metric_binary_accuracy()`
`metric_binary_crossentropy()`
`metric_binary_focal_crossentropy()`
`metric_binary_iou()`
`metric_categorical_accuracy()`
`metric_categorical_crossentropy()`
`metric_categorical_focal_crossentropy()`
`metric_categorical_hinge()`
`metric_cosine_similarity()`
`metric_f1_score()`
`metric_false_negatives()`
`metric_false_positives()`
`metric_fbeta_score()`
`metric_hinge()`
`metric_huber()`
`metric_iou()`
`metric_kl_divergence()`
`metric_log_cosh()`
`metric_log_cosh_error()`
`metric_mean()`
`metric_mean_absolute_error()`
`metric_mean_absolute_percentage_error()`

metric_mean_iou()
metric_mean_squared_error()
metric_mean_squared_logarithmic_error()
metric_mean_wrapper()
metric_one_hot_iou()
metric_one_hot_mean_iou()
metric_poisson()
metric_precision()
metric_precision_at_recall()
metric_r2_score()
metric_recall()
metric_root_mean_squared_error()
metric_sensitivity_at_specificity()
metric_sparse_categorical_accuracy()
metric_sparse_categorical_crossentropy()
metric_sparse_top_k_categorical_accuracy()
metric_specificity_at_sensitivity()
metric_squared_hinge()
metric_sum()
metric_top_k_categorical_accuracy()
metric_true_negatives()
metric_true_positives()

---

metric_root_mean_squared_error   *Computes root mean squared error metric between* y_true *and* y_pred*.*

Description

Formula:

```
loss <- sqrt(mean((y_pred - y_true) ^ 2))
```

Usage

```
metric_root_mean_squared_error(
  ...,
  name = "root_mean_squared_error",
  dtype = NULL
)
```

Arguments

...	For forward/backward compatability.
name	(Optional) string name of the metric instance.
dtype	(Optional) data type of the metric result.

Value

a `Metric` instance is returned. The `Metric` instance can be passed directly to `compile(metrics = )`, or used as a standalone object. See `?Metric` for example usage.

Examples

Standalone usage:

```
m <- metric_root_mean_squared_error()
m$update_state(rbind(c(0, 1), c(0, 0)), rbind(c(1, 1), c(0, 0)))
m$result()
```

```
## tf.Tensor(0.5, shape=(), dtype=float32)
```

```
m$reset_state()
m$update_state(rbind(c(0, 1), c(0, 0)), rbind(c(1, 1), c(0, 0)),
               sample_weight = c(1, 0))
m$result()
```

```
## tf.Tensor(0.70710677, shape=(), dtype=float32)
```

Usage with `compile()` API:

```
model %>% compile(
  optimizer = 'sgd',
  loss = 'mse',
  metrics = list(metric_root_mean_squared_error()))
```

See Also

- `https://keras.io/api/metrics/regression_metrics#rootmeansquarederror-class`

Other regression metrics:
`metric_cosine_similarity()`
`metric_log_cosh_error()`
`metric_mean_absolute_error()`
`metric_mean_absolute_percentage_error()`
`metric_mean_squared_error()`
`metric_mean_squared_logarithmic_error()`
`metric_r2_score()`

Other metrics:
`Metric()`
`custom_metric()`
`metric_auc()`
`metric_binary_accuracy()`

metric_binary_crossentropy()
metric_binary_focal_crossentropy()
metric_binary_iou()
metric_categorical_accuracy()
metric_categorical_crossentropy()
metric_categorical_focal_crossentropy()
metric_categorical_hinge()
metric_cosine_similarity()
metric_f1_score()
metric_false_negatives()
metric_false_positives()
metric_fbeta_score()
metric_hinge()
metric_huber()
metric_iou()
metric_kl_divergence()
metric_log_cosh()
metric_log_cosh_error()
metric_mean()
metric_mean_absolute_error()
metric_mean_absolute_percentage_error()
metric_mean_iou()
metric_mean_squared_error()
metric_mean_squared_logarithmic_error()
metric_mean_wrapper()
metric_one_hot_iou()
metric_one_hot_mean_iou()
metric_poisson()
metric_precision()
metric_precision_at_recall()
metric_r2_score()
metric_recall()
metric_recall_at_precision()
metric_sensitivity_at_specificity()
metric_sparse_categorical_accuracy()
metric_sparse_categorical_crossentropy()
metric_sparse_top_k_categorical_accuracy()
metric_specificity_at_sensitivity()
metric_squared_hinge()
metric_sum()
metric_top_k_categorical_accuracy()
metric_true_negatives()
metric_true_positives()

metric_sensitivity_at_specificity

Computes best sensitivity where specificity is >= specified value.

Description

`Sensitivity` measures the proportion of actual positives that are correctly identified as such `(tp / (tp + fn))`. `Specificity` measures the proportion of actual negatives that are correctly identified as such `(tn / (tn + fp))`.

This metric creates four local variables, `true_positives`, `true_negatives`, `false_positives` and `false_negatives` that are used to compute the sensitivity at the given specificity. The threshold for the given specificity value is computed and used to evaluate the corresponding sensitivity.

If `sample_weight` is `NULL`, weights default to 1. Use `sample_weight` of 0 to mask values.

If `class_id` is specified, we calculate precision by considering only the entries in the batch for which `class_id` is above the threshold predictions, and computing the fraction of them for which `class_id` is indeed a correct label.

For additional information about specificity and sensitivity, see the following.

Usage

```
metric_sensitivity_at_specificity(
  ...,
  specificity,
  num_thresholds = 200L,
  class_id = NULL,
  name = NULL,
  dtype = NULL
)
```

Arguments

`...`	For forward/backward compatability.
`specificity`	A scalar value in range `[0, 1]`.
`num_thresholds`	(Optional) Defaults to 200. The number of thresholds to use for matching the given specificity.
`class_id`	(Optional) Integer class ID for which we want binary metrics. This must be in the half-open interval `[0, num_classes)`, where `num_classes` is the last dimension of predictions.
`name`	(Optional) string name of the metric instance.
`dtype`	(Optional) data type of the metric result.

Value

a `Metric` instance is returned. The `Metric` instance can be passed directly to `compile(metrics = )`, or used as a standalone object. See `?Metric` for example usage.

Usage

Standalone usage:

```
m <- metric_sensitivity_at_specificity(specificity = 0.5)
m$update_state(c(0,   0,   0,   1,   1),
               c(0, 0.3, 0.8, 0.3, 0.8))
m$result()

## tf.Tensor(0.5, shape=(), dtype=float32)


m$reset_state()
m$update_state(c(0,   0,   0,   1,   1),
               c(0, 0.3, 0.8, 0.3, 0.8),
               sample_weight = c(1, 1, 2, 2, 1))
m$result()

## tf.Tensor(0.33333334, shape=(), dtype=float32)
```

Usage with `compile()` API:

```
model |> compile(
  optimizer = 'sgd',
  loss = 'binary_crossentropy',
  metrics = list(metric_sensitivity_at_specificity())
)
```

See Also

- https://keras.io/api/metrics/classification_metrics#sensitivityatspecificity-class

Other confusion metrics:
`metric_auc()`
`metric_false_negatives()`
`metric_false_positives()`
`metric_precision()`
`metric_precision_at_recall()`
`metric_recall()`
`metric_recall_at_precision()`
`metric_specificity_at_sensitivity()`
`metric_true_negatives()`
`metric_true_positives()`

Other metrics:
`Metric()`
`custom_metric()`
`metric_auc()`

metric_binary_accuracy()
metric_binary_crossentropy()
metric_binary_focal_crossentropy()
metric_binary_iou()
metric_categorical_accuracy()
metric_categorical_crossentropy()
metric_categorical_focal_crossentropy()
metric_categorical_hinge()
metric_cosine_similarity()
metric_f1_score()
metric_false_negatives()
metric_false_positives()
metric_fbeta_score()
metric_hinge()
metric_huber()
metric_iou()
metric_kl_divergence()
metric_log_cosh()
metric_log_cosh_error()
metric_mean()
metric_mean_absolute_error()
metric_mean_absolute_percentage_error()
metric_mean_iou()
metric_mean_squared_error()
metric_mean_squared_logarithmic_error()
metric_mean_wrapper()
metric_one_hot_iou()
metric_one_hot_mean_iou()
metric_poisson()
metric_precision()
metric_precision_at_recall()
metric_r2_score()
metric_recall()
metric_recall_at_precision()
metric_root_mean_squared_error()
metric_sparse_categorical_accuracy()
metric_sparse_categorical_crossentropy()
metric_sparse_top_k_categorical_accuracy()
metric_specificity_at_sensitivity()
metric_squared_hinge()
metric_sum()
metric_top_k_categorical_accuracy()
metric_true_negatives()
metric_true_positives()

metric_sparse_categorical_accuracy

Calculates how often predictions match integer labels.

Description

```
acc <- sample_weight %*% (y_true == which.max(y_pred))
```

You can provide logits of classes as `y_pred`, since argmax of logits and probabilities are same.

This metric creates two local variables, `total` and `count` that are used to compute the frequency with which `y_pred` matches `y_true`. This frequency is ultimately returned as `sparse categorical accuracy`: an idempotent operation that simply divides `total` by `count`.

If `sample_weight` is `NULL`, weights default to 1. Use `sample_weight` of 0 to mask values.

Usage

```
metric_sparse_categorical_accuracy(
  y_true,
  y_pred,
  ...,
  name = "sparse_categorical_accuracy",
  dtype = NULL
)
```

Arguments

`y_true`	Tensor of true targets.
`y_pred`	Tensor of predicted targets.
`...`	For forward/backward compatability.
`name`	(Optional) string name of the metric instance.
`dtype`	(Optional) data type of the metric result.

Value

If `y_true` and `y_pred` are missing, a `Metric` instance is returned. The `Metric` instance that can be passed directly to `compile(metrics = )`, or used as a standalone object. See `?Metric` for example usage. If `y_true` and `y_pred` are provided, then a tensor with the computed value is returned.

Usage

Standalone usage:

```
m <- metric_sparse_categorical_accuracy()
m$update_state(rbind(2L, 1L), rbind(c(0.1, 0.6, 0.3), c(0.05, 0.95, 0)))
m$result()

## tf.Tensor(0.5, shape=(), dtype=float32)
```

```
m$reset_state()
m$update_state(rbind(2L, 1L), rbind(c(0.1, 0.6, 0.3), c(0.05, 0.95, 0)),
               sample_weight = c(0.7, 0.3))
m$result()

## tf.Tensor(0.3, shape=(), dtype=float32)
```

Usage with `compile()` API:

```
model %>% compile(optimizer = 'sgd',
                  loss = 'sparse_categorical_crossentropy',
                  metrics = list(metric_sparse_categorical_accuracy()))
```

See Also

- https://keras.io/api/metrics/accuracy_metrics#sparsecategoricalaccuracy-class

Other accuracy metrics:
`metric_binary_accuracy()`
`metric_categorical_accuracy()`
`metric_sparse_top_k_categorical_accuracy()`
`metric_top_k_categorical_accuracy()`

Other metrics:
`Metric()`
`custom_metric()`
`metric_auc()`
`metric_binary_accuracy()`
`metric_binary_crossentropy()`
`metric_binary_focal_crossentropy()`
`metric_binary_iou()`
`metric_categorical_accuracy()`
`metric_categorical_crossentropy()`
`metric_categorical_focal_crossentropy()`
`metric_categorical_hinge()`
`metric_cosine_similarity()`
`metric_f1_score()`
`metric_false_negatives()`
`metric_false_positives()`
`metric_fbeta_score()`
`metric_hinge()`
`metric_huber()`
`metric_iou()`
`metric_kl_divergence()`
`metric_log_cosh()`
`metric_log_cosh_error()`
`metric_mean()`
`metric_mean_absolute_error()`

metric_mean_absolute_percentage_error()
metric_mean_iou()
metric_mean_squared_error()
metric_mean_squared_logarithmic_error()
metric_mean_wrapper()
metric_one_hot_iou()
metric_one_hot_mean_iou()
metric_poisson()
metric_precision()
metric_precision_at_recall()
metric_r2_score()
metric_recall()
metric_recall_at_precision()
metric_root_mean_squared_error()
metric_sensitivity_at_specificity()
metric_sparse_categorical_crossentropy()
metric_sparse_top_k_categorical_accuracy()
metric_specificity_at_sensitivity()
metric_squared_hinge()
metric_sum()
metric_top_k_categorical_accuracy()
metric_true_negatives()
metric_true_positives()

---

metric_sparse_categorical_crossentropy

Computes the crossentropy metric between the labels and predictions.

Description

Use this crossentropy metric when there are two or more label classes. It expects labels to be provided as integers. If you want to provide labels that are one-hot encoded, please use the `metric_categorical_crossentropy` metric instead.

There should be `num_classes` floating point values per feature for `y_pred` and a single floating point value per feature for `y_true`.

Usage

```
metric_sparse_categorical_crossentropy(
  y_true,
  y_pred,
  from_logits = FALSE,
  ignore_class = NULL,
  axis = -1L,
  ...,
  name = "sparse_categorical_crossentropy",
  dtype = NULL
)
```

Arguments

`y_true`	Ground truth values.
`y_pred`	The predicted values.
`from_logits`	(Optional) Whether output is expected to be a logits tensor. By default, we consider that output encodes a probability distribution.
`ignore_class`	Optional integer. The ID of a class to be ignored during loss computation. This is useful, for example, in segmentation problems featuring a "void" class (commonly -1 or 255) in segmentation maps. By default (`ignore_class=NULL`), all classes are considered.
`axis`	(Optional) Defaults to `-1`. The dimension along which entropy is computed.
`...`	For forward/backward compatability.
`name`	(Optional) string name of the metric instance.
`dtype`	(Optional) data type of the metric result.

Value

If `y_true` and `y_pred` are missing, a `Metric` instance is returned. The `Metric` instance that can be passed directly to `compile(metrics = )`, or used as a standalone object. See `?Metric` for example usage. If `y_true` and `y_pred` are provided, then a tensor with the computed value is returned.

Examples

Standalone usage:

```
m <- metric_sparse_categorical_crossentropy()
m$update_state(c(1, 2),
               rbind(c(0.05, 0.95, 0), c(0.1, 0.8, 0.1)))
m$result()

## tf.Tensor(1.1769392, shape=(), dtype=float32)


m$reset_state()
m$update_state(c(1, 2),
               rbind(c(0.05, 0.95, 0), c(0.1, 0.8, 0.1)),
               sample_weight = c(0.3, 0.7))
m$result()

## tf.Tensor(1.6271976, shape=(), dtype=float32)


# 1.6271976
```

Usage with `compile()` API:

```
model %>% compile(
    optimizer = 'sgd',
    loss = 'mse',
    metrics = list(metric_sparse_categorical_crossentropy()))
```

See Also

- https://keras.io/api/metrics/probabilistic_metrics#sparsecategoricalcrossentropy-class

Other losses:
`Loss()`
`loss_binary_crossentropy()`
`loss_binary_focal_crossentropy()`
`loss_categorical_crossentropy()`
`loss_categorical_focal_crossentropy()`
`loss_categorical_hinge()`
`loss_cosine_similarity()`
`loss_dice()`
`loss_hinge()`
`loss_huber()`
`loss_kl_divergence()`
`loss_log_cosh()`
`loss_mean_absolute_error()`
`loss_mean_absolute_percentage_error()`
`loss_mean_squared_error()`
`loss_mean_squared_logarithmic_error()`
`loss_poisson()`
`loss_sparse_categorical_crossentropy()`
`loss_squared_hinge()`
`metric_binary_crossentropy()`
`metric_binary_focal_crossentropy()`
`metric_categorical_crossentropy()`
`metric_categorical_focal_crossentropy()`
`metric_categorical_hinge()`
`metric_hinge()`
`metric_huber()`
`metric_kl_divergence()`
`metric_log_cosh()`
`metric_mean_absolute_error()`
`metric_mean_absolute_percentage_error()`
`metric_mean_squared_error()`
`metric_mean_squared_logarithmic_error()`
`metric_poisson()`
`metric_squared_hinge()`

Other metrics:
`Metric()`

custom_metric()
metric_auc()
metric_binary_accuracy()
metric_binary_crossentropy()
metric_binary_focal_crossentropy()
metric_binary_iou()
metric_categorical_accuracy()
metric_categorical_crossentropy()
metric_categorical_focal_crossentropy()
metric_categorical_hinge()
metric_cosine_similarity()
metric_f1_score()
metric_false_negatives()
metric_false_positives()
metric_fbeta_score()
metric_hinge()
metric_huber()
metric_iou()
metric_kl_divergence()
metric_log_cosh()
metric_log_cosh_error()
metric_mean()
metric_mean_absolute_error()
metric_mean_absolute_percentage_error()
metric_mean_iou()
metric_mean_squared_error()
metric_mean_squared_logarithmic_error()
metric_mean_wrapper()
metric_one_hot_iou()
metric_one_hot_mean_iou()
metric_poisson()
metric_precision()
metric_precision_at_recall()
metric_r2_score()
metric_recall()
metric_recall_at_precision()
metric_root_mean_squared_error()
metric_sensitivity_at_specificity()
metric_sparse_categorical_accuracy()
metric_sparse_top_k_categorical_accuracy()
metric_specificity_at_sensitivity()
metric_squared_hinge()
metric_sum()
metric_top_k_categorical_accuracy()
metric_true_negatives()
metric_true_positives()

Other probabilistic metrics:

metric_binary_crossentropy()
metric_categorical_crossentropy()
metric_kl_divergence()
metric_poisson()

metric_sparse_top_k_categorical_accuracy

Computes how often integer targets are in the top K predictions.

Description

Computes how often integer targets are in the top K predictions.

Usage

```
metric_sparse_top_k_categorical_accuracy(
  y_true,
  y_pred,
  k = 5L,
  ...,
  name = "sparse_top_k_categorical_accuracy",
  dtype = NULL
)
```

Arguments

y_true	Tensor of true targets.
y_pred	Tensor of predicted targets.
k	(Optional) Number of top elements to look at for computing accuracy. Defaults to 5.
...	For forward/backward compatability.
name	(Optional) string name of the metric instance.
dtype	(Optional) data type of the metric result.

Value

If y_true and y_pred are missing, a Metric instance is returned. The Metric instance that can be passed directly to compile(metrics =), or used as a standalone object. See ?Metric for example usage. If y_true and y_pred are provided, then a tensor with the computed value is returned.

Usage

Standalone usage:

```
m <- metric_sparse_top_k_categorical_accuracy(k = 1L)
m$update_state(
  rbind(2, 1),
  op_array(rbind(c(0.1, 0.9, 0.8), c(0.05, 0.95, 0)), dtype = "float32")
)
m$result()
```

```
## tf.Tensor(0.5, shape=(), dtype=float32)
```

```
m$reset_state()
m$update_state(
  rbind(2, 1),
  op_array(rbind(c(0.1, 0.9, 0.8), c(0.05, 0.95, 0)), dtype = "float32"),
  sample_weight = c(0.7, 0.3)
)
m$result()
```

```
## tf.Tensor(0.3, shape=(), dtype=float32)
```

Usage with `compile()` API:

```
model %>% compile(optimizer = 'sgd',
                  loss = 'sparse_categorical_crossentropy',
                  metrics = list(metric_sparse_top_k_categorical_accuracy()))
```

See Also

- https://keras.io/api/metrics/accuracy_metrics#sparsetopkcategoricalaccuracy-class

Other accuracy metrics:
`metric_binary_accuracy()`
`metric_categorical_accuracy()`
`metric_sparse_categorical_accuracy()`
`metric_top_k_categorical_accuracy()`

Other metrics:
`Metric()`
`custom_metric()`
`metric_auc()`
`metric_binary_accuracy()`
`metric_binary_crossentropy()`
`metric_binary_focal_crossentropy()`
`metric_binary_iou()`

metric_categorical_accuracy()
metric_categorical_crossentropy()
metric_categorical_focal_crossentropy()
metric_categorical_hinge()
metric_cosine_similarity()
metric_f1_score()
metric_false_negatives()
metric_false_positives()
metric_fbeta_score()
metric_hinge()
metric_huber()
metric_iou()
metric_kl_divergence()
metric_log_cosh()
metric_log_cosh_error()
metric_mean()
metric_mean_absolute_error()
metric_mean_absolute_percentage_error()
metric_mean_iou()
metric_mean_squared_error()
metric_mean_squared_logarithmic_error()
metric_mean_wrapper()
metric_one_hot_iou()
metric_one_hot_mean_iou()
metric_poisson()
metric_precision()
metric_precision_at_recall()
metric_r2_score()
metric_recall()
metric_recall_at_precision()
metric_root_mean_squared_error()
metric_sensitivity_at_specificity()
metric_sparse_categorical_accuracy()
metric_sparse_categorical_crossentropy()
metric_specificity_at_sensitivity()
metric_squared_hinge()
metric_sum()
metric_top_k_categorical_accuracy()
metric_true_negatives()
metric_true_positives()

metric_specificity_at_sensitivity

Computes best specificity where sensitivity is >= specified value.

Description

Sensitivity measures the proportion of actual positives that are correctly identified as such (tp / (tp + fn)). Specificity measures the proportion of actual negatives that are correctly identified as such (tn / (tn + fp)).

This metric creates four local variables, true_positives, true_negatives, false_positives and false_negatives that are used to compute the specificity at the given sensitivity. The threshold for the given sensitivity value is computed and used to evaluate the corresponding specificity.

If sample_weight is NULL, weights default to 1. Use sample_weight of 0 to mask values.

If class_id is specified, we calculate precision by considering only the entries in the batch for which class_id is above the threshold predictions, and computing the fraction of them for which class_id is indeed a correct label.

For additional information about specificity and sensitivity, see the following.

Usage

```
metric_specificity_at_sensitivity(
  ...,
  sensitivity,
  num_thresholds = 200L,
  class_id = NULL,
  name = NULL,
  dtype = NULL
)
```

Arguments

...	For forward/backward compatability.
sensitivity	A scalar value in range [0, 1].
num_thresholds	(Optional) Defaults to 200. The number of thresholds to use for matching the given sensitivity.
class_id	(Optional) Integer class ID for which we want binary metrics. This must be in the half-open interval [0, num_classes), where num_classes is the last dimension of predictions.
name	(Optional) string name of the metric instance.
dtype	(Optional) data type of the metric result.

Value

a Metric instance is returned. The Metric instance can be passed directly to compile(metrics =), or used as a standalone object. See ?Metric for example usage.

Usage

Standalone usage:

```
m <- metric_specificity_at_sensitivity(sensitivity = 0.5)
m$update_state(c(0,   0,   0,   1,   1),
               c(0, 0.3, 0.8, 0.3, 0.8))
m$result()

## tf.Tensor(0.6666667, shape=(), dtype=float32)


m$reset_state()
m$update_state(c(0,   0,   0,   1,   1),
               c(0, 0.3, 0.8, 0.3, 0.8),
               sample_weight = c(1, 1, 2, 2, 2))
m$result()

## tf.Tensor(0.5, shape=(), dtype=float32)
```

Usage with `compile()` API:

```
model |> compile(
  optimizer = 'sgd',
  loss = 'binary_crossentropy',
  metrics = list(metric_sensitivity_at_specificity())
)
```

See Also

- https://keras.io/api/metrics/classification_metrics#specificityatsensitivity-class

Other confusion metrics:
metric_auc()
metric_false_negatives()
metric_false_positives()
metric_precision()
metric_precision_at_recall()
metric_recall()
metric_recall_at_precision()
metric_sensitivity_at_specificity()
metric_true_negatives()
metric_true_positives()

Other metrics:
Metric()
custom_metric()
metric_auc()
metric_binary_accuracy()
metric_binary_crossentropy()
metric_binary_focal_crossentropy()
metric_binary_iou()

metric_categorical_accuracy()
metric_categorical_crossentropy()
metric_categorical_focal_crossentropy()
metric_categorical_hinge()
metric_cosine_similarity()
metric_f1_score()
metric_false_negatives()
metric_false_positives()
metric_fbeta_score()
metric_hinge()
metric_huber()
metric_iou()
metric_kl_divergence()
metric_log_cosh()
metric_log_cosh_error()
metric_mean()
metric_mean_absolute_error()
metric_mean_absolute_percentage_error()
metric_mean_iou()
metric_mean_squared_error()
metric_mean_squared_logarithmic_error()
metric_mean_wrapper()
metric_one_hot_iou()
metric_one_hot_mean_iou()
metric_poisson()
metric_precision()
metric_precision_at_recall()
metric_r2_score()
metric_recall()
metric_recall_at_precision()
metric_root_mean_squared_error()
metric_sensitivity_at_specificity()
metric_sparse_categorical_accuracy()
metric_sparse_categorical_crossentropy()
metric_sparse_top_k_categorical_accuracy()
metric_squared_hinge()
metric_sum()
metric_top_k_categorical_accuracy()
metric_true_negatives()
metric_true_positives()

metric_squared_hinge *Computes the hinge metric between* y_true *and* y_pred.

Description

Formula:

```
loss <- mean(square(maximum(1 - y_true * y_pred, 0)))
```

`y_true` values are expected to be -1 or 1. If binary (0 or 1) labels are provided we will convert them to -1 or 1.

Usage

```
metric_squared_hinge(y_true, y_pred, ..., name = "squared_hinge", dtype = NULL)
```

Arguments

`y_true`	The ground truth values. `y_true` values are expected to be -1 or 1. If binary (0 or 1) labels are provided we will convert them to -1 or 1 with shape = `[batch_size, d0, .. dN]`.
`y_pred`	The predicted values with shape = `[batch_size, d0, .. dN]`.
`...`	For forward/backward compatability.
`name`	(Optional) string name of the metric instance.
`dtype`	(Optional) data type of the metric result.

Value

If `y_true` and `y_pred` are missing, a `Metric` instance is returned. The `Metric` instance that can be passed directly to `compile(metrics = )`, or used as a standalone object. See `?Metric` for example usage. If `y_true` and `y_pred` are provided, then a tensor with the computed value is returned.

Usage

Standalone usage:

```
m <- metric_squared_hinge()
m$update_state(rbind(c(0, 1), c(0, 0)), rbind(c(0.6, 0.4), c(0.4, 0.6)))
m$result()
```

```
## tf.Tensor(1.86, shape=(), dtype=float32)
```

```
m$reset_state()
m$update_state(rbind(c(0, 1), c(0, 0)), rbind(c(0.6, 0.4), c(0.4, 0.6)),
               sample_weight = c(1, 0))
m$result()
```

```
## tf.Tensor(1.46, shape=(), dtype=float32)
```

See Also

- https://keras.io/api/metrics/hinge_metrics#squaredhinge-class

Other losses:
Loss()
loss_binary_crossentropy()
loss_binary_focal_crossentropy()
loss_categorical_crossentropy()
loss_categorical_focal_crossentropy()
loss_categorical_hinge()
loss_cosine_similarity()
loss_dice()
loss_hinge()
loss_huber()
loss_kl_divergence()
loss_log_cosh()
loss_mean_absolute_error()
loss_mean_absolute_percentage_error()
loss_mean_squared_error()
loss_mean_squared_logarithmic_error()
loss_poisson()
loss_sparse_categorical_crossentropy()
loss_squared_hinge()
metric_binary_crossentropy()
metric_binary_focal_crossentropy()
metric_categorical_crossentropy()
metric_categorical_focal_crossentropy()
metric_categorical_hinge()
metric_hinge()
metric_huber()
metric_kl_divergence()
metric_log_cosh()
metric_mean_absolute_error()
metric_mean_absolute_percentage_error()
metric_mean_squared_error()
metric_mean_squared_logarithmic_error()
metric_poisson()
metric_sparse_categorical_crossentropy()

Other metrics:
Metric()
custom_metric()
metric_auc()
metric_binary_accuracy()
metric_binary_crossentropy()
metric_binary_focal_crossentropy()
metric_binary_iou()
metric_categorical_accuracy()

metric_categorical_crossentropy()
metric_categorical_focal_crossentropy()
metric_categorical_hinge()
metric_cosine_similarity()
metric_f1_score()
metric_false_negatives()
metric_false_positives()
metric_fbeta_score()
metric_hinge()
metric_huber()
metric_iou()
metric_kl_divergence()
metric_log_cosh()
metric_log_cosh_error()
metric_mean()
metric_mean_absolute_error()
metric_mean_absolute_percentage_error()
metric_mean_iou()
metric_mean_squared_error()
metric_mean_squared_logarithmic_error()
metric_mean_wrapper()
metric_one_hot_iou()
metric_one_hot_mean_iou()
metric_poisson()
metric_precision()
metric_precision_at_recall()
metric_r2_score()
metric_recall()
metric_recall_at_precision()
metric_root_mean_squared_error()
metric_sensitivity_at_specificity()
metric_sparse_categorical_accuracy()
metric_sparse_categorical_crossentropy()
metric_sparse_top_k_categorical_accuracy()
metric_specificity_at_sensitivity()
metric_sum()
metric_top_k_categorical_accuracy()
metric_true_negatives()
metric_true_positives()

Other hinge metrics:
metric_categorical_hinge()
metric_hinge()

metric_sum *Compute the (weighted) sum of the given values.*

Description

For example, if `values` is `[1, 3, 5, 7]` then their sum is 16. If `sample_weight` was specified as `[1, 1, 0, 0]` then the sum would be 4.

This metric creates one variable, `total`. This is ultimately returned as the sum value.

Usage

```
metric_sum(..., name = "sum", dtype = NULL)
```

Arguments

`...`	For forward/backward compatability.
`name`	(Optional) string name of the metric instance.
`dtype`	(Optional) data type of the metric result.

Value

a `Metric` instance is returned. The `Metric` instance can be passed directly to `compile(metrics = )`, or used as a standalone object. See `?Metric` for example usage.

Examples

```
m <- metric_sum()
m$update_state(c(1, 3, 5, 7))
m$result()

## tf.Tensor(16.0, shape=(), dtype=float32)


m <- metric_sum()
m$update_state(c(1, 3, 5, 7), sample_weight = c(1, 1, 0, 0))
m$result()

## tf.Tensor(4.0, shape=(), dtype=float32)
```

See Also

Other reduction metrics:
`metric_mean()`
`metric_mean_wrapper()`

Other metrics:
`Metric()`
`custom_metric()`
`metric_auc()`
`metric_binary_accuracy()`

metric_binary_crossentropy()
metric_binary_focal_crossentropy()
metric_binary_iou()
metric_categorical_accuracy()
metric_categorical_crossentropy()
metric_categorical_focal_crossentropy()
metric_categorical_hinge()
metric_cosine_similarity()
metric_f1_score()
metric_false_negatives()
metric_false_positives()
metric_fbeta_score()
metric_hinge()
metric_huber()
metric_iou()
metric_kl_divergence()
metric_log_cosh()
metric_log_cosh_error()
metric_mean()
metric_mean_absolute_error()
metric_mean_absolute_percentage_error()
metric_mean_iou()
metric_mean_squared_error()
metric_mean_squared_logarithmic_error()
metric_mean_wrapper()
metric_one_hot_iou()
metric_one_hot_mean_iou()
metric_poisson()
metric_precision()
metric_precision_at_recall()
metric_r2_score()
metric_recall()
metric_recall_at_precision()
metric_root_mean_squared_error()
metric_sensitivity_at_specificity()
metric_sparse_categorical_accuracy()
metric_sparse_categorical_crossentropy()
metric_sparse_top_k_categorical_accuracy()
metric_specificity_at_sensitivity()
metric_squared_hinge()
metric_top_k_categorical_accuracy()
metric_true_negatives()
metric_true_positives()

metric_top_k_categorical_accuracy

Computes how often targets are in the top `K` *predictions.*

Description

Computes how often targets are in the top `K` predictions.

Usage

```
metric_top_k_categorical_accuracy(
  y_true,
  y_pred,
  k = 5L,
  ...,
  name = "top_k_categorical_accuracy",
  dtype = NULL
)
```

Arguments

`y_true`	Tensor of true targets.
`y_pred`	Tensor of predicted targets.
`k`	(Optional) Number of top elements to look at for computing accuracy. Defaults to 5.
`...`	For forward/backward compatability.
`name`	(Optional) string name of the metric instance.
`dtype`	(Optional) data type of the metric result.

Value

If `y_true` and `y_pred` are missing, a `Metric` instance is returned. The `Metric` instance that can be passed directly to `compile(metrics = )`, or used as a standalone object. See `?Metric` for example usage. If `y_true` and `y_pred` are provided, then a tensor with the computed value is returned.

Usage

Standalone usage:

```
m <- metric_top_k_categorical_accuracy(k = 1)
m$update_state(
  rbind(c(0, 0, 1), c(0, 1, 0)),
  op_array(rbind(c(0.1, 0.9, 0.8), c(0.05, 0.95, 0)), dtype = "float32")
)
m$result()

## tf.Tensor(0.5, shape=(), dtype=float32)
```

```
m$reset_state()
m$update_state(
  rbind(c(0, 0, 1), c(0, 1, 0)),
  op_array(rbind(c(0.1, 0.9, 0.8), c(0.05, 0.95, 0)), dtype = "float32"),
  sample_weight = c(0.7, 0.3))
m$result()

## tf.Tensor(0.3, shape=(), dtype=float32)
```

Usage with `compile()` API:

```
model.compile(optimizer = 'sgd',
              loss = 'categorical_crossentropy',
              metrics = list(metric_top_k_categorical_accuracy()))
```

See Also

- https://keras.io/api/metrics/accuracy_metrics#topkcategoricalaccuracy-class

Other accuracy metrics:
metric_binary_accuracy()
metric_categorical_accuracy()
metric_sparse_categorical_accuracy()
metric_sparse_top_k_categorical_accuracy()

Other metrics:
Metric()
custom_metric()
metric_auc()
metric_binary_accuracy()
metric_binary_crossentropy()
metric_binary_focal_crossentropy()
metric_binary_iou()
metric_categorical_accuracy()
metric_categorical_crossentropy()
metric_categorical_focal_crossentropy()
metric_categorical_hinge()
metric_cosine_similarity()
metric_f1_score()
metric_false_negatives()
metric_false_positives()
metric_fbeta_score()
metric_hinge()
metric_huber()
metric_iou()
metric_kl_divergence()
metric_log_cosh()
metric_log_cosh_error()

metric_mean()
metric_mean_absolute_error()
metric_mean_absolute_percentage_error()
metric_mean_iou()
metric_mean_squared_error()
metric_mean_squared_logarithmic_error()
metric_mean_wrapper()
metric_one_hot_iou()
metric_one_hot_mean_iou()
metric_poisson()
metric_precision()
metric_precision_at_recall()
metric_r2_score()
metric_recall()
metric_recall_at_precision()
metric_root_mean_squared_error()
metric_sensitivity_at_specificity()
metric_sparse_categorical_accuracy()
metric_sparse_categorical_crossentropy()
metric_sparse_top_k_categorical_accuracy()
metric_specificity_at_sensitivity()
metric_squared_hinge()
metric_sum()
metric_true_negatives()
metric_true_positives()

---

metric_true_negatives *Calculates the number of true negatives.*

Description

If sample_weight is given, calculates the sum of the weights of true negatives. This metric creates one local variable, accumulator that is used to keep track of the number of true negatives.

If sample_weight is NULL, weights default to 1. Use sample_weight of 0 to mask values.

Usage

```
metric_true_negatives(..., thresholds = NULL, name = NULL, dtype = NULL)
```

Arguments

...	For forward/backward compatability.
thresholds	(Optional) Defaults to 0.5. A float value, or a Python list of float threshold values in [0, 1]. A threshold is compared with prediction values to determine the truth value of predictions (i.e., above the threshold is TRUE, below is FALSE).

If used with a loss function that sets from_logits=TRUE (i.e. no sigmoid applied to predictions), thresholds should be set to 0. One metric value is generated for each threshold value.

name	(Optional) string name of the metric instance.
dtype	(Optional) data type of the metric result.

Value

a Metric instance is returned. The Metric instance can be passed directly to compile(metrics =), or used as a standalone object. See ?Metric for example usage.

Usage

Standalone usage:

```
m <- metric_true_negatives()
m$update_state(c(0, 1, 0, 0), c(1, 1, 0, 0))
m$result()

## tf.Tensor(2.0, shape=(), dtype=float32)


m$reset_state()
m$update_state(c(0, 1, 0, 0), c(1, 1, 0, 0), sample_weight = c(0, 0, 1, 0))
m$result()

## tf.Tensor(1.0, shape=(), dtype=float32)
```

See Also

- https://keras.io/api/metrics/classification_metrics#truenegatives-class

Other confusion metrics:
metric_auc()
metric_false_negatives()
metric_false_positives()
metric_precision()
metric_precision_at_recall()
metric_recall()
metric_recall_at_precision()
metric_sensitivity_at_specificity()
metric_specificity_at_sensitivity()
metric_true_positives()

Other metrics:
Metric()
custom_metric()

metric_auc()
metric_binary_accuracy()
metric_binary_crossentropy()
metric_binary_focal_crossentropy()
metric_binary_iou()
metric_categorical_accuracy()
metric_categorical_crossentropy()
metric_categorical_focal_crossentropy()
metric_categorical_hinge()
metric_cosine_similarity()
metric_f1_score()
metric_false_negatives()
metric_false_positives()
metric_fbeta_score()
metric_hinge()
metric_huber()
metric_iou()
metric_kl_divergence()
metric_log_cosh()
metric_log_cosh_error()
metric_mean()
metric_mean_absolute_error()
metric_mean_absolute_percentage_error()
metric_mean_iou()
metric_mean_squared_error()
metric_mean_squared_logarithmic_error()
metric_mean_wrapper()
metric_one_hot_iou()
metric_one_hot_mean_iou()
metric_poisson()
metric_precision()
metric_precision_at_recall()
metric_r2_score()
metric_recall()
metric_recall_at_precision()
metric_root_mean_squared_error()
metric_sensitivity_at_specificity()
metric_sparse_categorical_accuracy()
metric_sparse_categorical_crossentropy()
metric_sparse_top_k_categorical_accuracy()
metric_specificity_at_sensitivity()
metric_squared_hinge()
metric_sum()
metric_top_k_categorical_accuracy()
metric_true_positives()

metric_true_positives *Calculates the number of true positives.*

Description

If `sample_weight` is given, calculates the sum of the weights of true positives. This metric creates one local variable, `true_positives` that is used to keep track of the number of true positives.

If `sample_weight` is `NULL`, weights default to 1. Use `sample_weight` of 0 to mask values.

Usage

```
metric_true_positives(..., thresholds = NULL, name = NULL, dtype = NULL)
```

Arguments

`...`	For forward/backward compatability.
`thresholds`	(Optional) Defaults to `0.5`. A float value, or a Python list of float threshold values in `[0, 1]`. A threshold is compared with prediction values to determine the truth value of predictions (i.e., above the threshold is `TRUE`, below is `FALSE`). If used with a loss function that sets `from_logits=TRUE` (i.e. no sigmoid applied to predictions), `thresholds` should be set to 0. One metric value is generated for each threshold value.
`name`	(Optional) string name of the metric instance.
`dtype`	(Optional) data type of the metric result.

Value

a `Metric` instance is returned. The `Metric` instance can be passed directly to `compile(metrics = )`, or used as a standalone object. See `?Metric` for example usage.

Usage

Standalone usage:

```
m <- metric_true_positives()
m$update_state(c(0, 1, 1, 1), c(1, 0, 1, 1))
m$result()

## tf.Tensor(2.0, shape=(), dtype=float32)


m$reset_state()
m$update_state(c(0, 1, 1, 1), c(1, 0, 1, 1), sample_weight = c(0, 0, 1, 0))
m$result()

## tf.Tensor(1.0, shape=(), dtype=float32)
```

See Also

- https://keras.io/api/metrics/classification_metrics#truepositives-class

Other confusion metrics:
metric_auc()
metric_false_negatives()
metric_false_positives()
metric_precision()
metric_precision_at_recall()
metric_recall()
metric_recall_at_precision()
metric_sensitivity_at_specificity()
metric_specificity_at_sensitivity()
metric_true_negatives()

Other metrics:
Metric()
custom_metric()
metric_auc()
metric_binary_accuracy()
metric_binary_crossentropy()
metric_binary_focal_crossentropy()
metric_binary_iou()
metric_categorical_accuracy()
metric_categorical_crossentropy()
metric_categorical_focal_crossentropy()
metric_categorical_hinge()
metric_cosine_similarity()
metric_f1_score()
metric_false_negatives()
metric_false_positives()
metric_fbeta_score()
metric_hinge()
metric_huber()
metric_iou()
metric_kl_divergence()
metric_log_cosh()
metric_log_cosh_error()
metric_mean()
metric_mean_absolute_error()
metric_mean_absolute_percentage_error()
metric_mean_iou()
metric_mean_squared_error()
metric_mean_squared_logarithmic_error()
metric_mean_wrapper()
metric_one_hot_iou()
metric_one_hot_mean_iou()
metric_poisson()

metric_precision()
metric_precision_at_recall()
metric_r2_score()
metric_recall()
metric_recall_at_precision()
metric_root_mean_squared_error()
metric_sensitivity_at_specificity()
metric_sparse_categorical_accuracy()
metric_sparse_categorical_crossentropy()
metric_sparse_top_k_categorical_accuracy()
metric_specificity_at_sensitivity()
metric_squared_hinge()
metric_sum()
metric_top_k_categorical_accuracy()
metric_true_negatives()

Model *Subclass the base Keras* Model *Class*

Description

This is for advanced use cases where you need to subclass the base Model type, e.g., you want to override the train_step() method.

If you just want to create or define a keras model, prefer keras_model() or keras_model_sequential().

If you just want to encapsulate some custom logic and state, and don't need to customize training behavior (besides calling self$add_loss() in the call() method), prefer Layer().

Usage

```
Model(
  classname,
  initialize = NULL,
  call = NULL,
  train_step = NULL,
  predict_step = NULL,
  test_step = NULL,
  compute_loss = NULL,
  compute_metrics = NULL,
  ...,
  public = list(),
  private = list(),
  inherit = NULL,
  parent_env = parent.frame()
)
```

Arguments

`classname`	String, the name of the custom class. (Conventionally, CamelCase).
`initialize, call, train_step, predict_step, test_step, compute_loss, compute_metrics`	Optional methods that can be overridden.
`..., public`	Additional methods or public members of the custom class.
`private`	Named list of R objects (typically, functions) to include in instance private environments. `private` methods will have all the same symbols in scope as public methods (See section "Symbols in Scope"). Each instance will have it's own `private` environment. Any objects in `private` will be invisible from the Keras framework and the Python runtime.
`inherit`	What the custom class will subclass. By default, the base keras class.
`parent_env`	The R environment that all class methods will have as a grandparent.

Value

A model constructor function, which you can call to create an instance of the new model type.

Symbols in scope

All R function custom methods (public and private) will have the following symbols in scope:

- `self`: The custom class instance.
- `super`: The custom class superclass.
- `private`: An R environment specific to the class instance. Any objects assigned here are invisible to the Keras framework.
- `__class__` and `as.symbol(classname)`: the custom class type object.

See Also

`active_property()` (e.g., for a `metrics` property implemented as a function).

`normalize` *Normalizes an array.*

Description

If the input is an R array, an R array will be returned. If it's a backend tensor, a backend tensor will be returned.

Usage

```
normalize(x, axis = -1L, order = 2L)
```

Arguments

`x`	Array to normalize.
`axis`	axis along which to normalize.
`order`	Normalization order (e.g. `order=2` for L2 norm).

Value

A normalized copy of the array.

See Also

- `https://keras.io/api/utils/python_utils#normalize-function`

Other numerical utils:
`to_categorical()`

Other utils:
`audio_dataset_from_directory()`
`clear_session()`
`config_disable_interactive_logging()`
`config_disable_traceback_filtering()`
`config_enable_interactive_logging()`
`config_enable_traceback_filtering()`
`config_is_interactive_logging_enabled()`
`config_is_traceback_filtering_enabled()`
`get_file()`
`get_source_inputs()`
`image_array_save()`
`image_dataset_from_directory()`
`image_from_array()`
`image_load()`
`image_smart_resize()`
`image_to_array()`
`layer_feature_space()`
`pack_x_y_sample_weight()`
`pad_sequences()`
`set_random_seed()`
`split_dataset()`
`text_dataset_from_directory()`
`timeseries_dataset_from_array()`
`to_categorical()`
`unpack_x_y_sample_weight()`
`zip_lists()`

optimizer_adadelta *Optimizer that implements the Adadelta algorithm.*

Description

Adadelta optimization is a stochastic gradient descent method that is based on adaptive learning rate per dimension to address two drawbacks:

- The continual decay of learning rates throughout training.
- The need for a manually selected global learning rate.

Adadelta is a more robust extension of Adagrad that adapts learning rates based on a moving window of gradient updates, instead of accumulating all past gradients. This way, Adadelta continues learning even when many updates have been done. Compared to Adagrad, in the original version of Adadelta you don't have to set an initial learning rate. In this version, the initial learning rate can be set, as in most other Keras optimizers.

Usage

```
optimizer_adadelta(
  learning_rate = 0.001,
  rho = 0.95,
  epsilon = 1e-07,
  weight_decay = NULL,
  clipnorm = NULL,
  clipvalue = NULL,
  global_clipnorm = NULL,
  use_ema = FALSE,
  ema_momentum = 0.99,
  ema_overwrite_frequency = NULL,
  name = "adadelta",
  ...,
  loss_scale_factor = NULL,
  gradient_accumulation_steps = NULL
)
```

Arguments

`learning_rate`	A float, a `[LearningRateSchedule()]` instance, or a callable that takes no arguments and returns the actual value to use. The learning rate. Defaults to `0.001`. Note that `Adadelta` tends to benefit from higher initial learning rate values compared to other optimizers. To match the exact form in the original paper, use 1.0.
`rho`	A floating point value. The decay rate. Defaults to `0.95`.
`epsilon`	Small floating point value for maintaining numerical stability.
`weight_decay`	Float. If set, weight decay is applied.

`clipnorm`	Float. If set, the gradient of each weight is individually clipped so that its norm is no higher than this value.
`clipvalue`	Float. If set, the gradient of each weight is clipped to be no higher than this value.
`global_clipnorm`	Float. If set, the gradient of all weights is clipped so that their global norm is no higher than this value.
`use_ema`	Boolean, defaults to `FALSE`. If `TRUE`, exponential moving average (EMA) is applied. EMA consists of computing an exponential moving average of the weights of the model (as the weight values change after each training batch), and periodically overwriting the weights with their moving average.
`ema_momentum`	Float, defaults to 0.99. Only used if `use_ema = TRUE`. This is the momentum to use when computing the EMA of the model's weights: `new_average = ema_momentum * old_average + (1 - ema_momentum) * current_variable_value`.
`ema_overwrite_frequency`	Int or `NULL`, defaults to `NULL`. Only used if `use_ema = TRUE`. Every `ema_overwrite_frequency` steps of iterations, we overwrite the model variable by its moving average. If `NULL`, the optimizer does not overwrite model variables in the middle of training, and you need to explicitly overwrite the variables at the end of training by calling `optimizer$finalize_variable_values()` (which updates the model variables in-place). When using the built-in `fit()` training loop, this happens automatically after the last epoch, and you don't need to do anything.
`name`	String. The name to use for momentum accumulator weights created by the optimizer.
`...`	For forward/backward compatability.
`loss_scale_factor`	Float or `NULL`. If a float, the scale factor will be multiplied the loss before computing gradients, and the inverse of the scale factor will be multiplied by the gradients before updating variables. Useful for preventing underflow during mixed precision training. Alternately, `optimizer_loss_scale()` will automatically set a loss scale factor.
`gradient_accumulation_steps`	Int or `NULL`. If an int, model & optimizer variables will not be updated at every step; instead they will be updated every `gradient_accumulation_steps` steps, using the average value of the gradients since the last update. This is known as "gradient accumulation". This can be useful when your batch size is very small, in order to reduce gradient noise at each update step.

Value

an `Optimizer` instance

Reference

- Zeiler, 2012

See Also

- https://keras.io/api/optimizers/adadelta#adadelta-class

Other optimizers:
optimizer_adafactor()
optimizer_adagrad()
optimizer_adam()
optimizer_adam_w()
optimizer_adamax()
optimizer_ftrl()
optimizer_lion()
optimizer_loss_scale()
optimizer_nadam()
optimizer_rmsprop()
optimizer_sgd()

---

optimizer_adafactor *Optimizer that implements the Adafactor algorithm.*

Description

Adafactor is commonly used in NLP tasks, and has the advantage of taking less memory because it only saves partial information of previous gradients.

The default argument setup is based on the original paper (see reference). When gradients are of dimension > 2, Adafactor optimizer will delete the last 2 dimensions separately in its accumulator variables.

Usage

```
optimizer_adafactor(
  learning_rate = 0.001,
  beta_2_decay = -0.8,
  epsilon_1 = 1e-30,
  epsilon_2 = 0.001,
  clip_threshold = 1,
  relative_step = TRUE,
  weight_decay = NULL,
  clipnorm = NULL,
  clipvalue = NULL,
  global_clipnorm = NULL,
  use_ema = FALSE,
  ema_momentum = 0.99,
  ema_overwrite_frequency = NULL,
  name = "adafactor",
  ...,
  loss_scale_factor = NULL,
  gradient_accumulation_steps = NULL
)
```

Arguments

learning_rate	A float, a LearningRateSchedule() instance, or a callable that takes no arguments and returns the actual value to use. The learning rate. Defaults to 0.001.
beta_2_decay	float, defaults to -0.8. The decay rate of beta_2.
epsilon_1	float, defaults to 1e-30. A small offset to keep denominator away from 0.
epsilon_2	float, defaults to 1e-3. A small offset to avoid learning rate becoming too small by time.
clip_threshold	float, defaults to 1.0. Clipping threshold. This is a part of Adafactor algorithm, independent from clipnorm, clipvalue, and global_clipnorm.
relative_step	bool, defaults to TRUE. If learning_rate is a constant and relative_step=TRUE, learning rate will be adjusted based on current iterations. This is a default learning rate decay in Adafactor.
weight_decay	Float. If set, weight decay is applied.
clipnorm	Float. If set, the gradient of each weight is individually clipped so that its norm is no higher than this value.
clipvalue	Float. If set, the gradient of each weight is clipped to be no higher than this value.
global_clipnorm	Float. If set, the gradient of all weights is clipped so that their global norm is no higher than this value.
use_ema	Boolean, defaults to FALSE. If TRUE, exponential moving average (EMA) is applied. EMA consists of computing an exponential moving average of the weights of the model (as the weight values change after each training batch), and periodically overwriting the weights with their moving average.
ema_momentum	Float, defaults to 0.99. Only used if use_ema = TRUE. This is the momentum to use when computing the EMA of the model's weights: new_average = ema_momentum * old_average + (1 - ema_momentum) * current_variable_value.
ema_overwrite_frequency	Int or NULL, defaults to NULL. Only used if use_ema=TRUE. Every ema_overwrite_frequency steps of iterations, we overwrite the model variable by its moving average. If NULL, the optimizer does not overwrite model variables in the middle of training, and you need to explicitly overwrite the variables at the end of training by calling optimizer$finalize_variable_values() (which updates the model variables in-place). When using the built-in fit() training loop, this happens automatically after the last epoch, and you don't need to do anything.
name	String. The name to use for momentum accumulator weights created by the optimizer.
...	For forward/backward compatability.
loss_scale_factor	Float or NULL. If a float, the scale factor will be multiplied the loss before computing gradients, and the inverse of the scale factor will be multiplied by the gradients before updating variables. Useful for preventing underflow during mixed

precision training. Alternately, `optimizer_loss_scale()` will automatically set a loss scale factor.

`gradient_accumulation_steps`
: Int or `NULL`. If an int, model & optimizer variables will not be updated at every step; instead they will be updated every `gradient_accumulation_steps` steps, using the average value of the gradients since the last update. This is known as "gradient accumulation". This can be useful when your batch size is very small, in order to reduce gradient noise at each update step.

Value

an `Optimizer` instance

Reference

- Shazeer, Noam et al., 2018.

See Also

- `https://keras.io/api/optimizers/adafactor#adafactor-class`

Other optimizers:
`optimizer_adadelta()`
`optimizer_adagrad()`
`optimizer_adam()`
`optimizer_adam_w()`
`optimizer_adamax()`
`optimizer_ftrl()`
`optimizer_lion()`
`optimizer_loss_scale()`
`optimizer_nadam()`
`optimizer_rmsprop()`
`optimizer_sgd()`

`optimizer_adagrad` *Optimizer that implements the Adagrad algorithm.*

Description

Adagrad is an optimizer with parameter-specific learning rates, which are adapted relative to how frequently a parameter gets updated during training. The more updates a parameter receives, the smaller the updates.

Usage

```
optimizer_adagrad(
  learning_rate = 0.001,
  initial_accumulator_value = 0.1,
  epsilon = 1e-07,
  weight_decay = NULL,
  clipnorm = NULL,
  clipvalue = NULL,
  global_clipnorm = NULL,
  use_ema = FALSE,
  ema_momentum = 0.99,
  ema_overwrite_frequency = NULL,
  name = "adagrad",
  ...,
  loss_scale_factor = NULL,
  gradient_accumulation_steps = NULL
)
```

Arguments

learning_rate	A float, a LearningRateSchedule() instance, or a callable that takes no arguments and returns the actual value to use. The learning rate. Defaults to 0.001. Note that Adagrad tends to benefit from higher initial learning rate values compared to other optimizers. To match the exact form in the original paper, use 1.0.
initial_accumulator_value	Floating point value. Starting value for the accumulators (per-parameter momentum values). Must be non-negative.
epsilon	Small floating point value for maintaining numerical stability.
weight_decay	Float. If set, weight decay is applied.
clipnorm	Float. If set, the gradient of each weight is individually clipped so that its norm is no higher than this value.
clipvalue	Float. If set, the gradient of each weight is clipped to be no higher than this value.
global_clipnorm	Float. If set, the gradient of all weights is clipped so that their global norm is no higher than this value.
use_ema	Boolean, defaults to FALSE. If TRUE, exponential moving average (EMA) is applied. EMA consists of computing an exponential moving average of the weights of the model (as the weight values change after each training batch), and periodically overwriting the weights with their moving average.
ema_momentum	Float, defaults to 0.99. Only used if use_ema=TRUE. This is the momentum to use when computing the EMA of the model's weights: new_average = ema_momentum * old_average + (1 - ema_momentum) * current_variable_value.
ema_overwrite_frequency	Int or NULL, defaults to NULL. Only used if use_ema=TRUE. Every ema_overwrite_frequency steps of iterations, we overwrite the model variable by its moving average. If

	`NULL`, the optimizer does not overwrite model variables in the middle of training, and you need to explicitly overwrite the variables at the end of training by calling `optimizer$finalize_variable_values()` (which updates the model variables in-place). When using the built-in `fit()` training loop, this happens automatically after the last epoch, and you don't need to do anything.
`name`	String. The name to use for momentum accumulator weights created by the optimizer.
`...`	For forward/backward compatability.
`loss_scale_factor`	Float or `NULL`. If a float, the scale factor will be multiplied the loss before computing gradients, and the inverse of the scale factor will be multiplied by the gradients before updating variables. Useful for preventing underflow during mixed precision training. Alternately, `optimizer_loss_scale()` will automatically set a loss scale factor.
`gradient_accumulation_steps`	Int or `NULL`. If an int, model & optimizer variables will not be updated at every step; instead they will be updated every `gradient_accumulation_steps` steps, using the average value of the gradients since the last update. This is known as "gradient accumulation". This can be useful when your batch size is very small, in order to reduce gradient noise at each update step.

Value

an `Optimizer` instance

Reference

- Duchi et al., 2011.

See Also

- `https://keras.io/api/optimizers/adagrad#adagrad-class`

Other optimizers:
`optimizer_adadelta()`
`optimizer_adafactor()`
`optimizer_adam()`
`optimizer_adam_w()`
`optimizer_adamax()`
`optimizer_ftrl()`
`optimizer_lion()`
`optimizer_loss_scale()`
`optimizer_nadam()`
`optimizer_rmsprop()`
`optimizer_sgd()`

`optimizer_adam` *Optimizer that implements the Adam algorithm.*

Description

Adam optimization is a stochastic gradient descent method that is based on adaptive estimation of first-order and second-order moments.

According to Kingma et al., 2014, the method is "*computationally efficient, has little memory requirement, invariant to diagonal rescaling of gradients, and is well suited for problems that are large in terms of data/parameters*".

Usage

```
optimizer_adam(
  learning_rate = 0.001,
  beta_1 = 0.9,
  beta_2 = 0.999,
  epsilon = 1e-07,
  amsgrad = FALSE,
  weight_decay = NULL,
  clipnorm = NULL,
  clipvalue = NULL,
  global_clipnorm = NULL,
  use_ema = FALSE,
  ema_momentum = 0.99,
  ema_overwrite_frequency = NULL,
  name = "adam",
  ...,
  loss_scale_factor = NULL,
  gradient_accumulation_steps = NULL
)
```

Arguments

`learning_rate`	A float, a `LearningRateSchedule()` instance, or a callable that takes no arguments and returns the actual value to use. The learning rate. Defaults to `0.001`.
`beta_1`	A float value or a constant float tensor, or a callable that takes no arguments and returns the actual value to use. The exponential decay rate for the 1st moment estimates. Defaults to `0.9`.
`beta_2`	A float value or a constant float tensor, or a callable that takes no arguments and returns the actual value to use. The exponential decay rate for the 2nd moment estimates. Defaults to `0.999`.
`epsilon`	A small constant for numerical stability. This epsilon is "epsilon hat" in the Kingma and Ba paper (in the formula just before Section 2.1), not the epsilon in Algorithm 1 of the paper. Defaults to `1e-7`.

`amsgrad`	Boolean. Whether to apply AMSGrad variant of this algorithm from the paper "On the Convergence of Adam and beyond". Defaults to `FALSE`.
`weight_decay`	Float. If set, weight decay is applied.
`clipnorm`	Float. If set, the gradient of each weight is individually clipped so that its norm is no higher than this value.
`clipvalue`	Float. If set, the gradient of each weight is clipped to be no higher than this value.
`global_clipnorm`	Float. If set, the gradient of all weights is clipped so that their global norm is no higher than this value.
`use_ema`	Boolean, defaults to `FALSE`. If `TRUE`, exponential moving average (EMA) is applied. EMA consists of computing an exponential moving average of the weights of the model (as the weight values change after each training batch), and periodically overwriting the weights with their moving average.
`ema_momentum`	Float, defaults to 0.99. Only used if `use_ema=TRUE`. This is the momentum to use when computing the EMA of the model's weights: `new_average = ema_momentum * old_average + (1 - ema_momentum) * current_variable_value`.
`ema_overwrite_frequency`	Int or `NULL`, defaults to `NULL`. Only used if `use_ema=TRUE`. Every `ema_overwrite_frequency` steps of iterations, we overwrite the model variable by its moving average. If `NULL`, the optimizer does not overwrite model variables in the middle of training, and you need to explicitly overwrite the variables at the end of training by calling `optimizer$finalize_variable_values()` (which updates the model variables in-place). When using the built-in `fit()` training loop, this happens automatically after the last epoch, and you don't need to do anything.
`name`	String. The name to use for momentum accumulator weights created by the optimizer.
`...`	For forward/backward compatability.
`loss_scale_factor`	Float or `NULL`. If a float, the scale factor will be multiplied the loss before computing gradients, and the inverse of the scale factor will be multiplied by the gradients before updating variables. Useful for preventing underflow during mixed precision training. Alternately, `optimizer_loss_scale()` will automatically set a loss scale factor.
`gradient_accumulation_steps`	Int or `NULL`. If an int, model & optimizer variables will not be updated at every step; instead they will be updated every `gradient_accumulation_steps` steps, using the average value of the gradients since the last update. This is known as "gradient accumulation". This can be useful when your batch size is very small, in order to reduce gradient noise at each update step.

Value

an `Optimizer` instance

See Also

- https://keras.io/api/optimizers/adam#adam-class

Other optimizers:
optimizer_adadelta()
optimizer_adafactor()
optimizer_adagrad()
optimizer_adam_w()
optimizer_adamax()
optimizer_ftrl()
optimizer_lion()
optimizer_loss_scale()
optimizer_nadam()
optimizer_rmsprop()
optimizer_sgd()

---

optimizer_adamax *Optimizer that implements the Adamax algorithm.*

Description

Adamax, a variant of Adam based on the infinity norm, is a first-order gradient-based optimization method. Due to its capability of adjusting the learning rate based on data characteristics, it is suited to learn time-variant process, e.g., speech data with dynamically changed noise conditions. Default parameters follow those provided in the paper (see references below).

Initialization:

```
m <- 0  # Initialize initial 1st moment vector
u <- 0  # Initialize the exponentially weighted infinity norm
t <- 0  # Initialize timestep
```

The update rule for parameter w with gradient g is described at the end of section 7.1 of the paper (see the referenece section):

```
t <-  t + 1
m <- beta1 * m + (1 - beta) * g
u <- max(beta2 * u, abs(g))
current_lr <- learning_rate / (1 - beta1 ** t)
w <- w - current_lr * m / (u + epsilon)
```

Usage

```
optimizer_adamax(
  learning_rate = 0.001,
  beta_1 = 0.9,
  beta_2 = 0.999,
  epsilon = 1e-07,
  weight_decay = NULL,
  clipnorm = NULL,
  clipvalue = NULL,
  global_clipnorm = NULL,
  use_ema = FALSE,
  ema_momentum = 0.99,
  ema_overwrite_frequency = NULL,
  name = "adamax",
  ...,
  loss_scale_factor = NULL,
  gradient_accumulation_steps = NULL
)
```

Arguments

`learning_rate`	A float, a `LearningRateSchedule()` instance, or a callable that takes no arguments and returns the actual value to use. The learning rate. Defaults to `0.001`.
`beta_1`	A float value or a constant float tensor. The exponential decay rate for the 1st moment estimates.
`beta_2`	A float value or a constant float tensor. The exponential decay rate for the exponentially weighted infinity norm.
`epsilon`	A small constant for numerical stability. name: String. The name to use for momentum accumulator weights created by the optimizer.
`weight_decay`	Float. If set, weight decay is applied.
`clipnorm`	Float. If set, the gradient of each weight is individually clipped so that its norm is no higher than this value.
`clipvalue`	Float. If set, the gradient of each weight is clipped to be no higher than this value.
`global_clipnorm`	Float. If set, the gradient of all weights is clipped so that their global norm is no higher than this value.
`use_ema`	Boolean, defaults to `FALSE`. If `TRUE`, exponential moving average (EMA) is applied. EMA consists of computing an exponential moving average of the weights of the model (as the weight values change after each training batch), and periodically overwriting the weights with their moving average.
`ema_momentum`	Float, defaults to 0.99. Only used if `use_ema=TRUE`. This is the momentum to use when computing the EMA of the model's weights: `new_average = ema_momentum * old_average + (1 - ema_momentum) * current_variable_value`.
`ema_overwrite_frequency`	Int or NULL, defaults to NULL. Only used if `use_ema=TRUE`. Every `ema_overwrite_frequency` steps of iterations, we overwrite the model variable by its moving average. If NULL, the optimizer does not overwrite model variables in the middle of training, and you need to explicitly overwrite the variables at the end of training by calling `optimizer$finalize_variable_values()` (which updates the model

	variables in-place). When using the built-in `fit()` training loop, this happens automatically after the last epoch, and you don't need to do anything.
`name`	String, name for the object
`...`	For forward/backward compatability.
`loss_scale_factor`	Float or `NULL`. If a float, the scale factor will be multiplied the loss before computing gradients, and the inverse of the scale factor will be multiplied by the gradients before updating variables. Useful for preventing underflow during mixed precision training. Alternately, `optimizer_loss_scale()` will automatically set a loss scale factor.
`gradient_accumulation_steps`	Int or `NULL`. If an int, model & optimizer variables will not be updated at every step; instead they will be updated every `gradient_accumulation_steps` steps, using the average value of the gradients since the last update. This is known as "gradient accumulation". This can be useful when your batch size is very small, in order to reduce gradient noise at each update step.

Value

an `Optimizer` instance

Reference

- Kingma et al., 2014

See Also

- `https://keras.io/api/optimizers/adamax#adamax-class`

Other optimizers:
`optimizer_adadelta()`
`optimizer_adafactor()`
`optimizer_adagrad()`
`optimizer_adam()`
`optimizer_adam_w()`
`optimizer_ftrl()`
`optimizer_lion()`
`optimizer_loss_scale()`
`optimizer_nadam()`
`optimizer_rmsprop()`
`optimizer_sgd()`

optimizer_adam_w *Optimizer that implements the AdamW algorithm.*

Description

AdamW optimization is a stochastic gradient descent method that is based on adaptive estimation of first-order and second-order moments with an added method to decay weights per the techniques discussed in the paper, 'Decoupled Weight Decay Regularization' by Loshchilov, Hutter et al., 2019.

According to Kingma et al., 2014, the underying Adam method is "*computationally efficient, has little memory requirement, invariant to diagonal rescaling of gradients, and is well suited for problems that are large in terms of data/parameters*".

Usage

```
optimizer_adam_w(
  learning_rate = 0.001,
  weight_decay = 0.004,
  beta_1 = 0.9,
  beta_2 = 0.999,
  epsilon = 1e-07,
  amsgrad = FALSE,
  clipnorm = NULL,
  clipvalue = NULL,
  global_clipnorm = NULL,
  use_ema = FALSE,
  ema_momentum = 0.99,
  ema_overwrite_frequency = NULL,
  name = "adamw",
  ...,
  loss_scale_factor = NULL,
  gradient_accumulation_steps = NULL
)
```

Arguments

`learning_rate`	A float, a `LearningRateSchedule()` instance, or a callable that takes no arguments and returns the actual value to use. The learning rate. Defaults to `0.001`.
`weight_decay`	Float. If set, weight decay is applied.
`beta_1`	A float value or a constant float tensor, or a callable that takes no arguments and returns the actual value to use. The exponential decay rate for the 1st moment estimates. Defaults to `0.9`.
`beta_2`	A float value or a constant float tensor, or a callable that takes no arguments and returns the actual value to use. The exponential decay rate for the 2nd moment estimates. Defaults to `0.999`.

`epsilon`	A small constant for numerical stability. This epsilon is "epsilon hat" in the Kingma and Ba paper (in the formula just before Section 2.1), not the epsilon in Algorithm 1 of the paper. Defaults to 1e-7.
`amsgrad`	Boolean. Whether to apply AMSGrad variant of this algorithm from the paper "On the Convergence of Adam and beyond". Defaults to `FALSE`.
`clipnorm`	Float. If set, the gradient of each weight is individually clipped so that its norm is no higher than this value.
`clipvalue`	Float. If set, the gradient of each weight is clipped to be no higher than this value.
`global_clipnorm`	Float. If set, the gradient of all weights is clipped so that their global norm is no higher than this value.
`use_ema`	Boolean, defaults to `FALSE`. If `TRUE`, exponential moving average (EMA) is applied. EMA consists of computing an exponential moving average of the weights of the model (as the weight values change after each training batch), and periodically overwriting the weights with their moving average.
`ema_momentum`	Float, defaults to 0.99. Only used if `use_ema=TRUE`. This is the momentum to use when computing the EMA of the model's weights: `new_average = ema_momentum * old_average + (1 - ema_momentum) * current_variable_value`.
`ema_overwrite_frequency`	Int or `NULL`, defaults to `NULL`. Only used if `use_ema=TRUE`. Every `ema_overwrite_frequency` steps of iterations, we overwrite the model variable by its moving average. If `NULL`, the optimizer does not overwrite model variables in the middle of training, and you need to explicitly overwrite the variables at the end of training by calling `optimizer$finalize_variable_values()` (which updates the model variables in-place). When using the built-in `fit()` training loop, this happens automatically after the last epoch, and you don't need to do anything.
`name`	String. The name to use for momentum accumulator weights created by the optimizer.
`...`	For forward/backward compatability.
`loss_scale_factor`	Float or `NULL`. If a float, the scale factor will be multiplied the loss before computing gradients, and the inverse of the scale factor will be multiplied by the gradients before updating variables. Useful for preventing underflow during mixed precision training. Alternately, `optimizer_loss_scale()` will automatically set a loss scale factor.
`gradient_accumulation_steps`	Int or `NULL`. If an int, model & optimizer variables will not be updated at every step; instead they will be updated every `gradient_accumulation_steps` steps, using the average value of the gradients since the last update. This is known as "gradient accumulation". This can be useful when your batch size is very small, in order to reduce gradient noise at each update step.

Value

an `Optimizer` instance

References

- Loshchilov et al., 2019
- Kingma et al., 2014 for `adam`
- Reddi et al., 2018 for `amsgrad`.

See Also

- `https://keras.io/api/optimizers/adamw#adamw-class`

Other optimizers:
`optimizer_adadelta()`
`optimizer_adafactor()`
`optimizer_adagrad()`
`optimizer_adam()`
`optimizer_adamax()`
`optimizer_ftrl()`
`optimizer_lion()`
`optimizer_loss_scale()`
`optimizer_nadam()`
`optimizer_rmsprop()`
`optimizer_sgd()`

`optimizer_ftrl` *Optimizer that implements the FTRL algorithm.*

Description

"Follow The Regularized Leader" (FTRL) is an optimization algorithm developed at Google for click-through rate prediction in the early 2010s. It is most suitable for shallow models with large and sparse feature spaces. The algorithm is described by McMahan et al., 2013. The Keras version has support for both online L2 regularization (the L2 regularization described in the paper above) and shrinkage-type L2 regularization (which is the addition of an L2 penalty to the loss function).

Initialization:

```
n <- 0
sigma <- 0
z <- 0
```

Update rule for one variable `w`:

```
prev_n <- n
n <- n + g^2
sigma <- (n^(-lr_power) - prev_n^(-lr_power)) / lr
z <- z + g - sigma * w
if (abs(z) < lambda_1) {
  w <- 0
} else {
  w <- (sgn(z) * lambda_1 - z) / ((beta + sqrt(n)) / alpha + lambda_2)
}
```

Notation:

- `lr` is the learning rate
- g is the gradient for the variable
- `lambda_1` is the L1 regularization strength
- `lambda_2` is the L2 regularization strength
- `lr_power` is the power to scale n.

Check the documentation for the `l2_shrinkage_regularization_strength` parameter for more details when shrinkage is enabled, in which case gradient is replaced with a gradient with shrinkage.

Usage

```
optimizer_ftrl(
  learning_rate = 0.001,
  learning_rate_power = -0.5,
  initial_accumulator_value = 0.1,
  l1_regularization_strength = 0,
  l2_regularization_strength = 0,
  l2_shrinkage_regularization_strength = 0,
  beta = 0,
  weight_decay = NULL,
  clipnorm = NULL,
  clipvalue = NULL,
  global_clipnorm = NULL,
  use_ema = FALSE,
  ema_momentum = 0.99,
  ema_overwrite_frequency = NULL,
  name = "ftrl",
  ...,
  loss_scale_factor = NULL,
  gradient_accumulation_steps = NULL
)
```

Arguments

`learning_rate`	A float, a `LearningRateSchedule()` instance, or a callable that takes no arguments and returns the actual value to use. The learning rate. Defaults to `0.001`.
`learning_rate_power`	A float value, must be less or equal to zero. Controls how the learning rate decreases during training. Use zero for a fixed learning rate.
`initial_accumulator_value`	The starting value for accumulators. Only zero or positive values are allowed.

`l1_regularization_strength`	A float value, must be greater than or equal to zero. Defaults to `0.0`.
`l2_regularization_strength`	A float value, must be greater than or equal to zero. Defaults to `0.0`.
`l2_shrinkage_regularization_strength`	A float value, must be greater than or equal to zero. This differs from L2 above in that the L2 above is a stabilization penalty, whereas this L2 shrinkage is a magnitude penalty. When input is sparse shrinkage will only happen on the active weights.
`beta`	A float value, representing the beta value from the paper. Defaults to `0.0`.
`weight_decay`	Float. If set, weight decay is applied.
`clipnorm`	Float. If set, the gradient of each weight is individually clipped so that its norm is no higher than this value.
`clipvalue`	Float. If set, the gradient of each weight is clipped to be no higher than this value.
`global_clipnorm`	Float. If set, the gradient of all weights is clipped so that their global norm is no higher than this value.
`use_ema`	Boolean, defaults to `FALSE`. If `TRUE`, exponential moving average (EMA) is applied. EMA consists of computing an exponential moving average of the weights of the model (as the weight values change after each training batch), and periodically overwriting the weights with their moving average.
`ema_momentum`	Float, defaults to 0.99. Only used if `use_ema=TRUE`. This is the momentum to use when computing the EMA of the model's weights: `new_average = ema_momentum * old_average + (1 - ema_momentum) * current_variable_value`.
`ema_overwrite_frequency`	Int or NULL, defaults to NULL. Only used if `use_ema=TRUE`. Every `ema_overwrite_frequency` steps of iterations, we overwrite the model variable by its moving average. If NULL, the optimizer does not overwrite model variables in the middle of training, and you need to explicitly overwrite the variables at the end of training by calling `optimizer$finalize_variable_values()` (which updates the model variables in-place). When using the built-in `fit()` training loop, this happens automatically after the last epoch, and you don't need to do anything.
`name`	String. The name to use for momentum accumulator weights created by the optimizer.
`...`	For forward/backward compatability.
`loss_scale_factor`	Float or `NULL`. If a float, the scale factor will be multiplied the loss before computing gradients, and the inverse of the scale factor will be multiplied by the gradients before updating variables. Useful for preventing underflow during mixed precision training. Alternately, `optimizer_loss_scale` will automatically set a loss scale factor.
`gradient_accumulation_steps`	Int or `NULL`. If an int, model & optimizer variables will not be updated at every step; instead they will be updated every `gradient_accumulation_steps` steps,

using the average value of the gradients since the last update. This is known as "gradient accumulation". This can be useful when your batch size is very small, in order to reduce gradient noise at each update step.

Value

an `Optimizer` instance

See Also

- https://keras.io/api/optimizers/ftrl#ftrl-class

Other optimizers:
`optimizer_adadelta()`
`optimizer_adafactor()`
`optimizer_adagrad()`
`optimizer_adam()`
`optimizer_adam_w()`
`optimizer_adamax()`
`optimizer_lion()`
`optimizer_loss_scale()`
`optimizer_nadam()`
`optimizer_rmsprop()`
`optimizer_sgd()`

`optimizer_lion` *Optimizer that implements the Lion algorithm.*

Description

The Lion optimizer is a stochastic-gradient-descent method that uses the sign operator to control the magnitude of the update, unlike other adaptive optimizers such as Adam that rely on second-order moments. This make Lion more memory-efficient as it only keeps track of the momentum. According to the authors (see reference), its performance gain over Adam grows with the batch size. Because the update of Lion is produced through the sign operation, resulting in a larger norm, a suitable learning rate for Lion is typically 3-10x smaller than that for AdamW. The weight decay for Lion should be in turn 3-10x larger than that for AdamW to maintain a similar strength (lr * wd).

Usage

```
optimizer_lion(
  learning_rate = 0.001,
  beta_1 = 0.9,
  beta_2 = 0.99,
  weight_decay = NULL,
  clipnorm = NULL,
  clipvalue = NULL,
  global_clipnorm = NULL,
  use_ema = FALSE,
  ema_momentum = 0.99,
  ema_overwrite_frequency = NULL,
  name = "lion",
  ...,
  loss_scale_factor = NULL,
  gradient_accumulation_steps = NULL
)
```

Arguments

`learning_rate`	A float, a `LearningRateSchedule()` instance, or a callable that takes no arguments and returns the actual value to use. The learning rate. Defaults to `0.001`.
`beta_1`	A float value or a constant float tensor, or a callable that takes no arguments and returns the actual value to use. The rate to combine the current gradient and the 1st moment estimate. Defaults to `0.9`.
`beta_2`	A float value or a constant float tensor, or a callable that takes no arguments and returns the actual value to use. The exponential decay rate for the 1st moment estimate. Defaults to `0.99`.
`weight_decay`	Float. If set, weight decay is applied.
`clipnorm`	Float. If set, the gradient of each weight is individually clipped so that its norm is no higher than this value.
`clipvalue`	Float. If set, the gradient of each weight is clipped to be no higher than this value.
`global_clipnorm`	Float. If set, the gradient of all weights is clipped so that their global norm is no higher than this value.
`use_ema`	Boolean, defaults to `FALSE`. If `TRUE`, exponential moving average (EMA) is applied. EMA consists of computing an exponential moving average of the weights of the model (as the weight values change after each training batch), and periodically overwriting the weights with their moving average.
`ema_momentum`	Float, defaults to 0.99. Only used if `use_ema=TRUE`. This is the momentum to use when computing the EMA of the model's weights: `new_average = ema_momentum * old_average + (1 - ema_momentum) * current_variable_value`.
`ema_overwrite_frequency`	Int or `NULL`, defaults to `NULL`. Only used if `use_ema=TRUE`. Every `ema_overwrite_frequency` steps of iterations, we overwrite the model variable by its moving average. If `NULL`, the optimizer does not overwrite model variables in the middle of training, and you need to explicitly overwrite the variables at the end of training by calling `optimizer$finalize_variable_values()` (which updates the model variables in-place). When using the built-in `fit()` training loop, this happens automatically after the last epoch, and you don't need to do anything.
`name`	String. The name to use for momentum accumulator weights created by the optimizer.

`...` For forward/backward compatability.

`loss_scale_factor`
: Float or `NULL`. If a float, the scale factor will be multiplied the loss before computing gradients, and the inverse of the scale factor will be multiplied by the gradients before updating variables. Useful for preventing underflow during mixed precision training. Alternately, `optimizer_loss_scale()` will automatically set a loss scale factor.

`gradient_accumulation_steps`
: Int or `NULL`. If an int, model & optimizer variables will not be updated at every step; instead they will be updated every `gradient_accumulation_steps` steps, using the average value of the gradients since the last update. This is known as "gradient accumulation". This can be useful when your batch size is very small, in order to reduce gradient noise at each update step.

Value

an `Optimizer` instance

References

- Chen et al., 2023
- Authors' implementation

See Also

Other optimizers:
`optimizer_adadelta()`
`optimizer_adafactor()`
`optimizer_adagrad()`
`optimizer_adam()`
`optimizer_adam_w()`
`optimizer_adamax()`
`optimizer_ftrl()`
`optimizer_loss_scale()`
`optimizer_nadam()`
`optimizer_rmsprop()`
`optimizer_sgd()`

`optimizer_loss_scale` *An optimizer that dynamically scales the loss to prevent underflow.*

Description

Loss scaling is a technique to prevent numeric underflow in intermediate gradients when float16 is used. To prevent underflow, the loss is multiplied (or "scaled") by a certain factor called the "loss scale", which causes intermediate gradients to be scaled by the loss scale as well. The final gradients are divided (or "unscaled") by the loss scale to bring them back to their original value.

`LossScaleOptimizer` wraps another optimizer and applies dynamic loss scaling to it. This loss scale is dynamically updated over time as follows:

- On any train step, if a nonfinite gradient is encountered, the loss scale is halved, and the train step is skipped.
- If `dynamic_growth_steps` have ocurred since the last time the loss scale was updated, and no nonfinite gradients have occurred, the loss scale is doubled.

Usage

```
optimizer_loss_scale(
  inner_optimizer,
  initial_scale = 32768,
  dynamic_growth_steps = 2000L,
  ...,
  name = NULL,
  weight_decay = NULL,
  clipnorm = NULL,
  clipvalue = NULL,
  global_clipnorm = NULL,
  use_ema = NULL,
  ema_momentum = NULL,
  ema_overwrite_frequency = NULL,
  loss_scale_factor = NULL,
  gradient_accumulation_steps = NULL
)
```

Arguments

`inner_optimizer`	The keras `Optimizer` instance to wrap.
`initial_scale`	Float. The initial loss scale. This scale will be updated during training. It is recommended for this to be a very high number, because a loss scale that is too high gets lowered far more quickly than a loss scale that is too low gets raised.
`dynamic_growth_steps`	Int. How often to update the scale upwards. After every `dynamic_growth_steps` steps with finite gradients, the loss scale is doubled.
`...`	For forward/backward compatability.
`name`	String. The name to use for momentum accumulator weights created by the optimizer.
`weight_decay`	Float. If set, weight decay is applied.

`clipnorm`	Float. If set, the gradient of each weight is individually clipped so that its norm is no higher than this value.
`clipvalue`	Float. If set, the gradient of each weight is clipped to be no higher than this value.
`global_clipnorm`	Float. If set, the gradient of all weights is clipped so that their global norm is no higher than this value.
`use_ema`	Boolean, defaults to `FALSE`. If `TRUE`, exponential moving average (EMA) is applied. EMA consists of computing an exponential moving average of the weights of the model (as the weight values change after each training batch), and periodically overwriting the weights with their moving average.
`ema_momentum`	Float, defaults to 0.99. Only used if `use_ema=TRUE`. This is the momentum to use when computing the EMA of the model's weights: `new_average = ema_momentum * old_average + (1 - ema_momentum) * current_variable_value`.
`ema_overwrite_frequency`	Int or `NULL`, defaults to `NULL`. Only used if `use_ema=TRUE`. Every `ema_overwrite_frequency` steps of iterations, we overwrite the model variable by its moving average. If `NULL`, the optimizer does not overwrite model variables in the middle of training, and you need to explicitly overwrite the variables at the end of training by calling `optimizer$finalize_variable_values()` (which updates the model variables in-place). When using the built-in `fit()` training loop, this happens automatically after the last epoch, and you don't need to do anything.
`loss_scale_factor`	Float or `NULL`. If a float, the scale factor will be multiplied the loss before computing gradients, and the inverse of the scale factor will be multiplied by the gradients before updating variables. Useful for preventing underflow during mixed precision training. Alternately, `optimizer_loss_scale()` will automatically set a loss scale factor.
`gradient_accumulation_steps`	Int or `NULL`. If an int, model & optimizer variables will not be updated at every step; instead they will be updated every `gradient_accumulation_steps` steps, using the average value of the gradients since the last update. This is known as "gradient accumulation". This can be useful when your batch size is very small, in order to reduce gradient noise at each update step.

Value

an `Optimizer` instance

See Also

Other optimizers:
`optimizer_adadelta()`
`optimizer_adafactor()`
`optimizer_adagrad()`
`optimizer_adam()`
`optimizer_adam_w()`
`optimizer_adamax()`

optimizer_ftrl()
optimizer_lion()
optimizer_nadam()
optimizer_rmsprop()
optimizer_sgd()

---

optimizer_nadam *Optimizer that implements the Nadam algorithm.*

Description

Much like Adam is essentially RMSprop with momentum, Nadam is Adam with Nesterov momentum.

Usage

```
optimizer_nadam(
  learning_rate = 0.001,
  beta_1 = 0.9,
  beta_2 = 0.999,
  epsilon = 1e-07,
  weight_decay = NULL,
  clipnorm = NULL,
  clipvalue = NULL,
  global_clipnorm = NULL,
  use_ema = FALSE,
  ema_momentum = 0.99,
  ema_overwrite_frequency = NULL,
  name = "nadam",
  ...,
  loss_scale_factor = NULL,
  gradient_accumulation_steps = NULL
)
```

Arguments

learning_rate	A float, a LearningRateSchedule() instance, or a callable that takes no arguments and returns the actual value to use. The learning rate. Defaults to 0.001.
beta_1	A float value or a constant float tensor, or a callable that takes no arguments and returns the actual value to use. The exponential decay rate for the 1st moment estimates. Defaults to 0.9.
beta_2	A float value or a constant float tensor, or a callable that takes no arguments and returns the actual value to use. The exponential decay rate for the 2nd moment estimates. Defaults to 0.999.

`epsilon`	A small constant for numerical stability. This epsilon is "epsilon hat" in the Kingma and Ba paper (in the formula just before Section 2.1), not the epsilon in Algorithm 1 of the paper. Defaults to `1e-7`.
`weight_decay`	Float. If set, weight decay is applied.
`clipnorm`	Float. If set, the gradient of each weight is individually clipped so that its norm is no higher than this value.
`clipvalue`	Float. If set, the gradient of each weight is clipped to be no higher than this value.
`global_clipnorm`	Float. If set, the gradient of all weights is clipped so that their global norm is no higher than this value.
`use_ema`	Boolean, defaults to `FALSE`. If `TRUE`, exponential moving average (EMA) is applied. EMA consists of computing an exponential moving average of the weights of the model (as the weight values change after each training batch), and periodically overwriting the weights with their moving average.
`ema_momentum`	Float, defaults to 0.99. Only used if `use_ema=TRUE`. This is the momentum to use when computing the EMA of the model's weights: `new_average = ema_momentum * old_average + (1 - ema_momentum) * current_variable_value`.
`ema_overwrite_frequency`	Int or `NULL`, defaults to `NULL`. Only used if `use_ema=TRUE`. Every `ema_overwrite_frequency` steps of iterations, we overwrite the model variable by its moving average. If `NULL`, the optimizer does not overwrite model variables in the middle of training, and you need to explicitly overwrite the variables at the end of training by calling `optimizer$finalize_variable_values()` (which updates the model variables in-place). When using the built-in `fit()` training loop, this happens automatically after the last epoch, and you don't need to do anything.
`name`	String. The name to use for momentum accumulator weights created by the optimizer.
`...`	For forward/backward compatability.
`loss_scale_factor`	Float or `NULL`. If a float, the scale factor will be multiplied the loss before computing gradients, and the inverse of the scale factor will be multiplied by the gradients before updating variables. Useful for preventing underflow during mixed precision training. Alternately, `optimizer_loss_scale()` will automatically set a loss scale factor.
`gradient_accumulation_steps`	Int or `NULL`. If an int, model & optimizer variables will not be updated at every step; instead they will be updated every `gradient_accumulation_steps` steps, using the average value of the gradients since the last update. This is known as "gradient accumulation". This can be useful when your batch size is very small, in order to reduce gradient noise at each update step.

Value

an `Optimizer` instance

Reference

- Dozat, 2015.

See Also

- https://keras.io/api/optimizers/Nadam#nadam-class

Other optimizers:
`optimizer_adadelta()`
`optimizer_adafactor()`
`optimizer_adagrad()`
`optimizer_adam()`
`optimizer_adam_w()`
`optimizer_adamax()`
`optimizer_ftrl()`
`optimizer_lion()`
`optimizer_loss_scale()`
`optimizer_rmsprop()`
`optimizer_sgd()`

optimizer_rmsprop *Optimizer that implements the RMSprop algorithm.*

Description

The gist of RMSprop is to:

- Maintain a moving (discounted) average of the square of gradients
- Divide the gradient by the root of this average

This implementation of RMSprop uses plain momentum, not Nesterov momentum.

The centered version additionally maintains a moving average of the gradients, and uses that average to estimate the variance.

Usage

```
optimizer_rmsprop(
  learning_rate = 0.001,
  rho = 0.9,
  momentum = 0,
  epsilon = 1e-07,
  centered = FALSE,
  weight_decay = NULL,
  clipnorm = NULL,
  clipvalue = NULL,
  global_clipnorm = NULL,
  use_ema = FALSE,
  ema_momentum = 0.99,
  ema_overwrite_frequency = 100L,
  name = "rmsprop",
  ...,
  loss_scale_factor = NULL,
  gradient_accumulation_steps = NULL
)
```

Arguments

`learning_rate`	A float, a `learning_rate_schedule_*` instance, or a callable that takes no arguments and returns the actual value to use. The learning rate. Defaults to `0.001`.
`rho`	float, defaults to 0.9. Discounting factor for the old gradients.
`momentum`	float, defaults to 0.0. If not 0.0., the optimizer tracks the momentum value, with a decay rate equals to `1 - momentum`.
`epsilon`	A small constant for numerical stability. This epsilon is "epsilon hat" in the Kingma and Ba paper (in the formula just before Section 2.1), not the epsilon in Algorithm 1 of the paper. Defaults to 1e-7.
`centered`	Boolean. If `TRUE`, gradients are normalized by the estimated variance of the gradient; if FALSE, by the uncentered second moment. Setting this to `TRUE` may help with training, but is slightly more expensive in terms of computation and memory. Defaults to `FALSE`.
`weight_decay`	Float. If set, weight decay is applied.
`clipnorm`	Float. If set, the gradient of each weight is individually clipped so that its norm is no higher than this value.
`clipvalue`	Float. If set, the gradient of each weight is clipped to be no higher than this value.
`global_clipnorm`	Float. If set, the gradient of all weights is clipped so that their global norm is no higher than this value.
`use_ema`	Boolean, defaults to `FALSE`. If `TRUE`, exponential moving average (EMA) is applied. EMA consists of computing an exponential moving average of the weights of the model (as the weight values change after each training batch), and periodically overwriting the weights with their moving average.
`ema_momentum`	Float, defaults to 0.99. Only used if `use_ema=TRUE`. This is the momentum to use when computing the EMA of the model's weights: `new_average = ema_momentum * old_average + (1 - ema_momentum) * current_variable_value`.
`ema_overwrite_frequency`	Int or NULL, defaults to NULL. Only used if `use_ema=TRUE`. Every `ema_overwrite_frequency` steps of iterations, we overwrite the model variable by its moving average. If NULL, the optimizer does not overwrite model variables in the middle of training, and you need to explicitly overwrite the variables at the end of training by calling `optimizer$finalize_variable_values()` (which updates the model variables in-place). When using the built-in `fit()` training loop, this happens automatically after the last epoch, and you don't need to do anything.

`name`	String. The name to use for momentum accumulator weights created by the optimizer.
`...`	For forward/backward compatability.
`loss_scale_factor`	Float or `NULL`. If a float, the scale factor will be multiplied the loss before computing gradients, and the inverse of the scale factor will be multiplied by the gradients before updating variables. Useful for preventing underflow during mixed precision training. Alternately, `optimizer_loss_scale()` will automatically set a loss scale factor.
`gradient_accumulation_steps`	Int or `NULL`. If an int, model & optimizer variables will not be updated at every step; instead they will be updated every `gradient_accumulation_steps` steps, using the average value of the gradients since the last update. This is known as "gradient accumulation". This can be useful when your batch size is very small, in order to reduce gradient noise at each update step.

Value

an `Optimizer` instance

Usage

```
opt <- optimizer_rmsprop(learning_rate=0.1)
```

Reference

- Hinton, 2012

See Also

- `https://keras.io/api/optimizers/rmsprop#rmsprop-class`

Other optimizers:
`optimizer_adadelta()`
`optimizer_adafactor()`
`optimizer_adagrad()`
`optimizer_adam()`
`optimizer_adam_w()`
`optimizer_adamax()`
`optimizer_ftrl()`
`optimizer_lion()`
`optimizer_loss_scale()`
`optimizer_nadam()`
`optimizer_sgd()`

optimizer_sgd *Gradient descent (with momentum) optimizer.*

Description

Update rule for parameter `w` with gradient `g` when `momentum` is 0:

```
w <- w - learning_rate * g
```

Update rule when `momentum` is larger than 0:

```
velocity <- momentum * velocity - learning_rate * g
w <- w + velocity
```

When `nesterov=TRUE`, this rule becomes:

```
velocity <- momentum * velocity - learning_rate * g
w <- w + momentum * velocity - learning_rate * g
```

Usage

```
optimizer_sgd(
  learning_rate = 0.01,
  momentum = 0,
  nesterov = FALSE,
  weight_decay = NULL,
  clipnorm = NULL,
  clipvalue = NULL,
  global_clipnorm = NULL,
  use_ema = FALSE,
  ema_momentum = 0.99,
  ema_overwrite_frequency = NULL,
  name = "SGD",
  ...,
  loss_scale_factor = NULL,
  gradient_accumulation_steps = NULL
)
```

Arguments

`learning_rate`	A float, a `learning_rate_schedule_*` instance, or a callable that takes no arguments and returns the actual value to use. The learning rate. Defaults to `0.01`.
`momentum`	float hyperparameter >= 0 that accelerates gradient descent in the relevant direction and dampens oscillations. 0 is vanilla gradient descent. Defaults to `0.0`.
`nesterov`	boolean. Whether to apply Nesterov momentum. Defaults to `FALSE`.

`weight_decay`	Float. If set, weight decay is applied.
`clipnorm`	Float. If set, the gradient of each weight is individually clipped so that its norm is no higher than this value.
`clipvalue`	Float. If set, the gradient of each weight is clipped to be no higher than this value.
`global_clipnorm`	Float. If set, the gradient of all weights is clipped so that their global norm is no higher than this value.
`use_ema`	Boolean, defaults to `FALSE`. If `TRUE`, exponential moving average (EMA) is applied. EMA consists of computing an exponential moving average of the weights of the model (as the weight values change after each training batch), and periodically overwriting the weights with their moving average.
`ema_momentum`	Float, defaults to 0.99. Only used if `use_ema=TRUE`. This is the momentum to use when computing the EMA of the model's weights: `new_average = ema_momentum * old_average + (1 - ema_momentum) * current_variable_value`.
`ema_overwrite_frequency`	Int or NULL, defaults to NULL. Only used if `use_ema=TRUE`. Every `ema_overwrite_frequency` steps of iterations, we overwrite the model variable by its moving average. If NULL, the optimizer does not overwrite model variables in the middle of training, and you need to explicitly overwrite the variables at the end of training by calling `optimizer$finalize_variable_values()` (which updates the model variables in-place). When using the built-in `fit()` training loop, this happens automatically after the last epoch, and you don't need to do anything.
`name`	String. The name to use for momentum accumulator weights created by the optimizer.
`...`	For forward/backward compatability.
`loss_scale_factor`	Float or `NULL`. If a float, the scale factor will be multiplied the loss before computing gradients, and the inverse of the scale factor will be multiplied by the gradients before updating variables. Useful for preventing underflow during mixed precision training. Alternately, `optimizer_loss_scale()` will automatically set a loss scale factor.
`gradient_accumulation_steps`	Int or `NULL`. If an int, model & optimizer variables will not be updated at every step; instead they will be updated every `gradient_accumulation_steps` steps, using the average value of the gradients since the last update. This is known as "gradient accumulation". This can be useful when your batch size is very small, in order to reduce gradient noise at each update step.

Value

an `Optimizer` instance

See Also

- https://keras.io/api/optimizers/sgd#sgd-class

Other optimizers:
optimizer_adadelta()
optimizer_adafactor()
optimizer_adagrad()
optimizer_adam()
optimizer_adam_w()
optimizer_adamax()
optimizer_ftrl()
optimizer_lion()
optimizer_loss_scale()
optimizer_nadam()
optimizer_rmsprop()

---

op_abs — *Compute the absolute value element-wise.*

Description

Compute the absolute value element-wise.

Usage

```
op_abs(x)
```

Arguments

x	Input tensor

Value

An array containing the absolute value of each element in x.

Example

```
x <- op_convert_to_tensor(c(-1.2, 1.2))
op_abs(x)

## tf.Tensor([1.2 1.2], shape=(2), dtype=float32)
```

See Also

Other numpy ops:
op_add()
op_all()
op_any()
op_append()

op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_bincount()
op_broadcast_to()
op_ceil()
op_clip()
op_concatenate()
op_conj()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_cumprod()
op_cumsum()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_einsum()
op_empty()
op_equal()
op_exp()
op_expand_dims()
op_expm1()
op_eye()
op_flip()
op_floor()
op_floor_divide()
op_full()
op_full_like()
op_get_item()
op_greater()

op_greater_equal()
op_hstack()
op_identity()
op_imag()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_matmul()
op_max()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moveaxis()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_not_equal()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_repeat()

op_reshape()
op_roll()
op_round()
op_sign()
op_sin()
op_sinh()
op_size()
op_sort()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_subtract()
op_sum()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_var()
op_vdot()
op_vstack()
op_where()
op_zeros()
op_zeros_like()

Other ops:
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()

op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()

op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()

op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()

op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()

op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

op_add *Add arguments element-wise.*

Description

Add arguments element-wise.

Usage

```
op_add(x1, x2)
```

Arguments

x1	First input tensor.
x2	Second input tensor.

Value

The tensor containing the element-wise sum of x1 and x2.

Examples

```
x1 <- op_convert_to_tensor(c(1, 4))
x2 <- op_convert_to_tensor(c(5, 6))
op_add(x1, x2)

## tf.Tensor([ 6. 10.], shape=(2), dtype=float32)


# alias for x1 + x2
x1 + x2

## tf.Tensor([ 6. 10.], shape=(2), dtype=float32)
```

op_add also broadcasts shapes:

```
x1 <- op_convert_to_tensor(array(c(5, 5, 4, 6), dim =c(2, 2)))
x2 <- op_convert_to_tensor(c(5, 6))
op_add(x1, x2)
```

```
## tf.Tensor(
## [[10. 10.]
##  [10. 12.]], shape=(2, 2), dtype=float64)
```

Note that this function is automatically called when using the R operator + with tensors.

```
x <- op_ones(c(3))
op_add(x, x)
```

```
## tf.Tensor([2. 2. 2.], shape=(3), dtype=float32)
```

```
x + x
```

```
## tf.Tensor([2. 2. 2.], shape=(3), dtype=float32)
```

See Also

- https://keras.io/api/ops/numpy#add-function

Other numpy ops:
op_abs()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_bincount()
op_broadcast_to()
op_ceil()
op_clip()
op_concatenate()

op_conj()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_cumprod()
op_cumsum()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_einsum()
op_empty()
op_equal()
op_exp()
op_expand_dims()
op_expm1()
op_eye()
op_flip()
op_floor()
op_floor_divide()
op_full()
op_full_like()
op_get_item()
op_greater()
op_greater_equal()
op_hstack()
op_identity()
op_imag()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_logaddexp()
op_logical_and()
op_logical_not()

op_logical_or()
op_logical_xor()
op_logspace()
op_matmul()
op_max()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moveaxis()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_not_equal()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_repeat()
op_reshape()
op_roll()
op_round()
op_sign()
op_sin()
op_sinh()
op_size()
op_sort()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_subtract()
op_sum()
op_swapaxes()
op_take()

op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_var()
op_vdot()
op_vstack()
op_where()
op_zeros()
op_zeros_like()

Other ops:
op_abs()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()

op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()

op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()

op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()

op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

---

op_all — *Test whether all array elements along a given axis evaluate to* TRUE.

Description

Test whether all array elements along a given axis evaluate to TRUE.

Usage

```
op_all(x, axis = NULL, keepdims = FALSE)
```

Arguments

`x`	Input tensor.
`axis`	An integer or tuple of integers that represent the axis along which a logical AND reduction is performed. The default (`axis = NULL`) is to perform a logical AND over all the dimensions of the input array. `axis` may be negative, in which case it counts for the last to the first axis.
`keepdims`	If `TRUE`, axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the input array. Defaults to`FALSE`.

Value

The tensor containing the logical AND reduction over the `axis`.

Examples

```
x <- op_convert_to_tensor(c(TRUE, FALSE))
op_all(x)

## tf.Tensor(False, shape=(), dtype=bool)
```

```
(x <- op_convert_to_tensor(array(c(TRUE, FALSE, TRUE, TRUE, TRUE, TRUE), dim = c(3, 2))))

## tf.Tensor(
## [[ True  True]
##  [False  True]
##  [ True  True]], shape=(3, 2), dtype=bool)
```

```
op_all(x, axis = 1)

## tf.Tensor([False  True], shape=(2), dtype=bool)
```

`keepdims = TRUE` outputs a tensor with dimensions reduced to one.

```
op_all(x, keepdims = TRUE)

## tf.Tensor([[False]], shape=(1, 1), dtype=bool)
```

See Also

- https://keras.io/api/ops/numpy#all-function

Other numpy ops:
op_abs()
op_add()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_bincount()
op_broadcast_to()
op_ceil()
op_clip()
op_concatenate()
op_conj()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_cumprod()
op_cumsum()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_einsum()
op_empty()
op_equal()
op_exp()
op_expand_dims()
op_expm1()
op_eye()

op_flip()
op_floor()
op_floor_divide()
op_full()
op_full_like()
op_get_item()
op_greater()
op_greater_equal()
op_hstack()
op_identity()
op_imag()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_matmul()
op_max()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moveaxis()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_not_equal()
op_ones()
op_ones_like()
op_outer()
op_pad()

op_power()
op_prod()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_repeat()
op_reshape()
op_roll()
op_round()
op_sign()
op_sin()
op_sinh()
op_size()
op_sort()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_subtract()
op_sum()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_var()
op_vdot()
op_vstack()
op_where()
op_zeros()
op_zeros_like()

Other ops:
op_abs()
op_add()
op_any()
op_append()
op_arange()

op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()

op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()

op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()

op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()

op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

---

op_any — *Test whether any array element along a given axis evaluates to* TRUE.

Description

Test whether any array element along a given axis evaluates to TRUE.

Usage

```
op_any(x, axis = NULL, keepdims = FALSE)
```

Arguments

x	Input tensor.
axis	An integer or tuple of integers that represent the axis along which a logical OR reduction is performed. The default (axis = NULL) is to perform a logical OR over all the dimensions of the input array. axis may be negative, in which case it counts for the last to the first axis.
keepdims	If TRUE, axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the input array. Defaults to FALSE.

Value

The tensor containing the logical OR reduction over the axis.

Examples

```
x <- op_array(c(TRUE, FALSE))
op_any(x)
```

```
## tf.Tensor(True, shape=(), dtype=bool)
```

```
(x <- op_reshape(c(FALSE, FALSE, FALSE,
                   TRUE, FALSE, FALSE),
                 c(2, 3)))
```

```
## tf.Tensor(
## [[False False False]
##  [ True False False]], shape=(2, 3), dtype=bool)
```

```
op_any(x, axis = 1)
```

```
## tf.Tensor([ True False False], shape=(3), dtype=bool)
```

```
op_any(x, axis = 2)
```

```
## tf.Tensor([False  True], shape=(2), dtype=bool)
```

```
op_any(x, axis = -1)
```

```
## tf.Tensor([False  True], shape=(2), dtype=bool)
```

keepdims = TRUE outputs a tensor with dimensions reduced to one.

```
op_any(x, keepdims = TRUE)
```

```
## tf.Tensor([[ True]], shape=(1, 1), dtype=bool)
```

```
op_any(x, 1, keepdims = TRUE)
```

```
## tf.Tensor([[ True False False]], shape=(1, 3), dtype=bool)
```

See Also

- `https://keras.io/api/ops/numpy#any-function`

Other numpy ops:
`op_abs()`
`op_add()`
`op_all()`
`op_append()`
`op_arange()`
`op_arccos()`
`op_arccosh()`
`op_arcsin()`
`op_arcsinh()`
`op_arctan()`
`op_arctan2()`
`op_arctanh()`
`op_argmax()`
`op_argmin()`
`op_argsort()`
`op_array()`
`op_average()`
`op_bincount()`
`op_broadcast_to()`
`op_ceil()`
`op_clip()`
`op_concatenate()`
`op_conj()`
`op_copy()`
`op_correlate()`
`op_cos()`
`op_cosh()`
`op_count_nonzero()`
`op_cross()`
`op_cumprod()`
`op_cumsum()`
`op_diag()`
`op_diagonal()`
`op_diff()`
`op_digitize()`
`op_divide()`
`op_divide_no_nan()`
`op_dot()`
`op_einsum()`
`op_empty()`
`op_equal()`
`op_exp()`
`op_expand_dims()`
`op_expm1()`
`op_eye()`

op_flip()
op_floor()
op_floor_divide()
op_full()
op_full_like()
op_get_item()
op_greater()
op_greater_equal()
op_hstack()
op_identity()
op_imag()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_matmul()
op_max()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moveaxis()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_not_equal()
op_ones()
op_ones_like()
op_outer()
op_pad()

op_power()
op_prod()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_repeat()
op_reshape()
op_roll()
op_round()
op_sign()
op_sin()
op_sinh()
op_size()
op_sort()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_subtract()
op_sum()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_var()
op_vdot()
op_vstack()
op_where()
op_zeros()
op_zeros_like()

Other ops:
op_abs()
op_add()
op_all()
op_append()
op_arange()

op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()

op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()

op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()

op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()

op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

op_append *Append tensor x2 to the end of tensor x1.*

Description

Append tensor x2 to the end of tensor x1.

Usage

```
op_append(x1, x2, axis = NULL)
```

Arguments

x1	First input tensor.
x2	Second input tensor.
axis	Axis along which tensor x2 is appended to tensor x1. If NULL, both tensors are flattened before use.

Value

A tensor with the values of x2 appended to x1.

Examples

```
x1 <- op_convert_to_tensor(c(1, 2, 3))
x2 <- op_convert_to_tensor(rbind(c(4, 5, 6), c(7, 8, 9)))
op_append(x1, x2)

## tf.Tensor([1. 2. 3. 4. 5. 6. 7. 8. 9.], shape=(9), dtype=float64)
```

When axis is specified, x1 and x2 must have compatible shapes.

```
x1 <- op_convert_to_tensor(rbind(c(1, 2, 3), c(4, 5, 6)))
x2 <- op_convert_to_tensor(rbind(c(7, 8, 9)))
op_append(x1, x2, axis = 1)
```

```
## tf.Tensor(
## [[1. 2. 3.]
##  [4. 5. 6.]
##  [7. 8. 9.]], shape=(3, 3), dtype=float64)
```

```
x3 <- op_convert_to_tensor(c(7, 8, 9))
try(op_append(x1, x3, axis = 1))
```

```
## Error in py_call_impl(callable, call_args$unnamed, call_args$named) :
##  tensorflow.python.framework.errors_impl.InvalidArgumentError: {{function_node __wrapped__ConcatV
```

See Also

- https://keras.io/api/ops/numpy#append-function

Other numpy ops:
op_abs()
op_add()
op_all()
op_any()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_bincount()
op_broadcast_to()
op_ceil()
op_clip()
op_concatenate()
op_conj()
op_copy()
op_correlate()

op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_cumprod()
op_cumsum()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_einsum()
op_empty()
op_equal()
op_exp()
op_expand_dims()
op_expm1()
op_eye()
op_flip()
op_floor()
op_floor_divide()
op_full()
op_full_like()
op_get_item()
op_greater()
op_greater_equal()
op_hstack()
op_identity()
op_imag()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()

op_matmul()
op_max()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moveaxis()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_not_equal()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_repeat()
op_reshape()
op_roll()
op_round()
op_sign()
op_sin()
op_sinh()
op_size()
op_sort()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_subtract()
op_sum()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()

op_tensordot()
op_tile()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_var()
op_vdot()
op_vstack()
op_where()
op_zeros()
op_zeros_like()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()

op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()

op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()

op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()

op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

---

op_arange *Return evenly spaced values within a given interval.*

Description

`arange` can be called with a varying number of positional arguments:

- `arange(stop)`: Values are generated within the half-open interval `[0, stop)` (in other words, the interval including `start` but excluding `stop`).
- `arange(start, stop)`: Values are generated within the half-open interval `[start, stop)`.
- `arange(start, stop, step)`: Values are generated within the half-open interval `[start, stop)`, with spacing between values given by step.

Usage

```
op_arange(start, stop = NULL, step = 1L, dtype = NULL)
```

Arguments

`start`	Integer or real, representing the start of the interval. The interval includes this value.
`stop`	Integer or real, representing the end of the interval. The interval does not include this value, except in some cases where `step` is not an integer and floating point round-off affects the length of `out`. Defaults to `NULL`.
`step`	Integer or real, represent the spacing between values. For any output `out`, this is the distance between two adjacent values, `out[i+1] - out[i]`. The default step size is 1. If `step` is specified as a position argument, `start` must also be given.
`dtype`	The type of the output array. If `dtype` is not given, infer the data type from the other input arguments.

Value

Tensor of evenly spaced values. For floating point arguments, the length of the result is `ceiling((stop - start)/step)`. Because of floating point overflow, this rule may result in the last element of out being greater than stop.

Examples

```
op_arange(3L)

## tf.Tensor([0 1 2], shape=(3), dtype=int32)


op_arange(3) # float

## tf.Tensor([0. 1. 2.], shape=(3), dtype=float64)


op_arange(3, dtype = 'int32') #int

## tf.Tensor([0 1 2], shape=(3), dtype=int32)


op_arange(3L, 7L)

## tf.Tensor([3 4 5 6], shape=(4), dtype=int32)


op_arange(3L, 7L, 2L)

## tf.Tensor([3 5], shape=(2), dtype=int32)
```

See Also

- https://keras.io/api/ops/numpy#arange-function

Other numpy ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_bincount()
op_broadcast_to()
op_ceil()
op_clip()
op_concatenate()
op_conj()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_cumprod()
op_cumsum()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_einsum()
op_empty()
op_equal()
op_exp()
op_expand_dims()
op_expm1()
op_eye()

op_flip()
op_floor()
op_floor_divide()
op_full()
op_full_like()
op_get_item()
op_greater()
op_greater_equal()
op_hstack()
op_identity()
op_imag()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_matmul()
op_max()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moveaxis()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_not_equal()
op_ones()
op_ones_like()
op_outer()
op_pad()

op_power()
op_prod()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_repeat()
op_reshape()
op_roll()
op_round()
op_sign()
op_sin()
op_sinh()
op_size()
op_sort()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_subtract()
op_sum()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_var()
op_vdot()
op_vstack()
op_where()
op_zeros()
op_zeros_like()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()

op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()

op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()

op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()

op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()

op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

op_arccos — *Trigonometric inverse cosine, element-wise.*

Description

The inverse of cos so that, if y = cos(x), then x = arccos(y).

Usage

```
op_arccos(x)
```

Arguments

x	Input tensor.

Value

Tensor of the angle of the ray intersecting the unit circle at the given x-coordinate in radians [0, pi].

Examples

```
x <- op_convert_to_tensor(c(1, -1))
op_arccos(x)


## tf.Tensor([0.        3.1415927], shape=(2), dtype=float32)
```

See Also

- https://keras.io/api/ops/numpy#arccos-function

Other numpy ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_bincount()
op_broadcast_to()
op_ceil()
op_clip()
op_concatenate()
op_conj()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_cumprod()
op_cumsum()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_einsum()
op_empty()
op_equal()
op_exp()
op_expand_dims()
op_expm1()
op_eye()

op_flip()
op_floor()
op_floor_divide()
op_full()
op_full_like()
op_get_item()
op_greater()
op_greater_equal()
op_hstack()
op_identity()
op_imag()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_matmul()
op_max()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moveaxis()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_not_equal()
op_ones()
op_ones_like()
op_outer()
op_pad()

op_power()
op_prod()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_repeat()
op_reshape()
op_roll()
op_round()
op_sign()
op_sin()
op_sinh()
op_size()
op_sort()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_subtract()
op_sum()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_var()
op_vdot()
op_vstack()
op_where()
op_zeros()
op_zeros_like()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()

op_arange()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()

op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()

op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()

op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()

op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

op_arccosh *Inverse hyperbolic cosine, element-wise.*

Description

Inverse hyperbolic cosine, element-wise.

Usage

```
op_arccosh(x)
```

Arguments

x	Input tensor.

Value

Output tensor of same shape as x.

Examples

```
x <- op_convert_to_tensor(c(10, 100))
op_arccosh(x)


## tf.Tensor([2.993223 5.298292], shape=(2), dtype=float32)
```

See Also

Other numpy ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_bincount()
op_broadcast_to()
op_ceil()
op_clip()
op_concatenate()
op_conj()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_cumprod()
op_cumsum()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_einsum()
op_empty()
op_equal()
op_exp()
op_expand_dims()
op_expm1()
op_eye()
op_flip()

op_floor()
op_floor_divide()
op_full()
op_full_like()
op_get_item()
op_greater()
op_greater_equal()
op_hstack()
op_identity()
op_imag()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_matmul()
op_max()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moveaxis()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_not_equal()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()

op_prod()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_repeat()
op_reshape()
op_roll()
op_round()
op_sign()
op_sin()
op_sinh()
op_size()
op_sort()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_subtract()
op_sum()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_var()
op_vdot()
op_vstack()
op_where()
op_zeros()
op_zeros_like()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()

op_arccos()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()

op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()

op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()

op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()

op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

op_arcsin *Inverse sine, element-wise.*

Description

Inverse sine, element-wise.

Usage

```
op_arcsin(x)
```

Arguments

x	Input tensor.

Value

Tensor of the inverse sine of each element in x, in radians and in the closed interval [-pi/2, pi/2].

Examples

```
x <- op_convert_to_tensor(c(1, -1, 0))
op_arcsin(x)


## tf.Tensor([ 1.5707964 -1.5707964  0.       ], shape=(3), dtype=float32)
```

See Also

- https://keras.io/api/ops/numpy#arcsin-function

Other numpy ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_bincount()
op_broadcast_to()
op_ceil()
op_clip()
op_concatenate()
op_conj()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_cumprod()
op_cumsum()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_einsum()
op_empty()
op_equal()
op_exp()
op_expand_dims()
op_expm1()
op_eye()

op_flip()
op_floor()
op_floor_divide()
op_full()
op_full_like()
op_get_item()
op_greater()
op_greater_equal()
op_hstack()
op_identity()
op_imag()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_matmul()
op_max()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moveaxis()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_not_equal()
op_ones()
op_ones_like()
op_outer()
op_pad()

op_power()
op_prod()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_repeat()
op_reshape()
op_roll()
op_round()
op_sign()
op_sin()
op_sinh()
op_size()
op_sort()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_subtract()
op_sum()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_var()
op_vdot()
op_vstack()
op_where()
op_zeros()
op_zeros_like()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()

op_arange()
op_arccos()
op_arccosh()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()

op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()

op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()

op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()

op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

---

op_arcsinh *Inverse hyperbolic sine, element-wise.*

Description

Inverse hyperbolic sine, element-wise.

Usage

```
op_arcsinh(x)
```

Arguments

x	Input tensor.

Value

Output tensor of same shape as x.

Examples

```
x <- op_convert_to_tensor(c(1, -1, 0))
op_arcsinh(x)
```

```
## tf.Tensor([ 0.8813736 -0.8813736  0.        ], shape=(3), dtype=float32)
```

See Also

- https://keras.io/api/ops/numpy#arcsinh-function

Other numpy ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_bincount()
op_broadcast_to()
op_ceil()
op_clip()
op_concatenate()
op_conj()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_cumprod()
op_cumsum()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_einsum()
op_empty()
op_equal()
op_exp()
op_expand_dims()
op_expm1()
op_eye()

op_flip()
op_floor()
op_floor_divide()
op_full()
op_full_like()
op_get_item()
op_greater()
op_greater_equal()
op_hstack()
op_identity()
op_imag()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_matmul()
op_max()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moveaxis()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_not_equal()
op_ones()
op_ones_like()
op_outer()
op_pad()

op_power()
op_prod()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_repeat()
op_reshape()
op_roll()
op_round()
op_sign()
op_sin()
op_sinh()
op_size()
op_sort()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_subtract()
op_sum()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_var()
op_vdot()
op_vstack()
op_where()
op_zeros()
op_zeros_like()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()

op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()

op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()

op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()

op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()

op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

---

op_arctan — *Trigonometric inverse tangent, element-wise.*

Description

Trigonometric inverse tangent, element-wise.

Usage

```
op_arctan(x)
```

Arguments

x	Input tensor.

Value

Tensor of the inverse tangent of each element in x, in the interval [-pi/2, pi/2].

Examples

```
x <- op_convert_to_tensor(c(0, 1))
op_arctan(x)
```

```
## tf.Tensor([0.        0.7853982], shape=(2), dtype=float32)
```

See Also

- https://keras.io/api/ops/numpy#arctan-function

Other numpy ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_bincount()
op_broadcast_to()
op_ceil()
op_clip()
op_concatenate()
op_conj()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_cumprod()
op_cumsum()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_einsum()
op_empty()
op_equal()
op_exp()
op_expand_dims()
op_expm1()
op_eye()

op_flip()
op_floor()
op_floor_divide()
op_full()
op_full_like()
op_get_item()
op_greater()
op_greater_equal()
op_hstack()
op_identity()
op_imag()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_matmul()
op_max()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moveaxis()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_not_equal()
op_ones()
op_ones_like()
op_outer()
op_pad()

op_power()
op_prod()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_repeat()
op_reshape()
op_roll()
op_round()
op_sign()
op_sin()
op_sinh()
op_size()
op_sort()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_subtract()
op_sum()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_var()
op_vdot()
op_vstack()
op_where()
op_zeros()
op_zeros_like()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()

op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()

op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()

op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()

op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()

op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

`op_arctan2` *Element-wise arc tangent of* `x1/x2` *choosing the quadrant correctly.*

Description

The quadrant (i.e., branch) is chosen so that `arctan2(x1, x2)` is the signed angle in radians between the ray ending at the origin and passing through the point `(1, 0)`, and the ray ending at the origin and passing through the point `(x2, x1)`. (Note the role reversal: the "y-coordinate" is the first function parameter, the "x-coordinate" is the second.) By IEEE convention, this function is defined for `x2 = +/-0` and for either or both of `x1` and `x2` = `+/-inf`.

Usage

```
op_arctan2(x1, x2)
```

Arguments

`x1`	First input tensor.
`x2`	Second input tensor.

Value

Tensor of angles in radians, in the range `[-pi, pi]`.

Examples

Consider four points in different quadrants:

```
x <- op_array(c(-1, 1, 1, -1))
y <- op_array(c(-1, -1, 1, 1))
op_arctan2(y, x) * 180 / pi
```

```
## tf.Tensor([-135.  -45.   45.  135.], shape=(4), dtype=float32)
```

Note the order of the parameters. `arctan2` is defined also when x2 = 0 and at several other points, obtaining values in the range `[-pi, pi]`:

```
op_arctan2(
    op_array(c(1, -1)),
    op_array(c(0, 0))
)

## tf.Tensor([ 1.5707964 -1.5707964], shape=(2), dtype=float32)


op_arctan2(
    op_array(c(0, 0, Inf)),
    op_array(c(+0, -0, Inf))
)

## tf.Tensor([0.        3.1415927 0.7853982], shape=(3), dtype=float32)
```

See Also

- https://keras.io/api/ops/numpy#arctan2-function

Other numpy ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_bincount()
op_broadcast_to()
op_ceil()
op_clip()
op_concatenate()

op_conj()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_cumprod()
op_cumsum()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_einsum()
op_empty()
op_equal()
op_exp()
op_expand_dims()
op_expm1()
op_eye()
op_flip()
op_floor()
op_floor_divide()
op_full()
op_full_like()
op_get_item()
op_greater()
op_greater_equal()
op_hstack()
op_identity()
op_imag()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_logaddexp()
op_logical_and()
op_logical_not()

op_logical_or()
op_logical_xor()
op_logspace()
op_matmul()
op_max()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moveaxis()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_not_equal()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_repeat()
op_reshape()
op_roll()
op_round()
op_sign()
op_sin()
op_sinh()
op_size()
op_sort()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_subtract()
op_sum()
op_swapaxes()
op_take()

op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_var()
op_vdot()
op_vstack()
op_where()
op_zeros()
op_zeros_like()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()

op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()

op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()

op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()

op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

op_arctanh *Inverse hyperbolic tangent, element-wise.*

Description

Inverse hyperbolic tangent, element-wise.

Usage

```
op_arctanh(x)
```

Arguments

x	Input tensor.

Value

Output tensor of same shape as x.

See Also

- https://keras.io/api/ops/numpy#arctanh-function

Other numpy ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_bincount()
op_broadcast_to()
op_ceil()
op_clip()
op_concatenate()
op_conj()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_cumprod()
op_cumsum()
op_diag()
op_diagonal()

op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_einsum()
op_empty()
op_equal()
op_exp()
op_expand_dims()
op_expm1()
op_eye()
op_flip()
op_floor()
op_floor_divide()
op_full()
op_full_like()
op_get_item()
op_greater()
op_greater_equal()
op_hstack()
op_identity()
op_imag()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_matmul()
op_max()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()

op_mod()
op_moveaxis()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_not_equal()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_repeat()
op_reshape()
op_roll()
op_round()
op_sign()
op_sin()
op_sinh()
op_size()
op_sort()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_subtract()
op_sum()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_var()

op_vdot()
op_vstack()
op_where()
op_zeros()
op_zeros_like()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()

op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()

op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()

op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()

op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

op_argmax *Returns the indices of the maximum values along an axis.*

Description

Returns the indices of the maximum values along an axis.

Usage

```
op_argmax(x, axis = NULL)
```

Arguments

x	Input tensor.
axis	By default, the index is into the flattened tensor, otherwise along the specified axis.

Value

Tensor of indices. It has the same shape as x, with the dimension along axis removed. Note that the returned integer is 0-based (i.e., if the argmax is in the first index position, the returned value will be 0)

Examples

```
x <- op_arange(6L) |> op_reshape(c(2, 3)) |> op_add(10)
x

## tf.Tensor(
## [[10. 11. 12.]
##  [13. 14. 15.]], shape=(2, 3), dtype=float32)


op_argmax(x)

## tf.Tensor(5, shape=(), dtype=int32)


op_argmax(x, axis = 1)

## tf.Tensor([1 1 1], shape=(3), dtype=int32)


op_argmax(x, axis = 2)

## tf.Tensor([2 2], shape=(2), dtype=int32)
```

Note

This is similar to R max.col(x) - 1 for the case of a 2-d array (a matrix), or for an nd-array, apply(x, axis, which.max) - 1

See Also

- https://keras.io/api/ops/numpy#argmax-function

Other numpy ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()

op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmin()
op_argsort()
op_array()
op_average()
op_bincount()
op_broadcast_to()
op_ceil()
op_clip()
op_concatenate()
op_conj()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_cumprod()
op_cumsum()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_einsum()
op_empty()
op_equal()
op_exp()
op_expand_dims()
op_expm1()
op_eye()
op_flip()
op_floor()
op_floor_divide()
op_full()
op_full_like()
op_get_item()
op_greater()
op_greater_equal()
op_hstack()
op_identity()
op_imag()
op_isclose()

op_isfinite()
op_isinf()
op_isnan()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_matmul()
op_max()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moveaxis()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_not_equal()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_repeat()
op_reshape()
op_roll()
op_round()
op_sign()
op_sin()

op_sinh()
op_size()
op_sort()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_subtract()
op_sum()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_var()
op_vdot()
op_vstack()
op_where()
op_zeros()
op_zeros_like()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmin()
op_argsort()
op_array()
op_average()

op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()

op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()

op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()

op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()

op_zeros()
op_zeros_like()

---

op_argmin *Returns the indices of the minimum values along an axis.*

Description

Returns the indices of the minimum values along an axis.

Usage

```
op_argmin(x, axis = NULL)
```

Arguments

x	Input tensor.
axis	By default, the index is into the flattened tensor, otherwise along the specified axis.

Value

Tensor of indices. It has the same shape as x, with the dimension along axis removed.

Examples

```
x <- op_arange(6L) |> op_reshape(c(2, 3)) |> op_add(10)
x

## tf.Tensor(
## [[10. 11. 12.]
##  [13. 14. 15.]], shape=(2, 3), dtype=float32)


op_argmin(x)

## tf.Tensor(0, shape=(), dtype=int32)


op_argmin(x, axis = 1)

## tf.Tensor([0 0 0], shape=(3), dtype=int32)


op_argmin(x, axis = 2)

## tf.Tensor([0 0], shape=(2), dtype=int32)
```

Note

This is similar to an R expression `apply(x, axis, which.min) - 1`, where `x` is a R array.

See Also

- `https://keras.io/api/ops/numpy#argmin-function`

Other numpy ops:
`op_abs()`
`op_add()`
`op_all()`
`op_any()`
`op_append()`
`op_arange()`
`op_arccos()`
`op_arccosh()`
`op_arcsin()`
`op_arcsinh()`
`op_arctan()`
`op_arctan2()`
`op_arctanh()`
`op_argmax()`
`op_argsort()`
`op_array()`
`op_average()`
`op_bincount()`
`op_broadcast_to()`
`op_ceil()`
`op_clip()`
`op_concatenate()`
`op_conj()`
`op_copy()`
`op_correlate()`
`op_cos()`
`op_cosh()`
`op_count_nonzero()`
`op_cross()`
`op_cumprod()`
`op_cumsum()`
`op_diag()`
`op_diagonal()`
`op_diff()`
`op_digitize()`
`op_divide()`
`op_divide_no_nan()`
`op_dot()`
`op_einsum()`
`op_empty()`
`op_equal()`

op_exp()
op_expand_dims()
op_expm1()
op_eye()
op_flip()
op_floor()
op_floor_divide()
op_full()
op_full_like()
op_get_item()
op_greater()
op_greater_equal()
op_hstack()
op_identity()
op_imag()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_matmul()
op_max()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moveaxis()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_not_equal()

op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_repeat()
op_reshape()
op_roll()
op_round()
op_sign()
op_sin()
op_sinh()
op_size()
op_sort()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_subtract()
op_sum()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_var()
op_vdot()
op_vstack()
op_where()
op_zeros()
op_zeros_like()

Other ops:
op_abs()

op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()

op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()

op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()

op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()

op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

op_argsort *Returns the indices that would sort a tensor.*

Description

Returns the indices that would sort a tensor.

Usage

```
op_argsort(x, axis = -1L)
```

Arguments

x	Input tensor.
axis	Axis along which to sort. Defaults to -1 (the last axis). If NULL, the flattened tensor is used.

Value

Tensor of indices that sort x along the specified axis.

Examples

One dimensional array:

```
x <- op_array(c(3, 1, 2))
op_argsort(x)
```

```
## tf.Tensor([1 2 0], shape=(3), dtype=int32)
```

Two-dimensional array:

```
x <- op_array(rbind(c(0, 3),
                    c(3, 2),
                    c(4, 5)), dtype = "int32")
op_argsort(x, axis = 1)

## tf.Tensor(
## [[0 1]
##  [1 0]
##  [2 2]], shape=(3, 2), dtype=int32)
```

```
op_argsort(x, axis = 2)

## tf.Tensor(
## [[0 1]
##  [1 0]
##  [0 1]], shape=(3, 2), dtype=int32)
```

See Also

- https://keras.io/api/ops/numpy#argsort-function

Other numpy ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_array()
op_average()
op_bincount()
op_broadcast_to()
op_ceil()

op_clip()
op_concatenate()
op_conj()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_cumprod()
op_cumsum()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_einsum()
op_empty()
op_equal()
op_exp()
op_expand_dims()
op_expm1()
op_eye()
op_flip()
op_floor()
op_floor_divide()
op_full()
op_full_like()
op_get_item()
op_greater()
op_greater_equal()
op_hstack()
op_identity()
op_imag()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_logaddexp()

op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_matmul()
op_max()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moveaxis()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_not_equal()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_repeat()
op_reshape()
op_roll()
op_round()
op_sign()
op_sin()
op_sinh()
op_size()
op_sort()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_subtract()
op_sum()

op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_var()
op_vdot()
op_vstack()
op_where()
op_zeros()
op_zeros_like()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()

op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()

op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()

op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()

op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

op_array *Create a tensor.*

Description

Create a tensor.

Usage

```
op_array(x, dtype = NULL)
```

Arguments

x	Input tensor.
dtype	The desired data-type for the tensor.

Value

A tensor.

Examples

```
op_array(c(1, 2, 3))

## tf.Tensor([1. 2. 3.], shape=(3), dtype=float32)


op_array(c(1, 2, 3), dtype = "float32")

## tf.Tensor([1. 2. 3.], shape=(3), dtype=float32)


op_array(c(1, 2, 3), dtype = "int32")

## tf.Tensor([1 2 3], shape=(3), dtype=int32)
```

See Also

- https://keras.io/api/ops/numpy#array-function

Other numpy ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()

op_argsort()
op_average()
op_bincount()
op_broadcast_to()
op_ceil()
op_clip()
op_concatenate()
op_conj()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_cumprod()
op_cumsum()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_einsum()
op_empty()
op_equal()
op_exp()
op_expand_dims()
op_expm1()
op_eye()
op_flip()
op_floor()
op_floor_divide()
op_full()
op_full_like()
op_get_item()
op_greater()
op_greater_equal()
op_hstack()
op_identity()
op_imag()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_less()
op_less_equal()
op_linspace()

op_log()
op_log10()
op_log1p()
op_log2()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_matmul()
op_max()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moveaxis()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_not_equal()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_repeat()
op_reshape()
op_roll()
op_round()
op_sign()
op_sin()
op_sinh()
op_size()
op_sort()
op_split()
op_sqrt()
op_square()

op_squeeze()
op_stack()
op_std()
op_subtract()
op_sum()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_var()
op_vdot()
op_vstack()
op_where()
op_zeros()
op_zeros_like()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()

op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()

op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()

op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()

op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

op_average *Compute the weighted average along the specified axis.*

Description

Compute the weighted average along the specified axis.

Usage

```
op_average(x, axis = NULL, weights = NULL)
```

Arguments

`x`	Input tensor.
`axis`	Integer along which to average `x`. The default, `axis = NULL`, will average over all of the elements of the input tensor. If axis is negative it counts from the last to the first axis.
`weights`	Tensor of wieghts associated with the values in `x`. Each value in `x` contributes to the average according to its associated weight. The weights array can either be 1-D (in which case its length must be the size of a along the given axis) or of the same shape as `x`. If `weights = NULL` (default), then all data in `x` are assumed to have a weight equal to one. The 1-D calculation is: `avg = sum(a * weights) / sum(weights)`. The only constraint on weights is that `sum(weights)` must not be 0.

Value

Return the average along the specified axis.

Examples

```
data <- op_arange(1, 5, dtype = "int32")
data

## tf.Tensor([1 2 3 4], shape=(4), dtype=int32)


op_average(data)

## tf.Tensor(2.5, shape=(), dtype=float32)


op_average(
  op_arange(1, 11),
  weights = op_arange(10, 0, -1)
)

## tf.Tensor(4.0, shape=(), dtype=float64)


data <- op_arange(6) |> op_reshape(c(3, 2))
data
```

```
## tf.Tensor(
## [[0. 1.]
##  [2. 3.]
##  [4. 5.]], shape=(3, 2), dtype=float64)
```

```
op_average(
  data,
  axis = 2,
  weights = op_array(c(1/4, 3/4))
)
```

```
## tf.Tensor([0.75 2.75 4.75], shape=(3), dtype=float64)
```

```
# Error: Axis must be specified when shapes of a and weights differ.
try(op_average(
  data,
  weights = op_array(c(1/4, 3/4))
))
```

```
## Error in py_call_impl(callable, call_args$unnamed, call_args$named) :
##  tensorflow.python.framework.errors_impl.InvalidArgumentError: Expected 'tf.Tensor(False, shape=(
## 2
```

See Also

- https://keras.io/api/ops/numpy#average-function

Other numpy ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_bincount()

op_broadcast_to()
op_ceil()
op_clip()
op_concatenate()
op_conj()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_cumprod()
op_cumsum()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_einsum()
op_empty()
op_equal()
op_exp()
op_expand_dims()
op_expm1()
op_eye()
op_flip()
op_floor()
op_floor_divide()
op_full()
op_full_like()
op_get_item()
op_greater()
op_greater_equal()
op_hstack()
op_identity()
op_imag()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()

op_log2()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_matmul()
op_max()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moveaxis()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_not_equal()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_repeat()
op_reshape()
op_roll()
op_round()
op_sign()
op_sin()
op_sinh()
op_size()
op_sort()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()

op_subtract()
op_sum()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_var()
op_vdot()
op_vstack()
op_where()
op_zeros()
op_zeros_like()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()

op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()

op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()

op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()

op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

op_average_pool *Average pooling operation.*

Description

Average pooling operation.

Usage

```
op_average_pool(
  inputs,
  pool_size,
  strides = NULL,
  padding = "valid",
  data_format = NULL
)
```

Arguments

`inputs`	Tensor of rank N+2. `inputs` has shape `(batch_size,) + inputs_spatial_shape + (num_channels,)` if `data_format = "channels_last"`, or `(batch_size, num_channels) + inputs_spatial_shape` if `data_format = "channels_first"`. Pooling happens over the spatial dimensions only.
`pool_size`	int or tuple/list of integers of size `len(inputs_spatial_shape)`, specifying the size of the pooling window for each spatial dimension of the input tensor. If `pool_size` is int, then every spatial dimension shares the same `pool_size`.
`strides`	int or tuple/list of integers of size `len(inputs_spatial_shape)`. The stride of the sliding window for each spatial dimension of the input tensor. If `strides` is int, then every spatial dimension shares the same `strides`.
`padding`	string, either `"valid"` or `"same"`. `"valid"` means no padding is applied, and `"same"` results in padding evenly to the left/right or up/down of the input such that output has the same height/width dimension as the input when `strides = 1`.
`data_format`	A string, either `"channels_last"` or `"channels_first"`. `data_format` determines the ordering of the dimensions in the inputs. If `data_format = "channels_last"`, `inputs` is of shape `(batch_size, ..., channels)` while if `data_format = "channels_first"`, `inputs` is of shape `(batch_size, channels, ...)`.

Value

A tensor of rank N+2, the result of the average pooling operation.

See Also

- `https://keras.io/api/ops/nn#averagepool-function`

Other nn ops:
`op_batch_normalization()`
`op_binary_crossentropy()`
`op_categorical_crossentropy()`
`op_conv()`
`op_conv_transpose()`

op_ctc_loss()
op_depthwise_conv()
op_elu()
op_gelu()
op_hard_sigmoid()
op_hard_silu()
op_leaky_relu()
op_log_sigmoid()
op_log_softmax()
op_max_pool()
op_moments()
op_multi_hot()
op_normalize()
op_one_hot()
op_relu()
op_relu6()
op_selu()
op_separable_conv()
op_sigmoid()
op_silu()
op_softmax()
op_softplus()
op_softsign()
op_sparse_categorical_crossentropy()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()

op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()

op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()

op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()

op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

op_batch_normalization

Normalizes `x` *by* `mean` *and* `variance`*.*

Description

This op is typically used by the batch normalization step in a neural network. It normalizes the input tensor along the given axis.

Usage

```
op_batch_normalization(
  x,
  mean,
  variance,
  axis,
  offset = NULL,
  scale = NULL,
  epsilon = 0.001
)
```

Arguments

`x`	Input tensor.
`mean`	A mean vector of the same length as the `axis` dimension of the input thensor.
`variance`	A variance vector of the same length as the `axis` dimension of the input tensor.
`axis`	Integer, the axis that should be normalized.
`offset`	An offset vector of the same length as the `axis` dimension of the input tensor. If not `NULL`, `offset` is added to the normalized tensor. Defaults to `NULL`.
`scale`	A scale vector of the same length as the `axis` dimension of the input tensor. If not `NULL`, the normalized tensor is multiplied by `scale`. Defaults to `NULL`.
`epsilon`	Small float added to variance to avoid dividing by zero. Defaults to 1e-3.

Value

The normalized tensor.

Examples

```
x <- op_convert_to_tensor(rbind(c(0.1, 0.2, 0.3),
                                c(0.4, 0.5, 0.6),
                                c(0.7, 0.8, 0.9)))
op_batch_normalization(
  x,
  mean = c(0.4, 0.5, 0.6),
  variance = c(0.67, 0.67, 0.67),
  axis = -1
)

## tf.Tensor(
## [[-0.36623513 -0.36623513 -0.36623513]
##  [ 0.          0.          0.        ]
##  [ 0.36623513  0.36623513  0.36623513]], shape=(3, 3), dtype=float64)
```

See Also

- https://www.tensorflow.org/api_docs/python/tf/keras/ops/batch_normalization

Other nn ops:
op_average_pool()
op_binary_crossentropy()
op_categorical_crossentropy()
op_conv()
op_conv_transpose()
op_ctc_loss()
op_depthwise_conv()
op_elu()
op_gelu()
op_hard_sigmoid()
op_hard_silu()
op_leaky_relu()
op_log_sigmoid()
op_log_softmax()
op_max_pool()
op_moments()
op_multi_hot()
op_normalize()
op_one_hot()
op_relu()
op_relu6()
op_selu()
op_separable_conv()
op_sigmoid()
op_silu()
op_softmax()
op_softplus()
op_softsign()
op_sparse_categorical_crossentropy()

Other ops:
op_abs()
op_add()
op_all()

op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()

op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()

op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()

op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()

op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

---

op_binary_crossentropy

Computes binary cross-entropy loss between target and output tensor.

Description

The binary cross-entropy loss is commonly used in binary classification tasks where each input sample belongs to one of the two classes. It measures the dissimilarity between the target and output probabilities or logits.

Usage

```
op_binary_crossentropy(target, output, from_logits = FALSE)
```

Arguments

`target`	The target tensor representing the true binary labels. Its shape should match the shape of the `output` tensor.
`output`	The output tensor representing the predicted probabilities or logits. Its shape should match the shape of the `target` tensor.
`from_logits`	(optional) Whether `output` is a tensor of logits or probabilities. Set it to `TRUE` if `output` represents logits; otherwise, set it to `FALSE` if `output` represents probabilities. Defaults to `FALSE`.

Value

Integer tensor: The computed binary cross-entropy loss between `target` and `output`.

Examples

```
target <- op_array(c(0, 1, 1, 0))
output <- op_array(c(0.1, 0.9, 0.8, 0.2))
op_binary_crossentropy(target, output)

## tf.Tensor([0.10536055 0.10536055 0.22314353 0.22314353], shape=(4), dtype=float32)
```

See Also

- https://keras.io/api/ops/nn#binarycrossentropy-function

Other nn ops:
op_average_pool()
op_batch_normalization()
op_categorical_crossentropy()
op_conv()
op_conv_transpose()
op_ctc_loss()
op_depthwise_conv()
op_elu()
op_gelu()
op_hard_sigmoid()
op_hard_silu()
op_leaky_relu()
op_log_sigmoid()
op_log_softmax()
op_max_pool()
op_moments()
op_multi_hot()
op_normalize()
op_one_hot()
op_relu()
op_relu6()
op_selu()
op_separable_conv()
op_sigmoid()
op_silu()
op_softmax()
op_softplus()
op_softsign()
op_sparse_categorical_crossentropy()

Other ops:
op_abs()
op_add()
op_all()
op_any()

op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()

op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()

op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()

op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()

op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

op_bincount *Count the number of occurrences of each value in a tensor of integers.*

Description

Count the number of occurrences of each value in a tensor of integers.

Usage

```
op_bincount(x, weights = NULL, minlength = 0L, sparse = FALSE)
```

Arguments

x	Input tensor. It must be of dimension 1, and it must only contain non-negative integer(s).
weights	Weight tensor. It must have the same length as x. The default value is NULL. If specified, x is weighted by it, i.e. if n = x[i], out[n] += weight[i] instead of the default behavior out[n] += 1.
minlength	An integer. The default value is 0. If specified, there will be at least this number of bins in the output tensor. If greater than max(x) + 1, each value of the output at an index higher than max(x) is set to 0.
sparse	Whether to return a sparse tensor; for backends that support sparse tensors.

Value

1D tensor where each element gives the number of occurrence(s) of its index value in x. Its length is the maximum between max(x) + 1 and minlength.

Examples

```
(x <- op_array(c(1, 2, 2, 3), dtype = "uint8"))

## tf.Tensor([1 2 2 3], shape=(4), dtype=uint8)


op_bincount(x)

## tf.Tensor([0 1 2 1], shape=(4), dtype=int32)


(weights <- x / 2)

## tf.Tensor([0.5 1.  1.  1.5], shape=(4), dtype=float32)


op_bincount(x, weights = weights)

## tf.Tensor([0.  0.5 2.  1.5], shape=(4), dtype=float32)


minlength <- as.integer(op_max(x) + 1 + 2) # 6
op_bincount(x, minlength = minlength)

## tf.Tensor([0 1 2 1 0 0], shape=(6), dtype=int32)
```

See Also

- https://keras.io/api/ops/numpy#bincount-function

Other numpy ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()

op_array()
op_average()
op_broadcast_to()
op_ceil()
op_clip()
op_concatenate()
op_conj()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_cumprod()
op_cumsum()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_einsum()
op_empty()
op_equal()
op_exp()
op_expand_dims()
op_expm1()
op_eye()
op_flip()
op_floor()
op_floor_divide()
op_full()
op_full_like()
op_get_item()
op_greater()
op_greater_equal()
op_hstack()
op_identity()
op_imag()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_less()
op_less_equal()
op_linspace()
op_log()

op_log10()
op_log1p()
op_log2()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_matmul()
op_max()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moveaxis()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_not_equal()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_repeat()
op_reshape()
op_roll()
op_round()
op_sign()
op_sin()
op_sinh()
op_size()
op_sort()
op_split()
op_sqrt()
op_square()
op_squeeze()

op_stack()
op_std()
op_subtract()
op_sum()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_var()
op_vdot()
op_vstack()
op_where()
op_zeros()
op_zeros_like()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()

op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()

op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()

op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()

op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

op_broadcast_to *Broadcast a tensor to a new shape.*

Description

Broadcast a tensor to a new shape.

Usage

```
op_broadcast_to(x, shape)
```

Arguments

x	The tensor to broadcast.
shape	The shape of the desired tensor.

Value

A tensor with the desired shape.

Examples

```
x <- op_array(c(1, 2, 3))
op_broadcast_to(x, shape = c(3, 3))

## tf.Tensor(
## [[1. 2. 3.]
##  [1. 2. 3.]
##  [1. 2. 3.]], shape=(3, 3), dtype=float32)
```

See Also

- https://keras.io/api/ops/numpy#broadcastto-function

Other numpy ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()

op_average()
op_bincount()
op_ceil()
op_clip()
op_concatenate()
op_conj()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_cumprod()
op_cumsum()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_einsum()
op_empty()
op_equal()
op_exp()
op_expand_dims()
op_expm1()
op_eye()
op_flip()
op_floor()
op_floor_divide()
op_full()
op_full_like()
op_get_item()
op_greater()
op_greater_equal()
op_hstack()
op_identity()
op_imag()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()

op_log1p()
op_log2()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_matmul()
op_max()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moveaxis()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_not_equal()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_repeat()
op_reshape()
op_roll()
op_round()
op_sign()
op_sin()
op_sinh()
op_size()
op_sort()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()

op_std()
op_subtract()
op_sum()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_var()
op_vdot()
op_vstack()
op_where()
op_zeros()
op_zeros_like()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_cast()
op_categorical_crossentropy()
op_ceil()

op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()

op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()

op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()

op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

op_cast   *Cast a tensor to the desired dtype.*

Description

Cast a tensor to the desired dtype.

Usage

```
op_cast(x, dtype)
```

Arguments

x	A tensor or variable.
dtype	The target type.

Value

A tensor of the specified dtype.

Examples

```
(x <- op_arange(4))

## tf.Tensor([0. 1. 2. 3.], shape=(4), dtype=float64)


op_cast(x, dtype = "float16")

## tf.Tensor([0. 1. 2. 3.], shape=(4), dtype=float16)
```

See Also

- https://keras.io/api/ops/core#cast-function

Other core ops:
op_cond()
op_convert_to_numpy()
op_convert_to_tensor()
op_custom_gradient()
op_fori_loop()
op_is_tensor()
op_scatter()
op_scatter_update()
op_shape()
op_slice()
op_slice_update()
op_stop_gradient()
op_unstack()
op_vectorized_map()
op_while_loop()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()

op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()

op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()

op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()

op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

---

op_categorical_crossentropy

Computes categorical cross-entropy loss between target and output tensor.

Description

The categorical cross-entropy loss is commonly used in multi-class classification tasks where each input sample can belong to one of multiple classes. It measures the dissimilarity between the target and output probabilities or logits.

Usage

```
op_categorical_crossentropy(target, output, from_logits = FALSE, axis = -1L)
```

Arguments

`target`	The target tensor representing the true categorical labels. Its shape should match the shape of the `output` tensor except for the last dimension.
`output`	The output tensor representing the predicted probabilities or logits. Its shape should match the shape of the `target` tensor except for the last dimension.
`from_logits`	(optional) Whether `output` is a tensor of logits or probabilities. Set it to `TRUE` if `output` represents logits; otherwise, set it to `FALSE` if `output` represents probabilities. Defaults to `FALSE`.
`axis`	(optional) The axis along which the categorical cross-entropy is computed. Defaults to `-1`, which corresponds to the last dimension of the tensors.

Value

Integer tensor: The computed categorical cross-entropy loss between `target` and `output`.

Examples

```
target <- op_array(rbind(c(1, 0, 0),
                         c(0, 1, 0),
                         c(0, 0, 1)))
output <- op_array(rbind(c(0.9, 0.05, 0.05),
                         c(0.1, 0.8, 0.1),
                         c(0.2, 0.3, 0.5)))
op_categorical_crossentropy(target, output)

## tf.Tensor([0.10536052 0.22314355 0.69314718], shape=(3), dtype=float64)
```

See Also

- https://keras.io/api/ops/nn#categoricalcrossentropy-function

Other nn ops:
`op_average_pool()`
`op_batch_normalization()`
`op_binary_crossentropy()`
`op_conv()`
`op_conv_transpose()`
`op_ctc_loss()`
`op_depthwise_conv()`
`op_elu()`
`op_gelu()`
`op_hard_sigmoid()`
`op_hard_silu()`
`op_leaky_relu()`
`op_log_sigmoid()`
`op_log_softmax()`
`op_max_pool()`
`op_moments()`
`op_multi_hot()`
`op_normalize()`
`op_one_hot()`
`op_relu()`
`op_relu6()`
`op_selu()`
`op_separable_conv()`
`op_sigmoid()`
`op_silu()`
`op_softmax()`
`op_softplus()`
`op_softsign()`

op_sparse_categorical_crossentropy()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()

op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()

op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()

op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()

op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

---

op_ceil *Return the ceiling of the input, element-wise.*

Description

The ceil of the scalar x is the smallest integer i, such that i >= x.

Usage

```
op_ceil(x)
```

Arguments

x	Input tensor.

Value

The ceiling of each element in x, with float dtype.

See Also

- https://keras.io/api/ops/numpy#ceil-function

Other numpy ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_bincount()
op_broadcast_to()
op_clip()
op_concatenate()
op_conj()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_cumprod()
op_cumsum()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_einsum()
op_empty()
op_equal()
op_exp()
op_expand_dims()
op_expm1()
op_eye()

op_flip()
op_floor()
op_floor_divide()
op_full()
op_full_like()
op_get_item()
op_greater()
op_greater_equal()
op_hstack()
op_identity()
op_imag()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_matmul()
op_max()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moveaxis()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_not_equal()
op_ones()
op_ones_like()
op_outer()
op_pad()

op_power()
op_prod()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_repeat()
op_reshape()
op_roll()
op_round()
op_sign()
op_sin()
op_sinh()
op_size()
op_sort()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_subtract()
op_sum()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_var()
op_vdot()
op_vstack()
op_where()
op_zeros()
op_zeros_like()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()

op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()

op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()

op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()

op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()

op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

---

op_cholesky *Computes the Cholesky decomposition of a positive semi-definite matrix.*

Description

Computes the Cholesky decomposition of a positive semi-definite matrix.

Usage

```
op_cholesky(x)
```

Arguments

x	Input tensor of shape `(..., M, M)`.

Value

A tensor of shape `(..., M, M)` representing the lower triangular Cholesky factor of `x`.

See Also

Other linear algebra ops:
op_det()
op_eig()
op_inv()
op_lu_factor()
op_norm()
op_solve_triangular()
op_svd()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()

op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()

op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()

op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()

op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

---

op_clip *Clip (limit) the values in a tensor.*

Description

Given an interval, values outside the interval are clipped to the interval edges. For example, if an interval of `[0, 1]` is specified, values smaller than 0 become 0, and values larger than 1 become 1.

Usage

```
op_clip(x, x_min, x_max)
```

Arguments

`x`	Input tensor.
`x_min`	Minimum value.
`x_max`	Maximum value.

Value

The clipped tensor.

See Also

- https://keras.io/api/ops/numpy#clip-function

Other numpy ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_bincount()
op_broadcast_to()
op_ceil()
op_concatenate()
op_conj()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_cumprod()
op_cumsum()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_einsum()
op_empty()
op_equal()
op_exp()
op_expand_dims()
op_expm1()
op_eye()

op_flip()
op_floor()
op_floor_divide()
op_full()
op_full_like()
op_get_item()
op_greater()
op_greater_equal()
op_hstack()
op_identity()
op_imag()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_matmul()
op_max()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moveaxis()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_not_equal()
op_ones()
op_ones_like()
op_outer()
op_pad()

op_power()
op_prod()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_repeat()
op_reshape()
op_roll()
op_round()
op_sign()
op_sin()
op_sinh()
op_size()
op_sort()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_subtract()
op_sum()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_var()
op_vdot()
op_vstack()
op_where()
op_zeros()
op_zeros_like()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()

op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()

op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()

op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()

op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()

op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

`op_concatenate` *Join a sequence of tensors along an existing axis.*

Description

Join a sequence of tensors along an existing axis.

Usage

```
op_concatenate(xs, axis = 1L)
```

Arguments

`xs`	The sequence of tensors to concatenate.
`axis`	The axis along which the tensors will be joined. Defaults to `0`.

Value

The concatenated tensor.

See Also

- https://keras.io/api/ops/numpy#concatenate-function

Other numpy ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()

op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_bincount()
op_broadcast_to()
op_ceil()
op_clip()
op_conj()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_cumprod()
op_cumsum()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_einsum()
op_empty()
op_equal()
op_exp()
op_expand_dims()
op_expm1()
op_eye()
op_flip()
op_floor()
op_floor_divide()
op_full()
op_full_like()
op_get_item()
op_greater()
op_greater_equal()
op_hstack()
op_identity()

op_imag()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_matmul()
op_max()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moveaxis()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_not_equal()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_repeat()
op_reshape()
op_roll()
op_round()

op_sign()
op_sin()
op_sinh()
op_size()
op_sort()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_subtract()
op_sum()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_var()
op_vdot()
op_vstack()
op_where()
op_zeros()
op_zeros_like()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()

op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()

op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()

op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()

op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()

op_where()
op_while_loop()
op_zeros()
op_zeros_like()

---

op_cond *Conditionally applies* true_fn *or* false_fn.

Description

Conditionally applies true_fn or false_fn.

Usage

```
op_cond(pred, true_fn, false_fn)
```

Arguments

pred	Boolean scalar type
true_fn	Callable returning the output for the pred == TRUE case.
false_fn	Callable returning the output for the pred == FALSE case.

Value

The output of either true_fn or false_fn depending on pred.

Examples

```
fn <- tensorflow::tf_function(function(x) {
  op_cond(x > 0,
    true_fn = \() x + 1,
    false_fn = \() x - 1)
})

fn(tensorflow::as_tensor(1))

## tf.Tensor(2.0, shape=(), dtype=float64)


fn(tensorflow::as_tensor(-1))

## tf.Tensor(-2.0, shape=(), dtype=float64)
```

```
#
# Conditional side-effect (print only, no return value).
file <- tempfile(fileext = ".txt")
fn <- tensorflow::tf_function(function(epochs) {
  op_fori_loop(
    0, epochs,
    body_fun = \(epoch, state) {
      op_cond(epoch %% 20 == 0,
              \() {
                tensorflow::tf$print(
                  "epoch:", epoch,
                  output_stream = paste0("file://", file))
                NULL
              },
              \() {NULL})
      state
    },
    init_val = tensorflow::as_tensor(0))
})

fn(tensorflow::as_tensor(100))

## tf.Tensor(0.0, shape=(), dtype=float64)


readLines(file)

## [1] "epoch: 0"  "epoch: 20" "epoch: 40" "epoch: 60" "epoch: 80"


# cleanup
unlink(file)
```

See Also

Other core ops:
op_cast()
op_convert_to_numpy()
op_convert_to_tensor()
op_custom_gradient()
op_fori_loop()
op_is_tensor()
op_scatter()
op_scatter_update()
op_shape()
op_slice()
op_slice_update()
op_stop_gradient()

op_unstack()
op_vectorized_map()
op_while_loop()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()

op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()

op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()

op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()

`op_sum()`
`op_svd()`
`op_swapaxes()`
`op_take()`
`op_take_along_axis()`
`op_tan()`
`op_tanh()`
`op_tensordot()`
`op_tile()`
`op_top_k()`
`op_trace()`
`op_transpose()`
`op_tri()`
`op_tril()`
`op_triu()`
`op_unstack()`
`op_var()`
`op_vdot()`
`op_vectorized_map()`
`op_vstack()`
`op_where()`
`op_while_loop()`
`op_zeros()`
`op_zeros_like()`

`op_conj` *Returns the complex conjugate, element-wise.*

Description

The complex conjugate of a complex number is obtained by changing the sign of its imaginary part.

Usage

```
op_conj(x)
```

Arguments

`x`	Input tensor.

Value

The complex conjugate of each element in `x`.

See Also

- `https://keras.io/api/ops/numpy#conjugate-function`

Other numpy ops:
`op_abs()`
`op_add()`
`op_all()`
`op_any()`
`op_append()`
`op_arange()`
`op_arccos()`
`op_arccosh()`
`op_arcsin()`
`op_arcsinh()`
`op_arctan()`
`op_arctan2()`
`op_arctanh()`
`op_argmax()`
`op_argmin()`
`op_argsort()`
`op_array()`
`op_average()`
`op_bincount()`
`op_broadcast_to()`
`op_ceil()`
`op_clip()`
`op_concatenate()`
`op_copy()`
`op_correlate()`
`op_cos()`
`op_cosh()`
`op_count_nonzero()`
`op_cross()`
`op_cumprod()`
`op_cumsum()`
`op_diag()`
`op_diagonal()`
`op_diff()`
`op_digitize()`
`op_divide()`
`op_divide_no_nan()`
`op_dot()`
`op_einsum()`
`op_empty()`
`op_equal()`
`op_exp()`
`op_expand_dims()`
`op_expm1()`
`op_eye()`

op_flip()
op_floor()
op_floor_divide()
op_full()
op_full_like()
op_get_item()
op_greater()
op_greater_equal()
op_hstack()
op_identity()
op_imag()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_matmul()
op_max()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moveaxis()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_not_equal()
op_ones()
op_ones_like()
op_outer()
op_pad()

op_power()
op_prod()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_repeat()
op_reshape()
op_roll()
op_round()
op_sign()
op_sin()
op_sinh()
op_size()
op_sort()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_subtract()
op_sum()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_var()
op_vdot()
op_vstack()
op_where()
op_zeros()
op_zeros_like()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()

op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()

op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()

op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()

op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()

op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

---

op_conv *General N-D convolution.*

Description

This ops supports 1D, 2D and 3D convolution.

Usage

```
op_conv(
  inputs,
  kernel,
  strides = 1L,
  padding = "valid",
  data_format = NULL,
  dilation_rate = 1L
)
```

Arguments

`inputs`	Tensor of rank N+2. `inputs` has shape `(batch_size,) + inputs_spatial_shape + (num_channels,)` if `data_format = "channels_last"`, or `(batch_size, num_channels) + inputs_spatial_shape` if `data_format = "channels_first"`.
`kernel`	Tensor of rank N+2. `kernel` has shape `(kernel_spatial_shape, num_input_channels, num_output_` `num_input_channels` should match the number of channels in `inputs`.
`strides`	int or int tuple/list of `len(inputs_spatial_shape)`, specifying the strides of the convolution along each spatial dimension. If `strides` is int, then every spatial dimension shares the same `strides`.

padding	string, either "valid" or "same". "valid" means no padding is applied, and "same" results in padding evenly to the left/right or up/down of the input such that output has the same height/width dimension as the input when strides = 1.
data_format	A string, either "channels_last" or "channels_first". data_format determines the ordering of the dimensions in the inputs. If data_format = "channels_last", inputs is of shape (batch_size, ..., channels) while if data_format = "channels_first", inputs is of shape (batch_size, channels, ...).
dilation_rate	int or int tuple/list of len(inputs_spatial_shape), specifying the dilation rate to use for dilated convolution. If dilation_rate is int, then every spatial dimension shares the same dilation_rate.

Value

A tensor of rank N+2, the result of the conv operation.

See Also

- https://keras.io/api/ops/nn#conv-function

Other nn ops:
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_categorical_crossentropy()
op_conv_transpose()
op_ctc_loss()
op_depthwise_conv()
op_elu()
op_gelu()
op_hard_sigmoid()
op_hard_silu()
op_leaky_relu()
op_log_sigmoid()
op_log_softmax()
op_max_pool()
op_moments()
op_multi_hot()
op_normalize()
op_one_hot()
op_relu()
op_relu6()
op_selu()
op_separable_conv()
op_sigmoid()
op_silu()
op_softmax()
op_softplus()
op_softsign()

op_sparse_categorical_crossentropy()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()

op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()

op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()

op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()

op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

op_convert_to_numpy *Convert a tensor to a NumPy array.*

Description

Convert a tensor to a NumPy array.

Usage

```
op_convert_to_numpy(x)
```

Arguments

x	A tensor.

Value

A NumPy array.

See Also

- https://keras.io/api/ops/core#converttonumpy-function

Other core ops:
op_cast()
op_cond()
op_convert_to_tensor()
op_custom_gradient()
op_fori_loop()
op_is_tensor()
op_scatter()
op_scatter_update()
op_shape()
op_slice()
op_slice_update()
op_stop_gradient()
op_unstack()
op_vectorized_map()
op_while_loop()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()

op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()

op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()

op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()

op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

`op_convert_to_tensor` *Convert an array to a tensor.*

Description

Convert an array to a tensor.

Usage

```
op_convert_to_tensor(x, dtype = NULL, sparse = NULL)
```

Arguments

`x`	An array.
`dtype`	The target type.
`sparse`	Whether to keep sparse tensors. `FALSE` will cause sparse tensors to be densified. The default value of `NULL` means that sparse tensors are kept only if the backend supports them.

Value

A tensor of the specified `dtype`.

Examples

```
x <- array(c(1, 2, 3))
y <- op_convert_to_tensor(x)
y

## tf.Tensor([1. 2. 3.], shape=(3), dtype=float64)


op_convert_to_tensor(c(1, 3, 2, 0), "int32")

## tf.Tensor([1 3 2 0], shape=(4), dtype=int32)
```

See Also

- `op_array()`
- `https://keras.io/api/ops/core#converttotensor-function`

Other core ops:
`op_cast()`
`op_cond()`
`op_convert_to_numpy()`
`op_custom_gradient()`
`op_fori_loop()`
`op_is_tensor()`
`op_scatter()`
`op_scatter_update()`
`op_shape()`
`op_slice()`
`op_slice_update()`
`op_stop_gradient()`
`op_unstack()`

op_vectorized_map()
op_while_loop()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()

op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()

op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()

op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()

op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

---

op_conv_transpose *General N-D convolution transpose.*

Description

Also known as de-convolution. This ops supports 1D, 2D and 3D convolution.

Usage

```
op_conv_transpose(
  inputs,
  kernel,
  strides,
  padding = "valid",
  output_padding = NULL,
  data_format = NULL,
  dilation_rate = 1L
)
```

Arguments

inputs	Tensor of rank N+2. inputs has shape (batch_size,) + inputs_spatial_shape + (num_channels,) if data_format = "channels_last", or (batch_size, num_channels) + inputs_spatial_shape if data_format = "channels_first".
kernel	Tensor of rank N+2. kernel has shape [kernel_spatial_shape, num_output_channels, num_input_ num_input_channels should match the number of channels in inputs.
strides	int or int tuple/list of len(inputs_spatial_shape), specifying the strides of the convolution along each spatial dimension. If strides is int, then every spatial dimension shares the same strides.
padding	string, either "valid" or "same". "valid" means no padding is applied, and "same" results in padding evenly to the left/right or up/down of the input such that output has the same height/width dimension as the input when strides = 1.
output_padding	int or int tuple/list of len(inputs_spatial_shape), specifying the amount of padding along the height and width of the output tensor. Can be a single integer to specify the same value for all spatial dimensions. The amount of output padding along a given dimension must be lower than the stride along that same dimension. If set to NULL (default), the output shape is inferred.
data_format	A string, either "channels_last" or "channels_first". data_format determines the ordering of the dimensions in the inputs. If data_format = "channels_last", inputs is of shape (batch_size, ..., channels) while if data_format = "channels_first", inputs is of shape (batch_size, channels, ...).
dilation_rate	int or int tuple/list of len(inputs_spatial_shape), specifying the dilation rate to use for dilated convolution. If dilation_rate is int, then every spatial dimension shares the same dilation_rate.

Value

A tensor of rank N+2, the result of the conv operation.

See Also

- https://keras.io/api/ops/nn#convtranspose-function

Other nn ops:
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_categorical_crossentropy()
op_conv()
op_ctc_loss()
op_depthwise_conv()
op_elu()
op_gelu()
op_hard_sigmoid()
op_hard_silu()
op_leaky_relu()

op_log_sigmoid()
op_log_softmax()
op_max_pool()
op_moments()
op_multi_hot()
op_normalize()
op_one_hot()
op_relu()
op_relu6()
op_selu()
op_separable_conv()
op_sigmoid()
op_silu()
op_softmax()
op_softplus()
op_softsign()
op_sparse_categorical_crossentropy()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()

op_cond()
op_conj()
op_conv()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()

op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()

op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()

op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

op_copy *Returns a copy of* x.

Description

Returns a copy of x.

Usage

```
op_copy(x)
```

Arguments

x	Input tensor.

Value

A copy of x.

See Also

- https://keras.io/api/ops/numpy#copy-function

Other numpy ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_bincount()
op_broadcast_to()
op_ceil()
op_clip()
op_concatenate()
op_conj()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_cumprod()
op_cumsum()
op_diag()
op_diagonal()

op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_einsum()
op_empty()
op_equal()
op_exp()
op_expand_dims()
op_expm1()
op_eye()
op_flip()
op_floor()
op_floor_divide()
op_full()
op_full_like()
op_get_item()
op_greater()
op_greater_equal()
op_hstack()
op_identity()
op_imag()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_matmul()
op_max()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()

op_mod()
op_moveaxis()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_not_equal()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_repeat()
op_reshape()
op_roll()
op_round()
op_sign()
op_sin()
op_sinh()
op_size()
op_sort()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_subtract()
op_sum()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_var()

op_vdot()
op_vstack()
op_where()
op_zeros()
op_zeros_like()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()

op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()

op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()

op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()

op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

---

op_correlate — *Compute the cross-correlation of two 1-dimensional tensors.*

Description

Compute the cross-correlation of two 1-dimensional tensors.

Usage

```
op_correlate(x1, x2, mode = "valid")
```

Arguments

x1	First 1-dimensional input tensor of length M.
x2	Second 1-dimensional input tensor of length N.
mode	Either "valid", "same" or "full". By default the mode is set to "valid", which returns an output of length max(M, N) - min(M, N) + 1. "same" returns an output of length max(M, N). "full" mode returns the convolution at each point of overlap, with an output length of N+M-1.

Value

Output tensor, cross-correlation of `x1` and `x2`.

See Also

Other numpy ops:
`op_abs()`
`op_add()`
`op_all()`
`op_any()`
`op_append()`
`op_arange()`
`op_arccos()`
`op_arccosh()`
`op_arcsin()`
`op_arcsinh()`
`op_arctan()`
`op_arctan2()`
`op_arctanh()`
`op_argmax()`
`op_argmin()`
`op_argsort()`
`op_array()`
`op_average()`
`op_bincount()`
`op_broadcast_to()`
`op_ceil()`
`op_clip()`
`op_concatenate()`
`op_conj()`
`op_copy()`
`op_cos()`
`op_cosh()`
`op_count_nonzero()`
`op_cross()`
`op_cumprod()`
`op_cumsum()`
`op_diag()`
`op_diagonal()`
`op_diff()`
`op_digitize()`
`op_divide()`
`op_divide_no_nan()`
`op_dot()`
`op_einsum()`
`op_empty()`
`op_equal()`
`op_exp()`

op_expand_dims()
op_expm1()
op_eye()
op_flip()
op_floor()
op_floor_divide()
op_full()
op_full_like()
op_get_item()
op_greater()
op_greater_equal()
op_hstack()
op_identity()
op_imag()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_matmul()
op_max()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moveaxis()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_not_equal()
op_ones()

op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_repeat()
op_reshape()
op_roll()
op_round()
op_sign()
op_sin()
op_sinh()
op_size()
op_sort()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_subtract()
op_sum()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_var()
op_vdot()
op_vstack()
op_where()
op_zeros()
op_zeros_like()

Other ops:
op_abs()
op_add()

op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()

op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()

op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()

op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()

op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

op_cos *Cosine, element-wise.*

Description

Cosine, element-wise.

Usage

```
op_cos(x)
```

Arguments

x	Input tensor.

Value

The corresponding cosine values.

See Also

- https://keras.io/api/ops/numpy#cos-function

Other numpy ops:
op_abs()
op_add()
op_all()
op_any()
op_append()

op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_bincount()
op_broadcast_to()
op_ceil()
op_clip()
op_concatenate()
op_conj()
op_copy()
op_correlate()
op_cosh()
op_count_nonzero()
op_cross()
op_cumprod()
op_cumsum()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_einsum()
op_empty()
op_equal()
op_exp()
op_expand_dims()
op_expm1()
op_eye()
op_flip()
op_floor()
op_floor_divide()
op_full()
op_full_like()
op_get_item()
op_greater()
op_greater_equal()

op_hstack()
op_identity()
op_imag()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_matmul()
op_max()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moveaxis()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_not_equal()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_repeat()
op_reshape()

op_roll()
op_round()
op_sign()
op_sin()
op_sinh()
op_size()
op_sort()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_subtract()
op_sum()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_var()
op_vdot()
op_vstack()
op_where()
op_zeros()
op_zeros_like()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()

op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()

op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()

op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()

op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()

op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

---

op_cosh *Hyperbolic cosine, element-wise.*

Description

Hyperbolic cosine, element-wise.

Usage

```
op_cosh(x)
```

Arguments

x	Input tensor.

Value

Output tensor of same shape as x.

See Also

- https://keras.io/api/ops/numpy#cosh-function

Other numpy ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()

op_array()
op_average()
op_bincount()
op_broadcast_to()
op_ceil()
op_clip()
op_concatenate()
op_conj()
op_copy()
op_correlate()
op_cos()
op_count_nonzero()
op_cross()
op_cumprod()
op_cumsum()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_einsum()
op_empty()
op_equal()
op_exp()
op_expand_dims()
op_expm1()
op_eye()
op_flip()
op_floor()
op_floor_divide()
op_full()
op_full_like()
op_get_item()
op_greater()
op_greater_equal()
op_hstack()
op_identity()
op_imag()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_less()
op_less_equal()
op_linspace()
op_log()

op_log10()
op_log1p()
op_log2()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_matmul()
op_max()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moveaxis()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_not_equal()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_repeat()
op_reshape()
op_roll()
op_round()
op_sign()
op_sin()
op_sinh()
op_size()
op_sort()
op_split()
op_sqrt()
op_square()
op_squeeze()

op_stack()
op_std()
op_subtract()
op_sum()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_var()
op_vdot()
op_vstack()
op_where()
op_zeros()
op_zeros_like()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()

op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()

op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()

op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()

op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

---

op_count_nonzero *Counts the number of non-zero values in* `x` *along the given* `axis`*.*

Description

If no axis is specified then all non-zeros in the tensor are counted.

Usage

```
op_count_nonzero(x, axis = NULL)
```

Arguments

`x`	Input tensor.
`axis`	Axis or a tuple of axes along which to count the number of non-zeros. Defaults to `NULL`.

Value

An integer or a tensor of integers.

Examples

```
x <- op_array(rbind(c(0, 1, 7, 0),
                    c(3, 0, 2, 19)))
op_count_nonzero(x)

## tf.Tensor(5, shape=(), dtype=int32)


op_count_nonzero(x, axis = 1)

## tf.Tensor([1 1 2 1], shape=(4), dtype=int32)


op_count_nonzero(x, axis = 2)

## tf.Tensor([2 3], shape=(2), dtype=int32)
```

See Also

- https://keras.io/api/ops/numpy#countnonzero-function

Other numpy ops:
`op_abs()`
`op_add()`
`op_all()`
`op_any()`
`op_append()`
`op_arange()`
`op_arccos()`
`op_arccosh()`

op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_bincount()
op_broadcast_to()
op_ceil()
op_clip()
op_concatenate()
op_conj()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_cross()
op_cumprod()
op_cumsum()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_einsum()
op_empty()
op_equal()
op_exp()
op_expand_dims()
op_expm1()
op_eye()
op_flip()
op_floor()
op_floor_divide()
op_full()
op_full_like()
op_get_item()
op_greater()
op_greater_equal()
op_hstack()
op_identity()
op_imag()

op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_matmul()
op_max()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moveaxis()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_not_equal()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_repeat()
op_reshape()
op_roll()
op_round()
op_sign()

op_sin()
op_sinh()
op_size()
op_sort()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_subtract()
op_sum()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_var()
op_vdot()
op_vstack()
op_where()
op_zeros()
op_zeros_like()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()

op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()

op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()

op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()

op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()

op_while_loop()
op_zeros()
op_zeros_like()

op_cross *Returns the cross product of two (arrays of) vectors.*

Description

The cross product of x1 and x2 in R^3 is a vector perpendicular to both x1 and x2. If x1 and x2 are arrays of vectors, the vectors are defined by the last axis of x1 and x2 by default, and these axes can have dimensions 2 or 3.

Where the dimension of either x1 or x2 is 2, the third component of the input vector is assumed to be zero and the cross product calculated accordingly.

In cases where both input vectors have dimension 2, the z-component of the cross product is returned.

Usage

```
op_cross(x1, x2, axisa = -1L, axisb = -1L, axisc = -1L, axis = NULL)
```

Arguments

x1	Components of the first vector(s).
x2	Components of the second vector(s).
axisa	Axis of x1 that defines the vector(s). Defaults to -1.
axisb	Axis of x2 that defines the vector(s). Defaults to -1.
axisc	Axis of the result containing the cross product vector(s). Ignored if both input vectors have dimension 2, as the return is scalar. By default, the last axis.
axis	If defined, the axis of x1, x2 and the result that defines the vector(s) and cross product(s). Overrides axisa, axisb and axisc.

Value

Vector cross product(s).

Note

Torch backend does not support two dimensional vectors, or the arguments axisa, axisb and axisc. Use axis instead.

See Also

- https://keras.io/api/ops/numpy#cross-function

Other numpy ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_bincount()
op_broadcast_to()
op_ceil()
op_clip()
op_concatenate()
op_conj()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cumprod()
op_cumsum()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_einsum()
op_empty()
op_equal()
op_exp()
op_expand_dims()
op_expm1()
op_eye()

op_flip()
op_floor()
op_floor_divide()
op_full()
op_full_like()
op_get_item()
op_greater()
op_greater_equal()
op_hstack()
op_identity()
op_imag()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_matmul()
op_max()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moveaxis()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_not_equal()
op_ones()
op_ones_like()
op_outer()
op_pad()

op_power()
op_prod()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_repeat()
op_reshape()
op_roll()
op_round()
op_sign()
op_sin()
op_sinh()
op_size()
op_sort()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_subtract()
op_sum()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_var()
op_vdot()
op_vstack()
op_where()
op_zeros()
op_zeros_like()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()

op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()

op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()

op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()

op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()

op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

---

op_ctc_loss *CTC (Connectionist Temporal Classification) loss.*

Description

CTC (Connectionist Temporal Classification) loss.

Usage

```
op_ctc_loss(target, output, target_length, output_length, mask_index = 0L)
```

Arguments

target	A tensor of shape (batch_size, max_length) containing the true labels in integer format.
output	A tensor of shape (batch_size, max_length, num_classes) containing logits (the output of your model).
target_length	A tensor of shape (batch_size) containing the true label lengths.
output_length	A tensor of shape (batch_size) containing the output lengths.
mask_index	The index of the mask character in the vocabulary. Defaults to 0.

Value

A tensor, shape (batch_size), of loss values.

See Also

Other nn ops:
`op_average_pool()`
`op_batch_normalization()`
`op_binary_crossentropy()`
`op_categorical_crossentropy()`
`op_conv()`
`op_conv_transpose()`
`op_depthwise_conv()`
`op_elu()`
`op_gelu()`
`op_hard_sigmoid()`
`op_hard_silu()`
`op_leaky_relu()`
`op_log_sigmoid()`
`op_log_softmax()`
`op_max_pool()`
`op_moments()`
`op_multi_hot()`
`op_normalize()`
`op_one_hot()`
`op_relu()`
`op_relu6()`
`op_selu()`
`op_separable_conv()`
`op_sigmoid()`
`op_silu()`
`op_softmax()`
`op_softplus()`
`op_softsign()`
`op_sparse_categorical_crossentropy()`

Other ops:
`op_abs()`
`op_add()`
`op_all()`
`op_any()`
`op_append()`
`op_arange()`
`op_arccos()`
`op_arccosh()`
`op_arcsin()`
`op_arcsinh()`
`op_arctan()`
`op_arctan2()`
`op_arctanh()`
`op_argmax()`
`op_argmin()`

op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()

op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()

op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()

op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()

op_where()
op_while_loop()
op_zeros()
op_zeros_like()

op_cumprod *Return the cumulative product of elements along a given axis.*

Description

Return the cumulative product of elements along a given axis.

Usage

```
op_cumprod(x, axis = NULL, dtype = NULL)
```

Arguments

x	Input tensor.
axis	Axis along which the cumulative product is computed. By default the input is flattened.
dtype	dtype of returned tensor. Defaults to x$dtype.

Value

Output tensor.

See Also

- https://keras.io/api/ops/numpy#cumprod-function

Other numpy ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()

op_argsort()
op_array()
op_average()
op_bincount()
op_broadcast_to()
op_ceil()
op_clip()
op_concatenate()
op_conj()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_cumsum()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_einsum()
op_empty()
op_equal()
op_exp()
op_expand_dims()
op_expm1()
op_eye()
op_flip()
op_floor()
op_floor_divide()
op_full()
op_full_like()
op_get_item()
op_greater()
op_greater_equal()
op_hstack()
op_identity()
op_imag()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_less()
op_less_equal()
op_linspace()

op_log()
op_log10()
op_log1p()
op_log2()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_matmul()
op_max()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moveaxis()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_not_equal()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_repeat()
op_reshape()
op_roll()
op_round()
op_sign()
op_sin()
op_sinh()
op_size()
op_sort()
op_split()
op_sqrt()
op_square()

op_squeeze()
op_stack()
op_std()
op_subtract()
op_sum()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_var()
op_vdot()
op_vstack()
op_where()
op_zeros()
op_zeros_like()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()

op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()

op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()

op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()

op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

---

op_cumsum *Returns the cumulative sum of elements along a given axis.*

Description

Returns the cumulative sum of elements along a given axis.

Usage

```
op_cumsum(x, axis = NULL, dtype = NULL)
```

Arguments

`x`	Input tensor.
`axis`	Axis along which the cumulative sum is computed. By default the input is flattened.
`dtype`	dtype of returned tensor. Defaults to `x$dtype`.

Value

Output tensor.

See Also

- https://keras.io/api/ops/numpy#cumsum-function

Other numpy ops:
`op_abs()`
`op_add()`
`op_all()`
`op_any()`
`op_append()`
`op_arange()`
`op_arccos()`
`op_arccosh()`
`op_arcsin()`
`op_arcsinh()`
`op_arctan()`
`op_arctan2()`
`op_arctanh()`
`op_argmax()`
`op_argmin()`
`op_argsort()`
`op_array()`
`op_average()`
`op_bincount()`
`op_broadcast_to()`
`op_ceil()`
`op_clip()`
`op_concatenate()`
`op_conj()`
`op_copy()`
`op_correlate()`

op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_cumprod()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_einsum()
op_empty()
op_equal()
op_exp()
op_expand_dims()
op_expm1()
op_eye()
op_flip()
op_floor()
op_floor_divide()
op_full()
op_full_like()
op_get_item()
op_greater()
op_greater_equal()
op_hstack()
op_identity()
op_imag()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_matmul()

op_max()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moveaxis()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_not_equal()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_repeat()
op_reshape()
op_roll()
op_round()
op_sign()
op_sin()
op_sinh()
op_size()
op_sort()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_subtract()
op_sum()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()

op_tile()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_var()
op_vdot()
op_vstack()
op_where()
op_zeros()
op_zeros_like()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()

op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()

op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()

op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()

op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

op_custom_gradient *Decorator to define a function with a custom gradient.*

Description

This decorator allows fine grained control over the gradients of a sequence for operations. This may be useful for multiple reasons, including providing a more efficient or numerically stable gradient for a sequence of operations.

Usage

```
op_custom_gradient(f)
```

Arguments

f	Function `f(...)` that returns a tuple `(output, grad_fn)` where: • `...` is a sequence of unnamed arguments, each a tensor input or nested structure of tensor inputs to the function. • `output` is a (potentially nested structure of) tensor outputs of applying operations in forward_fn `f()` to `...`. • `grad_fn` is a function with the signature `grad_fn(..., upstream)` which returns a list of tensors the same size as (flattened) `...`: the derivatives of tensors in `output` with respect to the tensors in `...`. `upstream` is a tensor or sequence of tensors holding the initial value gradients for each tensor in `output`.

Value

A function `h(...)` which returns the same value as `f(...)[[1]]` and whose gradient is determined by `f(...)[[2]]`.

Example

Backend-agnostic example.

```
log1pexp <- op_custom_gradient(\(x) {

    e <- op_exp(x)

    grad <- function(..., upstream = NULL) {
      upstream <- upstream %||% ..1
      op_multiply(upstream, 1.0 - 1.0 / op_add(1, e))
    }

    tuple(op_log(1 + e), grad)

})

if(config_backend() == "tensorflow") {
  tf <- tensorflow::tf
  x <- op_convert_to_tensor(100.0)
  with(tf$GradientTape() %as% tape, {
    tape$watch(x)
    y <- log1pexp(x)
  })
  dy_dx <- tape$gradient(y, x)
  stopifnot(as.numeric(dy_dx) == 1)
}
```

Note

Note that the `grad` function that returns gradient computation requires `...` as well as an `upstream` named argument, depending on the backend being set. With the JAX and TensorFlow backends, it requires only one argument, whereas it might use the `upstream` argument in the case of the PyTorch backend.

When working with TensorFlow/JAX backend, `grad(upstream)` is sufficient. With PyTorch, the `grad` function requires `...` as well as `upstream`, e.g. `grad <- \(..., upstream)`. Follow the example above to use `op_custom_gradient()` in a way that is compatible with all backends.

See Also

- `https://www.tensorflow.org/api_docs/python/tf/keras/ops/custom_gradient`

Other core ops:
`op_cast()`
`op_cond()`
`op_convert_to_numpy()`
`op_convert_to_tensor()`
`op_fori_loop()`
`op_is_tensor()`
`op_scatter()`
`op_scatter_update()`
`op_shape()`
`op_slice()`
`op_slice_update()`
`op_stop_gradient()`
`op_unstack()`
`op_vectorized_map()`
`op_while_loop()`

Other ops:
`op_abs()`
`op_add()`
`op_all()`
`op_any()`
`op_append()`
`op_arange()`
`op_arccos()`
`op_arccosh()`
`op_arcsin()`
`op_arcsinh()`
`op_arctan()`
`op_arctan2()`
`op_arctanh()`
`op_argmax()`
`op_argmin()`
`op_argsort()`
`op_array()`
`op_average()`
`op_average_pool()`
`op_batch_normalization()`
`op_binary_crossentropy()`

op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()

op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()

op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()

op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

`op_depthwise_conv` *General N-D depthwise convolution.*

Description

This ops supports 1D and 2D depthwise convolution.

Usage

```
op_depthwise_conv(
  inputs,
  kernel,
  strides = 1L,
  padding = "valid",
  data_format = NULL,
  dilation_rate = 1L
)
```

Arguments

`inputs`	Tensor of rank N+2. `inputs` has shape `(batch_size,) + inputs_spatial_shape + (num_channels,)` if `data_format = "channels_last"`, or `(batch_size, num_channels) + inputs_spatial_shape` if `data_format = "channels_first"`.
`kernel`	Tensor of rank N+2. `kernel` has shape `[kernel_spatial_shape, num_input_channels, num_channel` `num_input_channels` should match the number of channels in `inputs`.
`strides`	int or int tuple/list of `len(inputs_spatial_shape)`, specifying the strides of the convolution along each spatial dimension. If `strides` is int, then every spatial dimension shares the same `strides`.
`padding`	string, either `"valid"` or `"same"`. `"valid"` means no padding is applied, and `"same"` results in padding evenly to the left/right or up/down of the input such that output has the same height/width dimension as the input when `strides = 1`.
`data_format`	A string, either `"channels_last"` or `"channels_first"`. `data_format` determines the ordering of the dimensions in the inputs. If `data_format = "channels_last"`, `inputs` is of shape `(batch_size, ..., channels)` while if `data_format = "channels_first"`, `inputs` is of shape `(batch_size, channels, ...)`.
`dilation_rate`	int or int tuple/list of `len(inputs_spatial_shape)`, specifying the dilation rate to use for dilated convolution. If `dilation_rate` is int, then every spatial dimension shares the same `dilation_rate`.

Value

A tensor of rank N+2, the result of the depthwise conv operation.

See Also

- https://keras.io/api/ops/nn#depthwiseconv-function

Other nn ops:
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_categorical_crossentropy()
op_conv()
op_conv_transpose()
op_ctc_loss()
op_elu()
op_gelu()
op_hard_sigmoid()
op_hard_silu()
op_leaky_relu()
op_log_sigmoid()
op_log_softmax()
op_max_pool()
op_moments()
op_multi_hot()
op_normalize()
op_one_hot()
op_relu()
op_relu6()
op_selu()
op_separable_conv()
op_sigmoid()
op_silu()
op_softmax()
op_softplus()
op_softsign()
op_sparse_categorical_crossentropy()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()

op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()

op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()

op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()

op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()

op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

---

op_det *Computes the determinant of a square tensor.*

Description

Computes the determinant of a square tensor.

Usage

```
op_det(x)
```

Arguments

x	Input tensor of shape `(..., M, M)`.

Value

A tensor of shape `(...)` represeting the determinant of `x`.

See Also

Other linear algebra ops:
op_cholesky()
op_eig()
op_inv()
op_lu_factor()
op_norm()
op_solve_triangular()
op_svd()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()

op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()

op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()

op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()

op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()

op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

op_diag — *Extract a diagonal or construct a diagonal array.*

Description

Extract a diagonal or construct a diagonal array.

Usage

```
op_diag(x, k = 0L)
```

Arguments

x	Input tensor. If x is 2-D, returns the k-th diagonal of x. If x is 1-D, return a 2-D tensor with x on the k-th diagonal.
k	The diagonal to consider. Defaults to 0. Use k > 0 for diagonals above the main diagonal, and k < 0 for diagonals below the main diagonal.

Value

The extracted diagonal or constructed diagonal tensor.

Examples

```
x <- op_arange(9L) |> op_reshape(c(3, 3))
x

## tf.Tensor(
## [[0 1 2]
##  [3 4 5]
##  [6 7 8]], shape=(3, 3), dtype=int32)


op_diag(x)
```

```
## tf.Tensor([0 4 8], shape=(3), dtype=int32)
```

```
op_diag(x, k = 1)
```

```
## tf.Tensor([1 5], shape=(2), dtype=int32)
```

```
op_diag(x, k = -1)
```

```
## tf.Tensor([3 7], shape=(2), dtype=int32)
```

```
op_diag(op_diag(x))
```

```
## tf.Tensor(
## [[0 0 0]
##  [0 4 0]
##  [0 0 8]], shape=(3, 3), dtype=int32)
```

See Also

- https://keras.io/api/ops/numpy#diag-function

Other numpy ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_bincount()
op_broadcast_to()
op_ceil()
op_clip()
op_concatenate()

op_conj()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_cumprod()
op_cumsum()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_einsum()
op_empty()
op_equal()
op_exp()
op_expand_dims()
op_expm1()
op_eye()
op_flip()
op_floor()
op_floor_divide()
op_full()
op_full_like()
op_get_item()
op_greater()
op_greater_equal()
op_hstack()
op_identity()
op_imag()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()

op_logical_xor()
op_logspace()
op_matmul()
op_max()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moveaxis()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_not_equal()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_repeat()
op_reshape()
op_roll()
op_round()
op_sign()
op_sin()
op_sinh()
op_size()
op_sort()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_subtract()
op_sum()
op_swapaxes()
op_take()
op_take_along_axis()

op_tan()
op_tanh()
op_tensordot()
op_tile()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_var()
op_vdot()
op_vstack()
op_where()
op_zeros()
op_zeros_like()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()

op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()

op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()

op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()

op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

op_diagonal *Return specified diagonals.*

Description

If x is 2-D, returns the diagonal of x with the given offset, i.e., the collection of elements of the form x[i, i+offset].

If x has more than two dimensions, the axes specified by axis1 and axis2 are used to determine the 2-D sub-array whose diagonal is returned.

The shape of the resulting array can be determined by removing `axis1` and `axis2` and appending an index to the right equal to the size of the resulting diagonals.

Usage

```
op_diagonal(x, offset = 0L, axis1 = 1L, axis2 = 2L)
```

Arguments

`x`	Input tensor.
`offset`	Offset of the diagonal from the main diagonal. Can be positive or negative. Defaults to `0` (main diagonal).
`axis1`	Axis to be used as the first axis of the 2-D sub-arrays. Defaults to `1` (first axis).
`axis2`	Axis to be used as the second axis of the 2-D sub-arrays. Defaults to `2` (second axis).

Value

Tensor of diagonals.

Examples

```
x <- op_arange(4L) |> op_reshape(c(2, 2))
x

## tf.Tensor(
## [[0 1]
##  [2 3]], shape=(2, 2), dtype=int32)


op_diagonal(x)

## tf.Tensor([0 3], shape=(2), dtype=int32)


op_diagonal(x, offset = 1)

## tf.Tensor([1], shape=(1), dtype=int32)


x <- op_array(1:8) |> op_reshape(c(2, 2, 2))
x

## tf.Tensor(
## [[[1 2]
##   [3 4]]
##
##  [[5 6]
##   [7 8]]], shape=(2, 2, 2), dtype=int32)
```

```
x |> op_diagonal(0)

## tf.Tensor(
## [[1 7]
##  [2 8]], shape=(2, 2), dtype=int32)


x |> op_diagonal(0, 1, 2) # same as above, the default

## tf.Tensor(
## [[1 7]
##  [2 8]], shape=(2, 2), dtype=int32)


x |> op_diagonal(0, 2, 3)

## tf.Tensor(
## [[1 4]
##  [5 8]], shape=(2, 2), dtype=int32)
```

See Also

- https://keras.io/api/ops/numpy#diagonal-function

Other numpy ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_bincount()
op_broadcast_to()
op_ceil()
op_clip()
op_concatenate()

op_conj()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_cumprod()
op_cumsum()
op_diag()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_einsum()
op_empty()
op_equal()
op_exp()
op_expand_dims()
op_expm1()
op_eye()
op_flip()
op_floor()
op_floor_divide()
op_full()
op_full_like()
op_get_item()
op_greater()
op_greater_equal()
op_hstack()
op_identity()
op_imag()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()

op_logical_xor()
op_logspace()
op_matmul()
op_max()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moveaxis()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_not_equal()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_repeat()
op_reshape()
op_roll()
op_round()
op_sign()
op_sin()
op_sinh()
op_size()
op_sort()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_subtract()
op_sum()
op_swapaxes()
op_take()
op_take_along_axis()

op_tan()
op_tanh()
op_tensordot()
op_tile()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_var()
op_vdot()
op_vstack()
op_where()
op_zeros()
op_zeros_like()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()

op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()

op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()

op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()

op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

---

op_diff — *Calculate the n-th discrete difference along the given axis.*

Description

The first difference is given by `out[i] = a[i+1] - a[i]` along the given axis, higher differences are calculated by using `diff` recursively.

Usage

```
op_diff(a, n = 1L, axis = -1L)
```

Arguments

`a`	Input tensor.
`n`	The number of times values are differenced. Defaults to `1`.
`axis`	Axis to compute discrete difference(s) along. Defaults to `-1` (last axis).

Value

Tensor of diagonals.

Examples

```
x <- op_array(c(1, 2, 4, 7, 0))
op_diff(x)

## tf.Tensor([ 1.  2.  3. -7.], shape=(4), dtype=float32)


op_diff(x, n = 2)

## tf.Tensor([  1.   1. -10.], shape=(3), dtype=float32)


x <- op_array(rbind(c(1, 3, 6, 10),
                    c(0, 5, 6, 8)))
op_diff(x)

## tf.Tensor(
## [[2. 3. 4.]
##  [5. 1. 2.]], shape=(2, 3), dtype=float64)


op_diff(x, axis = 1)

## tf.Tensor([[-1.  2.  0. -2.]], shape=(1, 4), dtype=float64)
```

See Also

Other numpy ops:
`op_abs()`
`op_add()`
`op_all()`
`op_any()`
`op_append()`

op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_bincount()
op_broadcast_to()
op_ceil()
op_clip()
op_concatenate()
op_conj()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_cumprod()
op_cumsum()
op_diag()
op_diagonal()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_einsum()
op_empty()
op_equal()
op_exp()
op_expand_dims()
op_expm1()
op_eye()
op_flip()
op_floor()
op_floor_divide()
op_full()
op_full_like()
op_get_item()
op_greater()
op_greater_equal()

op_hstack()
op_identity()
op_imag()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_matmul()
op_max()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moveaxis()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_not_equal()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_repeat()
op_reshape()

op_roll()
op_round()
op_sign()
op_sin()
op_sinh()
op_size()
op_sort()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_subtract()
op_sum()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_var()
op_vdot()
op_vstack()
op_where()
op_zeros()
op_zeros_like()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()

op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()

op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()

op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()

op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()

op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

---

op_digitize *Returns the indices of the bins to which each value in* x *belongs.*

Description

Returns the indices of the bins to which each value in x belongs.

Usage

```
op_digitize(x, bins)
```

Arguments

x	Input array to be binned.
bins	Array of bins. It has to be one-dimensional and monotonically increasing.

Value

Output array of indices, of same shape as x.

Examples

```
x <- op_array(c(0.0, 1.0, 3.0, 1.6))
bins <- array(c(0.0, 3.0, 4.5, 7.0))
op_digitize(x, bins)

## tf.Tensor([1 1 2 1], shape=(4), dtype=int32)


# array([1, 1, 2, 1])
```

See Also

- https://keras.io/api/ops/numpy#digitize-function

Other numpy ops:
op_abs()
op_add()
op_all()
op_any()

op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_bincount()
op_broadcast_to()
op_ceil()
op_clip()
op_concatenate()
op_conj()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_cumprod()
op_cumsum()
op_diag()
op_diagonal()
op_diff()
op_divide()
op_divide_no_nan()
op_dot()
op_einsum()
op_empty()
op_equal()
op_exp()
op_expand_dims()
op_expm1()
op_eye()
op_flip()
op_floor()
op_floor_divide()
op_full()
op_full_like()
op_get_item()
op_greater()

op_greater_equal()
op_hstack()
op_identity()
op_imag()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_matmul()
op_max()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moveaxis()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_not_equal()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_repeat()

op_reshape()
op_roll()
op_round()
op_sign()
op_sin()
op_sinh()
op_size()
op_sort()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_subtract()
op_sum()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_var()
op_vdot()
op_vstack()
op_where()
op_zeros()
op_zeros_like()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()

op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()

op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()

op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()

op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()

op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

---

op_divide *Divide arguments element-wise.*

Description

Note that this function is automatically called when using the R operator * with a tensor.

```
(x <- op_arange(4))

## tf.Tensor([0. 1. 2. 3.], shape=(4), dtype=float64)


op_divide(x, 2)

## tf.Tensor([0.  0.5 1.  1.5], shape=(4), dtype=float64)


x / 2

## tf.Tensor([0.  0.5 1.  1.5], shape=(4), dtype=float64)
```

Usage

```
op_divide(x1, x2)
```

Arguments

x1	First input tensor.
x2	Second input tensor.

Value

Output tensor, the quotient x1/x2, element-wise.

Example

```
op_divide(3, 2)

## tf.Tensor(1.5, shape=(), dtype=float32)
```

See Also

- https://keras.io/api/ops/numpy#divide-function

Other numpy ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_bincount()
op_broadcast_to()
op_ceil()
op_clip()
op_concatenate()
op_conj()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_cumprod()
op_cumsum()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide_no_nan()
op_dot()

op_einsum()
op_empty()
op_equal()
op_exp()
op_expand_dims()
op_expm1()
op_eye()
op_flip()
op_floor()
op_floor_divide()
op_full()
op_full_like()
op_get_item()
op_greater()
op_greater_equal()
op_hstack()
op_identity()
op_imag()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_matmul()
op_max()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moveaxis()
op_multiply()
op_nan_to_num()
op_ndim()

op_negative()
op_nonzero()
op_not_equal()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_repeat()
op_reshape()
op_roll()
op_round()
op_sign()
op_sin()
op_sinh()
op_size()
op_sort()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_subtract()
op_sum()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_var()
op_vdot()
op_vstack()
op_where()
op_zeros()
op_zeros_like()

Other ops:
`op_abs()`
`op_add()`
`op_all()`
`op_any()`
`op_append()`
`op_arange()`
`op_arccos()`
`op_arccosh()`
`op_arcsin()`
`op_arcsinh()`
`op_arctan()`
`op_arctan2()`
`op_arctanh()`
`op_argmax()`
`op_argmin()`
`op_argsort()`
`op_array()`
`op_average()`
`op_average_pool()`
`op_batch_normalization()`
`op_binary_crossentropy()`
`op_bincount()`
`op_broadcast_to()`
`op_cast()`
`op_categorical_crossentropy()`
`op_ceil()`
`op_cholesky()`
`op_clip()`
`op_concatenate()`
`op_cond()`
`op_conj()`
`op_conv()`
`op_conv_transpose()`
`op_convert_to_numpy()`
`op_convert_to_tensor()`
`op_copy()`
`op_correlate()`
`op_cos()`
`op_cosh()`
`op_count_nonzero()`
`op_cross()`
`op_ctc_loss()`
`op_cumprod()`
`op_cumsum()`
`op_custom_gradient()`
`op_depthwise_conv()`

op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()

op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()

op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()

op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

`op_divide_no_nan` *Safe element-wise division which returns 0 where the denominator is 0.*

Description

Safe element-wise division which returns 0 where the denominator is 0.

Usage

```
op_divide_no_nan(x1, x2)
```

Arguments

x1	First input tensor.
x2	Second input tensor.

Value

The quotient `x1/x2`, element-wise, with zero where x2 is zero.

See Also

Other numpy ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_bincount()
op_broadcast_to()
op_ceil()
op_clip()
op_concatenate()
op_conj()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_cumprod()
op_cumsum()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_dot()
op_einsum()
op_empty()
op_equal()
op_exp()
op_expand_dims()
op_expm1()
op_eye()
op_flip()

op_floor()
op_floor_divide()
op_full()
op_full_like()
op_get_item()
op_greater()
op_greater_equal()
op_hstack()
op_identity()
op_imag()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_matmul()
op_max()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moveaxis()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_not_equal()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()

op_prod()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_repeat()
op_reshape()
op_roll()
op_round()
op_sign()
op_sin()
op_sinh()
op_size()
op_sort()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_subtract()
op_sum()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_var()
op_vdot()
op_vstack()
op_where()
op_zeros()
op_zeros_like()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()

op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_dot()
op_eig()

op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()

op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()

op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()

op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

---

op_dot *Dot product of two tensors.*

Description

- If both x1 and x2 are 1-D tensors, it is inner product of vectors (without complex conjugation).
- If both x1 and x2 are 2-D tensors, it is matrix multiplication.
- If either x1 or x2 is 0-D (scalar), it is equivalent to x1 * x2.
- If x1 is an N-D tensor and x2 is a 1-D tensor, it is a sum product over the last axis of x1 and x2.
- If x1 is an N-D tensor and x2 is an M-D tensor (where M >= 2), it is a sum product over the last axis of x1 and the second-to-last axis of x2: `dot(x1, x2)[i,j,k,m] = sum(a[i,j,:] * b[k,:,m])`.

Usage

```
op_dot(x1, x2)
```

Arguments

x1	First argument.
x2	Second argument.

Value

Dot product of x1 and x2.

Note

Torch backend does not accept 0-D tensors as arguments.

See Also

- https://keras.io/api/ops/numpy#dot-function

Other numpy ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_bincount()
op_broadcast_to()
op_ceil()
op_clip()
op_concatenate()
op_conj()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_cumprod()
op_cumsum()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_einsum()
op_empty()
op_equal()
op_exp()
op_expand_dims()
op_expm1()
op_eye()

op_flip()
op_floor()
op_floor_divide()
op_full()
op_full_like()
op_get_item()
op_greater()
op_greater_equal()
op_hstack()
op_identity()
op_imag()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_matmul()
op_max()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moveaxis()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_not_equal()
op_ones()
op_ones_like()
op_outer()
op_pad()

op_power()
op_prod()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_repeat()
op_reshape()
op_roll()
op_round()
op_sign()
op_sin()
op_sinh()
op_size()
op_sort()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_subtract()
op_sum()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_var()
op_vdot()
op_vstack()
op_where()
op_zeros()
op_zeros_like()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()

op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()

op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()

op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()

op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()

op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

---

op_eig *Computes the eigenvalues and eigenvectors of a square matrix.*

Description

Computes the eigenvalues and eigenvectors of a square matrix.

Usage

```
op_eig(x)
```

Arguments

x	Input tensor of shape `(..., M, M)`.

Value

A list of two tensors: a tensor of shape `(..., M)` containing eigenvalues and a tensor of shape `(..., M, M)` containing eigenvectors.

See Also

Other linear algebra ops:
op_cholesky()
op_det()
op_inv()
op_lu_factor()
op_norm()
op_solve_triangular()
op_svd()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()

op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()

op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()

op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()

op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

---

op_einsum *Evaluates the Einstein summation convention on the operands.*

Description

Evaluates the Einstein summation convention on the operands.

Usage

```
op_einsum(subscripts, ...)
```

Arguments

subscripts	Specifies the subscripts for summation as comma separated list of subscript labels. An implicit (classical Einstein summation) calculation is performed unless the explicit indicator -> is included as well as subscript labels of the precise output form.
...	The operands to compute the Einstein sum of.

Value

The calculation based on the Einstein summation convention.

Examples

```
a <- op_arange(25) |> op_reshape(c(5, 5))
b <- op_arange(5)
c <- op_arange(6) |> op_reshape(c(2, 3))
```

Trace of a matrix:

```
op_einsum("ii", a)
op_trace(a)

## tf.Tensor(60.0, shape=(), dtype=float64)
## tf.Tensor(60.0, shape=(), dtype=float64)
```

Extract the diagonal:

```
op_einsum("ii -> i", a)
op_diag(a)

## tf.Tensor([ 0.  6. 12. 18. 24.], shape=(5), dtype=float64)
## tf.Tensor([ 0.  6. 12. 18. 24.], shape=(5), dtype=float64)
```

Sum over an axis:

```
op_einsum("ij -> i", a)
op_sum(a, axis = 2)

## tf.Tensor([ 10.  35.  60.  85. 110.], shape=(5), dtype=float64)
## tf.Tensor([ 10.  35.  60.  85. 110.], shape=(5), dtype=float64)
```

For higher dimensional tensors summing a single axis can be done with ellipsis:

```
op_einsum("...j -> ...", a)
op_sum(a, axis = -1)

## tf.Tensor([ 10.  35.  60.  85. 110.], shape=(5), dtype=float64)
## tf.Tensor([ 10.  35.  60.  85. 110.], shape=(5), dtype=float64)
```

Compute a matrix transpose or reorder any number of axes:

```
op_einsum("ji", c) # return c unchanged

## tf.Tensor(
## [[0. 1. 2.]
##  [3. 4. 5.]], shape=(2, 3), dtype=float64)
```

```
op_einsum("ij -> ji", c) # transpose
op_transpose(c)          # same as above

## tf.Tensor(
## [[0. 3.]
##  [1. 4.]
##  [2. 5.]], shape=(3, 2), dtype=float64)
## tf.Tensor(
## [[0. 3.]
##  [1. 4.]
##  [2. 5.]], shape=(3, 2), dtype=float64)
```

Matrix vector multiplication:

```
op_einsum("ij, j", a, b)
op_einsum("...j, j", a, b)
a %*% b
op_matmul(a, b)

## tf.Tensor([ 30.  80. 130. 180. 230.], shape=(5), dtype=float64)
## tf.Tensor([ 30.  80. 130. 180. 230.], shape=(5), dtype=float64)
## tf.Tensor([ 30.  80. 130. 180. 230.], shape=(5), dtype=float64)
## tf.Tensor([ 30.  80. 130. 180. 230.], shape=(5), dtype=float64)
```

See Also

- https://keras.io/api/ops/numpy#einsum-function

Other numpy ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()

op_bincount()
op_broadcast_to()
op_ceil()
op_clip()
op_concatenate()
op_conj()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_cumprod()
op_cumsum()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_empty()
op_equal()
op_exp()
op_expand_dims()
op_expm1()
op_eye()
op_flip()
op_floor()
op_floor_divide()
op_full()
op_full_like()
op_get_item()
op_greater()
op_greater_equal()
op_hstack()
op_identity()
op_imag()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()

op_log2()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_matmul()
op_max()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moveaxis()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_not_equal()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_repeat()
op_reshape()
op_roll()
op_round()
op_sign()
op_sin()
op_sinh()
op_size()
op_sort()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()

op_subtract()
op_sum()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_var()
op_vdot()
op_vstack()
op_where()
op_zeros()
op_zeros_like()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()

op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()

op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()

op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()

op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

`op_elu` *Exponential Linear Unit activation function.*

Description

It is defined as:
`f(x) =  alpha * (exp(x) - 1.) for x < 0`, `f(x) = x for x >= 0`.

Usage

```
op_elu(x, alpha = 1)
```

Arguments

`x`	Input tensor.
`alpha`	A scalar, slope of positive section. Defaults to `1.0`.

Value

A tensor with the same shape as `x`.

Examples

```
x <- op_array(c(-1., 0., 1.))
op_elu(x)

## tf.Tensor([-0.63212055  0.          1.        ], shape=(3), dtype=float32)
```

See Also

- `https://keras.io/api/ops/nn#elu-function`

Other nn ops:
`op_average_pool()`
`op_batch_normalization()`
`op_binary_crossentropy()`
`op_categorical_crossentropy()`
`op_conv()`
`op_conv_transpose()`
`op_ctc_loss()`
`op_depthwise_conv()`
`op_gelu()`
`op_hard_sigmoid()`
`op_hard_silu()`
`op_leaky_relu()`
`op_log_sigmoid()`
`op_log_softmax()`
`op_max_pool()`
`op_moments()`
`op_multi_hot()`
`op_normalize()`
`op_one_hot()`

op_relu()
op_relu6()
op_selu()
op_separable_conv()
op_sigmoid()
op_silu()
op_softmax()
op_softplus()
op_softsign()
op_sparse_categorical_crossentropy()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()

op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()

op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()

op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()

op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

---

op_empty *Return a tensor of given shape and type filled with uninitialized data.*

Description

Return a tensor of given shape and type filled with uninitialized data.

Usage

```
op_empty(shape, dtype = NULL)
```

Arguments

shape	Shape of the empty tensor.
dtype	Desired data type of the empty tensor.

Value

The empty tensor.

See Also

- https://keras.io/api/ops/numpy#empty-function

Other numpy ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_bincount()
op_broadcast_to()
op_ceil()
op_clip()
op_concatenate()
op_conj()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_cumprod()
op_cumsum()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_einsum()
op_equal()

op_exp()
op_expand_dims()
op_expm1()
op_eye()
op_flip()
op_floor()
op_floor_divide()
op_full()
op_full_like()
op_get_item()
op_greater()
op_greater_equal()
op_hstack()
op_identity()
op_imag()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_matmul()
op_max()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moveaxis()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_not_equal()

op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_repeat()
op_reshape()
op_roll()
op_round()
op_sign()
op_sin()
op_sinh()
op_size()
op_sort()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_subtract()
op_sum()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_var()
op_vdot()
op_vstack()
op_where()
op_zeros()
op_zeros_like()

Other ops:
op_abs()

op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()

op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()

op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()

op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()

op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

---

op_equal *Returns* (x1 == x2) *element-wise.*

Description

Note that this function is automatically called when using the R operator == with a tensor.

```
(x <- op_arange(4))
```

```
## tf.Tensor([0. 1. 2. 3.], shape=(4), dtype=float64)
```

```
op_equal(x, 2)
```

```
## tf.Tensor([False False  True False], shape=(4), dtype=bool)
```

```
x == 2
```

```
## tf.Tensor([False False  True False], shape=(4), dtype=bool)
```

Usage

```
op_equal(x1, x2)
```

Arguments

x1	Tensor to compare.
x2	Tensor to compare.

Value

Output tensor, element-wise comparison of x1 and x2.

See Also

- https://keras.io/api/ops/numpy#equal-function

Other numpy ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_bincount()
op_broadcast_to()
op_ceil()
op_clip()
op_concatenate()
op_conj()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_cumprod()
op_cumsum()
op_diag()
op_diagonal()
op_diff()
op_digitize()

op_divide()
op_divide_no_nan()
op_dot()
op_einsum()
op_empty()
op_exp()
op_expand_dims()
op_expm1()
op_eye()
op_flip()
op_floor()
op_floor_divide()
op_full()
op_full_like()
op_get_item()
op_greater()
op_greater_equal()
op_hstack()
op_identity()
op_imag()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_matmul()
op_max()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moveaxis()
op_multiply()

op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_not_equal()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_repeat()
op_reshape()
op_roll()
op_round()
op_sign()
op_sin()
op_sinh()
op_size()
op_sort()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_subtract()
op_sum()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_var()
op_vdot()
op_vstack()
op_where()

op_zeros()
op_zeros_like()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()

op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()

op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()

op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()

op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

---

op_erf *Computes the error function of* x*, element-wise.*

Description

Computes the error function of x, element-wise.

Usage

```
op_erf(x)
```

Arguments

x	Input tensor.

Value

A tensor with the same dtype as x.

Examples

```
x <- op_array(c(-3, -2, -1, 0, 1))
op_erf(x)

## tf.Tensor([-0.99997777 -0.9953222  -0.84270084  0.         0.84270084], shape=(5), dtype=float32)


# array([-0.99998 , -0.99532, -0.842701,  0.,  0.842701], dtype=float32)
```

See Also

Other math ops:
op_erfinv()
op_extract_sequences()
op_fft()
op_fft2()
op_in_top_k()
op_irfft()
op_istft()
op_logsumexp()
op_qr()
op_rfft()
op_rsqrt()
op_segment_max()
op_segment_sum()
op_solve()
op_stft()
op_top_k()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()

op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()

op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()

op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()

op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()

op_zeros_like()

---

op_erfinv *Computes the inverse error function of* x*, element-wise.*

Description

Computes the inverse error function of x, element-wise.

Usage

```
op_erfinv(x)
```

Arguments

x	Input tensor.

Value

A tensor with the same dtype as x.

Examples

```
x <- op_array(c(-0.5, -0.2, -0.1, 0.0, 0.3))
op_erfinv(x)

## tf.Tensor([-0.4769363  -0.17914344 -0.088856   0.         0.27246267], shape=(5), dtype=float32)
```

See Also

Other math ops:
op_erf()
op_extract_sequences()
op_fft()
op_fft2()
op_in_top_k()
op_irfft()
op_istft()
op_logsumexp()
op_qr()
op_rfft()
op_rsqrt()
op_segment_max()
op_segment_sum()
op_solve()
op_stft()

op_top_k()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()

op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()

op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()

op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()

op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

op_exp *Calculate the exponential of all elements in the input tensor.*

Description

Calculate the exponential of all elements in the input tensor.

Usage

```
op_exp(x)
```

Arguments

x	Input tensor.

Value

Output tensor, element-wise exponential of x.

See Also

- https://keras.io/api/ops/numpy#exp-function

Other numpy ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_bincount()
op_broadcast_to()
op_ceil()
op_clip()
op_concatenate()
op_conj()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_cumprod()
op_cumsum()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_einsum()
op_empty()
op_equal()
op_expand_dims()
op_expm1()
op_eye()

op_flip()
op_floor()
op_floor_divide()
op_full()
op_full_like()
op_get_item()
op_greater()
op_greater_equal()
op_hstack()
op_identity()
op_imag()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_matmul()
op_max()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moveaxis()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_not_equal()
op_ones()
op_ones_like()
op_outer()
op_pad()

op_power()
op_prod()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_repeat()
op_reshape()
op_roll()
op_round()
op_sign()
op_sin()
op_sinh()
op_size()
op_sort()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_subtract()
op_sum()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_var()
op_vdot()
op_vstack()
op_where()
op_zeros()
op_zeros_like()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()

op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()

op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()

op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()

op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()

op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

---

op_expand_dims *Expand the shape of a tensor.*

Description

Insert a new axis at the `axis` position in the expanded tensor shape.

Usage

```
op_expand_dims(x, axis)
```

Arguments

`x`	Input tensor.
`axis`	Position in the expanded axes where the new axis (or axes) is placed.

Value

Output tensor with the number of dimensions increased.

See Also

- https://keras.io/api/ops/numpy#expanddims-function

Other numpy ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()

op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_bincount()
op_broadcast_to()
op_ceil()
op_clip()
op_concatenate()
op_conj()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_cumprod()
op_cumsum()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_einsum()
op_empty()
op_equal()
op_exp()
op_expm1()
op_eye()
op_flip()
op_floor()
op_floor_divide()
op_full()
op_full_like()
op_get_item()
op_greater()
op_greater_equal()
op_hstack()
op_identity()

op_imag()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_matmul()
op_max()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moveaxis()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_not_equal()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_repeat()
op_reshape()
op_roll()
op_round()

op_sign()
op_sin()
op_sinh()
op_size()
op_sort()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_subtract()
op_sum()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_var()
op_vdot()
op_vstack()
op_where()
op_zeros()
op_zeros_like()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()

op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expm1()

op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()

op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()

op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()

op_where()
op_while_loop()
op_zeros()
op_zeros_like()

op_expm1 *Calculate* `exp(x) - 1` *for all elements in the tensor.*

Description

Calculate `exp(x) - 1` for all elements in the tensor.

Usage

```
op_expm1(x)
```

Arguments

`x`	Input values.

Value

Output tensor, element-wise exponential minus one.

See Also

- https://keras.io/api/ops/numpy#expm1-function

Other numpy ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()

op_bincount()
op_broadcast_to()
op_ceil()
op_clip()
op_concatenate()
op_conj()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_cumprod()
op_cumsum()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_einsum()
op_empty()
op_equal()
op_exp()
op_expand_dims()
op_eye()
op_flip()
op_floor()
op_floor_divide()
op_full()
op_full_like()
op_get_item()
op_greater()
op_greater_equal()
op_hstack()
op_identity()
op_imag()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()

op_log2()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_matmul()
op_max()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moveaxis()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_not_equal()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_repeat()
op_reshape()
op_roll()
op_round()
op_sign()
op_sin()
op_sinh()
op_size()
op_sort()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()

op_subtract()
op_sum()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_var()
op_vdot()
op_vstack()
op_where()
op_zeros()
op_zeros_like()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()

op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()

op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()

op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()

op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

---

op_extract_sequences *Expands the dimension of last axis into sequences of* sequence_length.

Description

Slides a window of size `sequence_length` over the last axis of the input with a stride of `sequence_stride`, replacing the last axis with `[num_sequences, sequence_length]` sequences.

If the dimension along the last axis is N, the number of sequences can be computed by:
`num_sequences = 1 + (N - sequence_length) // sequence_stride`

Usage

```
op_extract_sequences(x, sequence_length, sequence_stride)
```

Arguments

`x`	Input tensor.
`sequence_length`	An integer representing the sequences length.
`sequence_stride`	An integer representing the sequences hop size.

Value

A tensor of sequences with shape `[..., num_sequences, sequence_length]`.

Examples

```
x <- op_convert_to_tensor(1:6)
op_extract_sequences(x, 3, 2)

## tf.Tensor(
## [[1 2 3]
##  [3 4 5]], shape=(2, 3), dtype=int32)
```

See Also

- https://keras.io/api/ops/core#extractsequences-function

Other math ops:
`op_erf()`
`op_erfinv()`
`op_fft()`
`op_fft2()`
`op_in_top_k()`
`op_irfft()`
`op_istft()`
`op_logsumexp()`
`op_qr()`
`op_rfft()`
`op_rsqrt()`
`op_segment_max()`

op_segment_sum()
op_solve()
op_stft()
op_top_k()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()

op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()

op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()

op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()

op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

---

op_eye *Return a 2-D tensor with ones on the diagonal and zeros elsewhere.*

Description

Return a 2-D tensor with ones on the diagonal and zeros elsewhere.

Usage

```
op_eye(N, M = NULL, k = 0L, dtype = NULL)
```

Arguments

N	Number of rows in the output.
M	Number of columns in the output. If NULL, defaults to N.
k	Index of the diagonal: 0 (the default) refers to the main diagonal, a positive value refers to an upper diagonal, and a negative value to a lower diagonal.
dtype	Data type of the returned tensor.

Value

Tensor with ones on the k-th diagonal and zeros elsewhere.

See Also

- https://keras.io/api/ops/numpy#eye-function

Other numpy ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_bincount()
op_broadcast_to()
op_ceil()
op_clip()
op_concatenate()
op_conj()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_cumprod()
op_cumsum()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_einsum()
op_empty()

op_equal()
op_exp()
op_expand_dims()
op_expm1()
op_flip()
op_floor()
op_floor_divide()
op_full()
op_full_like()
op_get_item()
op_greater()
op_greater_equal()
op_hstack()
op_identity()
op_imag()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_matmul()
op_max()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moveaxis()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_not_equal()

op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_repeat()
op_reshape()
op_roll()
op_round()
op_sign()
op_sin()
op_sinh()
op_size()
op_sort()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_subtract()
op_sum()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_var()
op_vdot()
op_vstack()
op_where()
op_zeros()
op_zeros_like()

Other ops:
op_abs()

op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()

op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()

op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()

op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()

op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

---

op_fft *Computes the Fast Fourier Transform along last axis of input.*

Description

Computes the Fast Fourier Transform along last axis of input.

Usage

```
op_fft(x)
```

Arguments

x	list of the real and imaginary parts of the input tensor. Both tensors provided should be of floating type.

Value

A list containing two tensors - the real and imaginary parts of the output tensor.

Examples

```
x = c(op_array(c(1., 2.)),
      op_array(c(0., 1.)))
op_fft(x)
```

```
## [[1]]
## tf.Tensor([ 3. -1.], shape=(2), dtype=float32)
##
## [[2]]
## tf.Tensor([ 1. -1.], shape=(2), dtype=float32)
```

See Also

- https://keras.io/api/ops/fft#fft-function

Other math ops:
op_erf()
op_erfinv()
op_extract_sequences()
op_fft2()
op_in_top_k()
op_irfft()
op_istft()
op_logsumexp()
op_qr()
op_rfft()
op_rsqrt()
op_segment_max()
op_segment_sum()
op_solve()
op_stft()
op_top_k()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()

op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft2()

op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()

op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()

op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

`op_fft2` *Computes the 2D Fast Fourier Transform along the last two axes of input.*

Description

Computes the 2D Fast Fourier Transform along the last two axes of input.

Usage

```
op_fft2(x)
```

Arguments

`x`	list of the real and imaginary parts of the input tensor. Both tensors provided should be of floating type.

Value

A list containing two tensors - the real and imaginary parts of the output.

Examples

```
x <- c(op_array(rbind(c(1, 2),
                      c(2, 1))),
       op_array(rbind(c(0, 1),
                      c(1, 0))))
op_fft2(x)
```

```
## [[1]]
## tf.Tensor(
## [[ 6.  0.]
##  [ 0. -2.]], shape=(2, 2), dtype=float64)
##
## [[2]]
## tf.Tensor(
## [[ 2.  0.]
##  [ 0. -2.]], shape=(2, 2), dtype=float64)
```

See Also

- https://keras.io/api/ops/fft#fft2-function

Other math ops:
op_erf()
op_erfinv()
op_extract_sequences()
op_fft()
op_in_top_k()
op_irfft()
op_istft()
op_logsumexp()
op_qr()
op_rfft()
op_rsqrt()
op_segment_max()
op_segment_sum()
op_solve()
op_stft()
op_top_k()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()

op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()

op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()

op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()

op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

op_flip *Reverse the order of elements in the tensor along the given axis.*

Description

The shape of the tensor is preserved, but the elements are reordered.

Usage

```
op_flip(x, axis = NULL)
```

Arguments

x	Input tensor.
axis	Axis or axes along which to flip the tensor. The default, axis = NULL, will flip over all of the axes of the input tensor.

Value

Output tensor with entries of axis reversed.

See Also

- https://keras.io/api/ops/numpy#flip-function

Other numpy ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_bincount()
op_broadcast_to()
op_ceil()
op_clip()
op_concatenate()
op_conj()
op_copy()
op_correlate()
op_cos()

op_cosh()
op_count_nonzero()
op_cross()
op_cumprod()
op_cumsum()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_einsum()
op_empty()
op_equal()
op_exp()
op_expand_dims()
op_expm1()
op_eye()
op_floor()
op_floor_divide()
op_full()
op_full_like()
op_get_item()
op_greater()
op_greater_equal()
op_hstack()
op_identity()
op_imag()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_matmul()
op_max()

op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moveaxis()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_not_equal()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_repeat()
op_reshape()
op_roll()
op_round()
op_sign()
op_sin()
op_sinh()
op_size()
op_sort()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_subtract()
op_sum()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()

op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_var()
op_vdot()
op_vstack()
op_where()
op_zeros()
op_zeros_like()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()

op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()

op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()

op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()

op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

---

op_floor *Return the floor of the input, element-wise.*

Description

The floor of the scalar `x` is the largest integer `i`, such that `i <= x`.

Usage

```
op_floor(x)
```

Arguments

x	Input tensor.

Value

Output tensor, element-wise floor of x.

See Also

- https://keras.io/api/ops/numpy#floor-function

Other numpy ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_bincount()
op_broadcast_to()
op_ceil()
op_clip()
op_concatenate()
op_conj()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_cumprod()
op_cumsum()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_einsum()
op_empty()

op_equal()
op_exp()
op_expand_dims()
op_expm1()
op_eye()
op_flip()
op_floor_divide()
op_full()
op_full_like()
op_get_item()
op_greater()
op_greater_equal()
op_hstack()
op_identity()
op_imag()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_matmul()
op_max()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moveaxis()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_not_equal()

op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_repeat()
op_reshape()
op_roll()
op_round()
op_sign()
op_sin()
op_sinh()
op_size()
op_sort()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_subtract()
op_sum()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_var()
op_vdot()
op_vstack()
op_where()
op_zeros()
op_zeros_like()

Other ops:
op_abs()

op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()

op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()

op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()

op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()

op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

---

op_floor_divide *Returns the largest integer smaller or equal to the division of inputs.*

Description

Note that this function is automatically called when using the R operator `%/%` with a tensor.

```
(x <- op_arange(10))
```

```
## tf.Tensor([0. 1. 2. 3. 4. 5. 6. 7. 8. 9.], shape=(10), dtype=float64)
```

```
op_floor_divide(x, 2)
```

```
## tf.Tensor([0. 0. 1. 1. 2. 2. 3. 3. 4. 4.], shape=(10), dtype=float64)
```

```
x %/% 2
```

```
## tf.Tensor([0. 0. 1. 1. 2. 2. 3. 3. 4. 4.], shape=(10), dtype=float64)
```

Usage

```
op_floor_divide(x1, x2)
```

Arguments

x1	Numerator.
x2	Denominator.

Value

Output tensor, y <- `floor(x1/x2)`

See Also

Other numpy ops:
`op_abs()`
`op_add()`
`op_all()`
`op_any()`
`op_append()`
`op_arange()`
`op_arccos()`
`op_arccosh()`
`op_arcsin()`
`op_arcsinh()`
`op_arctan()`
`op_arctan2()`
`op_arctanh()`
`op_argmax()`
`op_argmin()`
`op_argsort()`
`op_array()`
`op_average()`
`op_bincount()`
`op_broadcast_to()`
`op_ceil()`
`op_clip()`
`op_concatenate()`
`op_conj()`
`op_copy()`
`op_correlate()`
`op_cos()`
`op_cosh()`
`op_count_nonzero()`
`op_cross()`
`op_cumprod()`
`op_cumsum()`
`op_diag()`
`op_diagonal()`
`op_diff()`
`op_digitize()`
`op_divide()`

op_divide_no_nan()
op_dot()
op_einsum()
op_empty()
op_equal()
op_exp()
op_expand_dims()
op_expm1()
op_eye()
op_flip()
op_floor()
op_full()
op_full_like()
op_get_item()
op_greater()
op_greater_equal()
op_hstack()
op_identity()
op_imag()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_matmul()
op_max()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moveaxis()
op_multiply()
op_nan_to_num()

op_ndim()
op_negative()
op_nonzero()
op_not_equal()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_repeat()
op_reshape()
op_roll()
op_round()
op_sign()
op_sin()
op_sinh()
op_size()
op_sort()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_subtract()
op_sum()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_var()
op_vdot()
op_vstack()
op_where()
op_zeros()

op_zeros_like()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()

op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()

op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()

op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()

op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

---

op_fori_loop *For loop implementation.*

Description

For loop implementation.

Usage

```
op_fori_loop(lower, upper, body_fun, init_val)
```

Arguments

lower	The initial value of the loop variable.
upper	The upper bound of the loop variable.
body_fun	A callable that represents the loop body. Must take two arguments: the loop variable and the loop state. The loop state should be updated and returned by this function.
init_val	The initial value of the loop state.

Value

The final state after the loop.

Examples

```
lower <- 0L
upper <- 10L
body_fun <- function(i, state) state + i
init_state <- 0L
final_state <- op_fori_loop(lower, upper, body_fun, init_state)
final_state

## tf.Tensor(45, shape=(), dtype=int32)
```

See Also

Other core ops:
op_cast()
op_cond()
op_convert_to_numpy()
op_convert_to_tensor()
op_custom_gradient()
op_is_tensor()
op_scatter()
op_scatter_update()
op_shape()
op_slice()
op_slice_update()
op_stop_gradient()
op_unstack()
op_vectorized_map()
op_while_loop()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()

op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()

op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()

op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()

op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()

op_zeros()
op_zeros_like()

---

op_full *Return a new tensor of given shape and type, filled with* fill_value*.*

Description

Return a new tensor of given shape and type, filled with fill_value.

Usage

```
op_full(shape, fill_value, dtype = NULL)
```

Arguments

shape	Shape of the new tensor.
fill_value	Fill value.
dtype	Desired data type of the tensor.

Value

Output tensor.

See Also

- https://keras.io/api/ops/numpy#full-function

Other numpy ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()

op_bincount()
op_broadcast_to()
op_ceil()
op_clip()
op_concatenate()
op_conj()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_cumprod()
op_cumsum()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_einsum()
op_empty()
op_equal()
op_exp()
op_expand_dims()
op_expm1()
op_eye()
op_flip()
op_floor()
op_floor_divide()
op_full_like()
op_get_item()
op_greater()
op_greater_equal()
op_hstack()
op_identity()
op_imag()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()

op_log2()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_matmul()
op_max()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moveaxis()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_not_equal()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_repeat()
op_reshape()
op_roll()
op_round()
op_sign()
op_sin()
op_sinh()
op_size()
op_sort()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()

op_subtract()
op_sum()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_var()
op_vdot()
op_vstack()
op_where()
op_zeros()
op_zeros_like()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()

op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full_like()
op_gelu()

op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()

op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()

op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

op_full_like   *Return a full tensor with the same shape and type as the given tensor.*

Description

Return a full tensor with the same shape and type as the given tensor.

Usage

```
op_full_like(x, fill_value, dtype = NULL)
```

Arguments

`x`	Input tensor.
`fill_value`	Fill value.
`dtype`	Overrides data type of the result.

Value

Tensor of `fill_value` with the same shape and type as `x`.

See Also

- `https://keras.io/api/ops/numpy#fulllike-function`

Other numpy ops:
`op_abs()`
`op_add()`
`op_all()`
`op_any()`
`op_append()`
`op_arange()`
`op_arccos()`
`op_arccosh()`
`op_arcsin()`
`op_arcsinh()`
`op_arctan()`
`op_arctan2()`
`op_arctanh()`
`op_argmax()`
`op_argmin()`
`op_argsort()`
`op_array()`
`op_average()`
`op_bincount()`
`op_broadcast_to()`
`op_ceil()`
`op_clip()`
`op_concatenate()`
`op_conj()`
`op_copy()`
`op_correlate()`
`op_cos()`

op_cosh()
op_count_nonzero()
op_cross()
op_cumprod()
op_cumsum()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_einsum()
op_empty()
op_equal()
op_exp()
op_expand_dims()
op_expm1()
op_eye()
op_flip()
op_floor()
op_floor_divide()
op_full()
op_get_item()
op_greater()
op_greater_equal()
op_hstack()
op_identity()
op_imag()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_matmul()
op_max()

op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moveaxis()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_not_equal()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_repeat()
op_reshape()
op_roll()
op_round()
op_sign()
op_sin()
op_sinh()
op_size()
op_sort()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_subtract()
op_sum()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()

op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_var()
op_vdot()
op_vstack()
op_where()
op_zeros()
op_zeros_like()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()

op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()

op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()

op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()

op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

---

op_gelu *Gaussian Error Linear Unit (GELU) activation function.*

Description

If approximate is TRUE, it is defined as: f(x) = 0.5 * x * (1 + tanh(sqrt(2 / pi) * (x + 0.044715 * x^3)))

Or if approximate is FALSE, it is defined as: f(x) = x * P(X <= x) = 0.5 * x * (1 + erf(x / sqrt(2))), where P(X) ~ N(0, 1).

Usage

```
op_gelu(x, approximate = TRUE)
```

Arguments

`x`	Input tensor.
`approximate`	Approximate version of GELU activation. Defaults to `TRUE`.

Value

A tensor with the same shape as `x`.

Examples

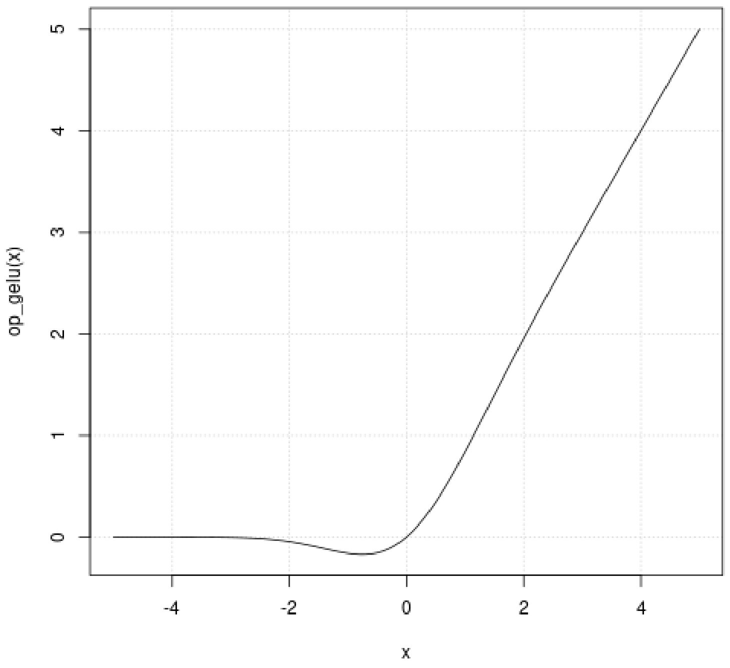
```
x <- op_array(c(-1., 0., 1.))
op_gelu(x)
```

```
## tf.Tensor([-0.15880796  0.          0.841192  ], shape=(3), dtype=float32)
```

```
op_gelu(x, FALSE)
```

```
## tf.Tensor([-0.15865526  0.          0.8413447 ], shape=(3), dtype=float32)
```

```
x <- seq(-5, 5, .1)
plot(x, op_gelu(x),
     type = "l", #, frame.plot = FALSE,
     panel.first = grid())
```



See Also

- https://keras.io/api/ops/nn#gelu-function

Other nn ops:
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_categorical_crossentropy()
op_conv()
op_conv_transpose()
op_ctc_loss()
op_depthwise_conv()
op_elu()
op_hard_sigmoid()
op_hard_silu()
op_leaky_relu()
op_log_sigmoid()
op_log_softmax()
op_max_pool()
op_moments()
op_multi_hot()
op_normalize()
op_one_hot()
op_relu()
op_relu6()
op_selu()
op_separable_conv()
op_sigmoid()
op_silu()
op_softmax()
op_softplus()
op_softsign()
op_sparse_categorical_crossentropy()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()

op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()

op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()

op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()

op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()

op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

---

op_get_item *Return* x[key].

Description

Return x[key].

Usage

```
op_get_item(x, key)
```

Arguments

x	A dictionary-like object
key	Generally, a string, but most object with a __hash__ method are acceptable.

Value

key.

Note

Generally, calling x[[key]] or x$key is preferable.

See Also

- https://keras.io/api/ops/numpy#getitem-function

Other numpy ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()

op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_bincount()
op_broadcast_to()
op_ceil()
op_clip()
op_concatenate()
op_conj()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_cumprod()
op_cumsum()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_einsum()
op_empty()
op_equal()
op_exp()
op_expand_dims()
op_expm1()
op_eye()
op_flip()
op_floor()
op_floor_divide()
op_full()
op_full_like()
op_greater()
op_greater_equal()
op_hstack()
op_identity()
op_imag()
op_isclose()
op_isfinite()
op_isinf()

op_isnan()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_matmul()
op_max()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moveaxis()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_not_equal()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_repeat()
op_reshape()
op_roll()
op_round()
op_sign()
op_sin()
op_sinh()
op_size()

op_sort()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_subtract()
op_sum()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_var()
op_vdot()
op_vstack()
op_where()
op_zeros()
op_zeros_like()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()

op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()

op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()

op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()

op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

op_greater *Return the truth value of* x1 > x2 *element-wise.*

Description

Note that this function is automatically called when using the R operator > with a tensor.

```
(x <- op_arange(4))

## tf.Tensor([0. 1. 2. 3.], shape=(4), dtype=float64)


op_greater(x, 2)

## tf.Tensor([False False False  True], shape=(4), dtype=bool)


x > 2

## tf.Tensor([False False False  True], shape=(4), dtype=bool)
```

Usage

```
op_greater(x1, x2)
```

Arguments

x1	First input tensor.
x2	Second input tensor.

Value

Output tensor, element-wise comparison of x1 and x2.

See Also

- https://keras.io/api/ops/numpy#greater-function

Other numpy ops:
op_abs()
op_add()
op_all()
op_any()
op_append()

op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_bincount()
op_broadcast_to()
op_ceil()
op_clip()
op_concatenate()
op_conj()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_cumprod()
op_cumsum()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_einsum()
op_empty()
op_equal()
op_exp()
op_expand_dims()
op_expm1()
op_eye()
op_flip()
op_floor()
op_floor_divide()
op_full()
op_full_like()
op_get_item()
op_greater_equal()

op_hstack()
op_identity()
op_imag()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_matmul()
op_max()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moveaxis()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_not_equal()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_repeat()
op_reshape()

op_roll()
op_round()
op_sign()
op_sin()
op_sinh()
op_size()
op_sort()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_subtract()
op_sum()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_var()
op_vdot()
op_vstack()
op_where()
op_zeros()
op_zeros_like()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()

op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()

op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()

op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()

op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()

op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

---

op_greater_equal *Return the truth value of* x1 >= x2 *element-wise.*

Description

Note that this function is automatically called when using the R operator >= with a tensor.

```
(x <- op_arange(4))
```

```
## tf.Tensor([0. 1. 2. 3.], shape=(4), dtype=float64)
```

```
op_greater_equal(x, 2)
```

```
## tf.Tensor([False False  True  True], shape=(4), dtype=bool)
```

```
x >= 2
```

```
## tf.Tensor([False False  True  True], shape=(4), dtype=bool)
```

Usage

```
op_greater_equal(x1, x2)
```

Arguments

x1	First input tensor.
x2	Second input tensor.

Value

Output tensor, element-wise comparison of x1 and x2.

See Also

- https://keras.io/api/ops/numpy#greaterequal-function

Other numpy ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_bincount()
op_broadcast_to()
op_ceil()
op_clip()
op_concatenate()
op_conj()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_cumprod()
op_cumsum()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_einsum()
op_empty()
op_equal()
op_exp()
op_expand_dims()
op_expm1()

op_eye()
op_flip()
op_floor()
op_floor_divide()
op_full()
op_full_like()
op_get_item()
op_greater()
op_hstack()
op_identity()
op_imag()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_matmul()
op_max()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moveaxis()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_not_equal()
op_ones()
op_ones_like()
op_outer()
op_pad()

op_power()
op_prod()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_repeat()
op_reshape()
op_roll()
op_round()
op_sign()
op_sin()
op_sinh()
op_size()
op_sort()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_subtract()
op_sum()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_var()
op_vdot()
op_vstack()
op_where()
op_zeros()
op_zeros_like()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()

op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()

op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()

op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()

op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()

op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

op_hard_sigmoid *Hard sigmoid activation function.*

Description

It is defined as:
`0 if x < -2.5`, `1 if x > 2.5`, `(0.2 * x) + 0.5 if -2.5 <= x <= 2.5`.

Usage

```
op_hard_sigmoid(x)
```

Arguments

x	Input tensor.

Value

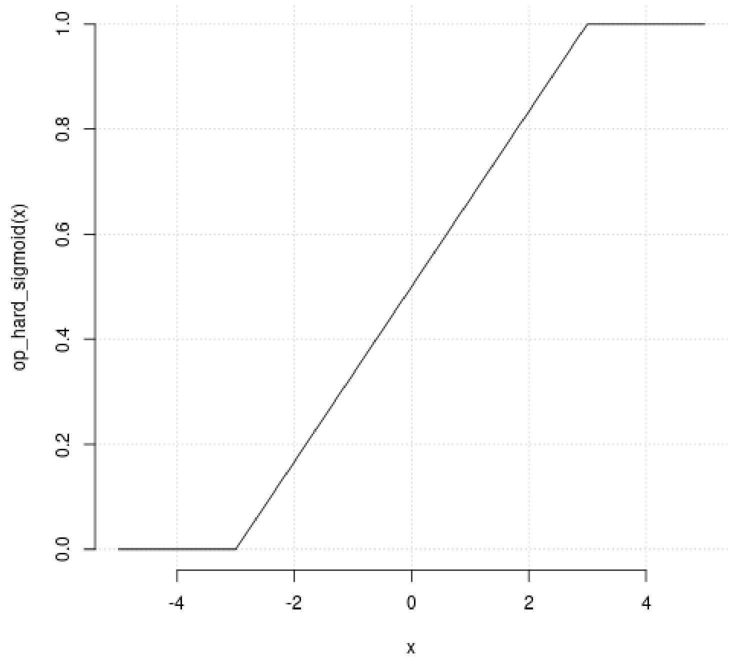
A tensor with the same shape as x.

Examples

```
x <- op_array(c(-1., 0., 1.))
op_hard_sigmoid(x)

## tf.Tensor([0.33333334 0.5        0.6666667 ], shape=(3), dtype=float32)


x <- as.array(seq(-5, 5, .1))
plot(x, op_hard_sigmoid(x),
     type = 'l', panel.first = grid(), frame.plot = FALSE)
```



See Also

- https://keras.io/api/ops/nn#hardsigmoid-function

Other nn ops:
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_categorical_crossentropy()
op_conv()
op_conv_transpose()
op_ctc_loss()
op_depthwise_conv()
op_elu()
op_gelu()
op_hard_silu()
op_leaky_relu()
op_log_sigmoid()
op_log_softmax()
op_max_pool()
op_moments()
op_multi_hot()
op_normalize()
op_one_hot()

op_relu()
op_relu6()
op_selu()
op_separable_conv()
op_sigmoid()
op_silu()
op_softmax()
op_softplus()
op_softsign()
op_sparse_categorical_crossentropy()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()

op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()

op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()

op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()

op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

---

op_hard_silu *Hard SiLU activation function, also known as Hard Swish.*

Description

It is defined as:

- `0` if if `x < -3`
- `x` if `x > 3`
- `x * (x + 3) / 6` if `-3 <= x <= 3`

It's a faster, piecewise linear approximation of the silu activation.

Usage

```
op_hard_silu(x)

op_hard_swish(x)
```

Arguments

x	Input tensor.

Value

A tensor with the same shape as x.

Examples

```
x <- op_convert_to_tensor(c(-3.0, -1.0, 0.0, 1.0, 3.0))
op_hard_silu(x)

## tf.Tensor([-0.      -0.33333334  0.       0.6666667   3.      ], shape=(5), dtype=float32)
```

See Also

Other nn ops:
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_categorical_crossentropy()
op_conv()
op_conv_transpose()
op_ctc_loss()
op_depthwise_conv()
op_elu()
op_gelu()
op_hard_sigmoid()
op_leaky_relu()
op_log_sigmoid()
op_log_softmax()
op_max_pool()
op_moments()
op_multi_hot()
op_normalize()
op_one_hot()
op_relu()
op_relu6()
op_selu()
op_separable_conv()
op_sigmoid()
op_silu()

op_softmax()
op_softplus()
op_softsign()
op_sparse_categorical_crossentropy()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()

op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()

op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()

op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()

op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

---

op_hstack *Stack tensors in sequence horizontally (column wise).*

Description

This is equivalent to concatenation along the first axis for 1-D tensors, and along the second axis for all other tensors.

Usage

```
op_hstack(xs)
```

Arguments

xs	Sequence of tensors.

Value

The tensor formed by stacking the given tensors.

See Also

- https://keras.io/api/ops/numpy#hstack-function

Other numpy ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_bincount()
op_broadcast_to()
op_ceil()
op_clip()
op_concatenate()
op_conj()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_cumprod()
op_cumsum()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_einsum()
op_empty()
op_equal()
op_exp()
op_expand_dims()
op_expm1()

op_eye()
op_flip()
op_floor()
op_floor_divide()
op_full()
op_full_like()
op_get_item()
op_greater()
op_greater_equal()
op_identity()
op_imag()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_matmul()
op_max()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moveaxis()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_not_equal()
op_ones()
op_ones_like()
op_outer()
op_pad()

op_power()
op_prod()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_repeat()
op_reshape()
op_roll()
op_round()
op_sign()
op_sin()
op_sinh()
op_size()
op_sort()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_subtract()
op_sum()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_var()
op_vdot()
op_vstack()
op_where()
op_zeros()
op_zeros_like()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()

op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()

op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()

op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()

op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()

op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

---

op_identity *Return the identity tensor.*

Description

The identity tensor is a square tensor with ones on the main diagonal and zeros elsewhere.

Usage

```
op_identity(n, dtype = NULL)
```

Arguments

n	Number of rows (and columns) in the `n x n` output tensor.
dtype	Data type of the output tensor.

Value

The identity tensor.

See Also

- https://keras.io/api/ops/numpy#identity-function

Other numpy ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()

op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_bincount()
op_broadcast_to()
op_ceil()
op_clip()
op_concatenate()
op_conj()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_cumprod()
op_cumsum()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_einsum()
op_empty()
op_equal()
op_exp()
op_expand_dims()
op_expm1()
op_eye()
op_flip()
op_floor()
op_floor_divide()
op_full()
op_full_like()
op_get_item()
op_greater()
op_greater_equal()
op_hstack()

op_imag()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_matmul()
op_max()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moveaxis()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_not_equal()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_repeat()
op_reshape()
op_roll()
op_round()

op_sign()
op_sin()
op_sinh()
op_size()
op_sort()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_subtract()
op_sum()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_var()
op_vdot()
op_vstack()
op_where()
op_zeros()
op_zeros_like()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()

op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()

op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()

op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()

op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()

op_where()
op_while_loop()
op_zeros()
op_zeros_like()

op_imag *Return the imaginary part of the complex argument.*

Description

Return the imaginary part of the complex argument.

Usage

```
op_imag(x)
```

Arguments

x	Input tensor.

Value

The imaginary component of the complex argument.

See Also

- https://keras.io/api/ops/numpy#imag-function

Other numpy ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()

op_bincount()
op_broadcast_to()
op_ceil()
op_clip()
op_concatenate()
op_conj()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_cumprod()
op_cumsum()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_einsum()
op_empty()
op_equal()
op_exp()
op_expand_dims()
op_expm1()
op_eye()
op_flip()
op_floor()
op_floor_divide()
op_full()
op_full_like()
op_get_item()
op_greater()
op_greater_equal()
op_hstack()
op_identity()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()

op_log2()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_matmul()
op_max()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moveaxis()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_not_equal()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_repeat()
op_reshape()
op_roll()
op_round()
op_sign()
op_sin()
op_sinh()
op_size()
op_sort()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()

op_subtract()
op_sum()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_var()
op_vdot()
op_vstack()
op_where()
op_zeros()
op_zeros_like()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()

op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()

op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()

op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()

op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

---

op_image_affine_transform

Applies the given transform(s) to the image(s).

Description

Applies the given transform(s) to the image(s).

Usage

```
op_image_affine_transform(
  image,
  transform,
  interpolation = "bilinear",
  fill_mode = "constant",
  fill_value = 0L,
  data_format = "channels_last"
)
```

Arguments

`image`	Input image or batch of images. Must be 3D or 4D.
`transform`	Projective transform matrix/matrices. A vector of length 8 or tensor of size N x 8. If one row of transform is `[a0, a1, a2, b0, b1, b2, c0, c1]`, then it maps the output point `(x, y)` to a transformed input point `(x', y') = ((a0 x + a1 y + a2) / k, (b0 x + b1 y` where `k = c0 x + c1 y + 1`. The transform is inverted compared to the transform mapping input points to output points. Note that gradients are not backpropagated into transformation parameters. Note that `c0` and `c1` are only effective when using TensorFlow backend and will be considered as `0` when using other backends.
`interpolation`	Interpolation method. Available methods are `"nearest"`, and `"bilinear"`. Defaults to `"bilinear"`.
`fill_mode`	Points outside the boundaries of the input are filled according to the given mode. Available methods are `"constant"`, `"nearest"`, `"wrap"` and `"reflect"`. Defaults to `"constant"`. • `"reflect"`: `(d c b a | a b c d | d c b a)` The input is extended by reflecting about the edge of the last pixel. • `"constant"`: `(k k k k | a b c d | k k k k)` The input is extended by filling all values beyond the edge with the same constant value k specified by `fill_value`. • `"wrap"`: `(a b c d | a b c d | a b c d)` The input is extended by wrapping around to the opposite edge. • `"nearest"`: `(a a a a | a b c d | d d d d)` The input is extended by the nearest pixel.
`fill_value`	Value used for points outside the boundaries of the input if `fill_mode = "constant"`. Defaults to `0`.
`data_format`	string, either `"channels_last"` or `"channels_first"`. The ordering of the dimensions in the inputs. `"channels_last"` corresponds to inputs with shape `(batch, height, width, channels)` while `"channels_first"` corresponds to inputs with shape `(batch, channels, height, weight)`. It defaults to the `image_data_format` value found in your Keras config file at `~/.keras/keras.json`. If you never set it, then it will be `"channels_last"`.

Value

Applied affine transform image or batch of images.

Examples

```
x <- random_uniform(c(2, 64, 80, 3)) # batch of 2 RGB images
transform <- op_array(rbind(c(1.5, 0, -20, 0, 1.5, -16, 0, 0),  # zoom
                           c(1, 0, -20, 0, 1, -16, 0, 0)))  # translation))
y <- op_image_affine_transform(x, transform)
shape(y)

## shape(2, 64, 80, 3)


# (2, 64, 80, 3)

x <- random_uniform(c(64, 80, 3)) # single RGB image
transform <- op_array(c(1.0, 0.5, -20, 0.5, 1.0, -16, 0, 0))  # shear
y <- op_image_affine_transform(x, transform)
shape(y)

## shape(64, 80, 3)


# (64, 80, 3)

x <- random_uniform(c(2, 3, 64, 80)) # batch of 2 RGB images
transform <- op_array(rbind(
  c(1.5, 0,-20, 0, 1.5,-16, 0, 0),  # zoom
  c(1, 0,-20, 0, 1,-16, 0, 0)  # translation
))
y <- op_image_affine_transform(x, transform, data_format = "channels_first")
shape(y)

## shape(2, 3, 64, 80)


# (2, 3, 64, 80)
```

See Also

- https://keras.io/api/ops/image#affinetransform-function

Other image ops:
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()

op_image_resize()

Other image utils:
image_array_save()
image_from_array()
image_load()
image_smart_resize()
image_to_array()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()

op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()

op_identity()
op_imag()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()

op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()

op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

---

op_image_crop *Crop* images *to a specified* height *and* width.

Description

Crop images to a specified height and width.

Usage

```
op_image_crop(
  images,
  top_cropping = NULL,
  left_cropping = NULL,
  target_height = NULL,
  target_width = NULL,
  bottom_cropping = NULL,
  right_cropping = NULL
)
```

Arguments

`images`	4-D batch of images of shape `(batch, height, width, channels)` or 3-D single image of shape `(height, width, channels)`.
`top_cropping`	Number of columns to crop from the top.
`left_cropping`	Number of columns to crop from the left.
`target_height`	Height of the output images.
`target_width`	Width of the output images.
`bottom_cropping`	Number of columns to crop from the bottom.
`right_cropping`	Number of columns to crop from the right.

Value

If images were 4D, a 4D float Tensor of shape `(batch, target_height, target_width, channels)`
If images were 3D, a 3D float Tensor of shape `(target_height, target_width, channels)`

Examples

```
images <- op_reshape(op_arange(1, 28, dtype="float32"), c(3, 3, 3))
images[, , 1] # print the first channel of the images

cropped_images <- op_image_crop(images, 0, 0, 2, 2)
cropped_images[, , 1] # print the first channel of the cropped images
```

See Also

- https://www.tensorflow.org/api_docs/python/tf/keras/ops/image/crop_images

Other image ops:
`op_image_affine_transform()`
`op_image_extract_patches()`
`op_image_map_coordinates()`
`op_image_pad()`
`op_image_resize()`

Other image utils:
`image_array_save()`
`image_from_array()`
`image_load()`

image_smart_resize()
image_to_array()
op_image_affine_transform()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()

op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()

op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()

op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()

op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

---

op_image_extract_patches

Extracts patches from the image(s).

Description

Extracts patches from the image(s).

Usage

```
op_image_extract_patches(
  image,
  size,
  strides = NULL,
  dilation_rate = 1L,
  padding = "valid",
  data_format = "channels_last"
)
```

Arguments

image	Input image or batch of images. Must be 3D or 4D.
size	Patch size int or list (patch_height, patch_width)
strides	strides along height and width. If not specified, or if NULL, it defaults to the same value as size.
dilation_rate	This is the input stride, specifying how far two consecutive patch samples are in the input. For value other than 1, strides must be 1. NOTE: strides > 1 is not supported in conjunction with dilation_rate > 1
padding	The type of padding algorithm to use: "same" or "valid".
data_format	string, either "channels_last" or "channels_first". The ordering of the dimensions in the inputs. "channels_last" corresponds to inputs with shape (batch, height, width, channels) while "channels_first" corresponds to inputs with shape (batch, channels, height, weight). It defaults to the image_data_format value found in your Keras config file at ~/.keras/keras.json. If you never set it, then it will be "channels_last".

Value

Extracted patches 3D (if not batched) or 4D (if batched)

Examples

```
image <- random_uniform(c(2, 20, 20, 3), dtype = "float32") # batch of 2 RGB images
patches <- op_image_extract_patches(image, c(5, 5))
shape(patches)

## shape(2, 4, 4, 75)


# (2, 4, 4, 75)
image <- random_uniform(c(20, 20, 3), dtype = "float32") # 1 RGB image
patches <- op_image_extract_patches(image, c(3, 3), c(1, 1))
shape(patches)

## shape(18, 18, 27)


# (18, 18, 27)
```

See Also

- https://keras.io/api/ops/image#extractpatches-function

Other image ops:
op_image_affine_transform()
op_image_crop()
op_image_map_coordinates()

op_image_pad()
op_image_resize()

Other image utils:
image_array_save()
image_from_array()
image_load()
image_smart_resize()
image_to_array()
op_image_affine_transform()
op_image_crop()
op_image_map_coordinates()
op_image_pad()
op_image_resize()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()

op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()

op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()

op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()

`op_sort()`
`op_sparse_categorical_crossentropy()`
`op_split()`
`op_sqrt()`
`op_square()`
`op_squeeze()`
`op_stack()`
`op_std()`
`op_stft()`
`op_stop_gradient()`
`op_subtract()`
`op_sum()`
`op_svd()`
`op_swapaxes()`
`op_take()`
`op_take_along_axis()`
`op_tan()`
`op_tanh()`
`op_tensordot()`
`op_tile()`
`op_top_k()`
`op_trace()`
`op_transpose()`
`op_tri()`
`op_tril()`
`op_triu()`
`op_unstack()`
`op_var()`
`op_vdot()`
`op_vectorized_map()`
`op_vstack()`
`op_where()`
`op_while_loop()`
`op_zeros()`
`op_zeros_like()`

`op_image_map_coordinates`
Map the input array to new coordinates by interpolation..

Description

Note that interpolation near boundaries differs from the scipy function, because we fixed an outstanding bug scipy/issues/2640.

Usage

```
op_image_map_coordinates(
  input,
  coordinates,
  order,
  fill_mode = "constant",
  fill_value = 0L
)
```

Arguments

`input`	The input array.
`coordinates`	The coordinates at which input is evaluated.
`order`	The order of the spline interpolation. The order must be `0` or `1`. `0` indicates the nearest neighbor and `1` indicates the linear interpolation.
`fill_mode`	Points outside the boundaries of the input are filled according to the given mode. Available methods are `"constant"`, `"nearest"`, `"wrap"` and `"mirror"` and `"reflect"`. Defaults to `"constant"`. • `"constant"`: `(k k k k | a b c d | k k k k)` The input is extended by filling all values beyond the edge with the same constant value k specified by `fill_value`. • `"nearest"`: `(a a a a | a b c d | d d d d)` The input is extended by the nearest pixel. • `"wrap"`: `(a b c d | a b c d | a b c d)` The input is extended by wrapping around to the opposite edge. • `"mirror"`: `(c d c b | a b c d | c b a b)` The input is extended by mirroring about the edge. • `"reflect"`: `(d c b a | a b c d | d c b a)` The input is extended by reflecting about the edge of the last pixel.
`fill_value`	Value used for points outside the boundaries of the input if `fill_mode = "constant"`. Defaults to `0`.

Value

Output image or batch of images.

See Also

Other image ops:
`op_image_affine_transform()`
`op_image_crop()`
`op_image_extract_patches()`
`op_image_pad()`
`op_image_resize()`

Other image utils:
`image_array_save()`

image_from_array()
image_load()
image_smart_resize()
image_to_array()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_pad()
op_image_resize()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()

op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()

op_image_extract_patches()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()

op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()

op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

---

op_image_pad *Pad* images *with zeros to the specified* height *and* width.

Description

Pad images with zeros to the specified height and width.

Usage

```
op_image_pad(
  images,
  top_padding = NULL,
  left_padding = NULL,
  target_height = NULL,
  target_width = NULL,
  bottom_padding = NULL,
  right_padding = NULL
)
```

Arguments

`images`	4D Tensor of shape `(batch, height, width, channels)` or 3D Tensor of shape `(height, width, channels)`.
`top_padding`	Number of rows of zeros to add on top.
`left_padding`	Number of columns of zeros to add on the left.
`target_height`	Height of output images.
`target_width`	Width of output images.
`bottom_padding`	Number of rows of zeros to add at the bottom.
`right_padding`	Number of columns of zeros to add on the right.

Value

- If `images` were 4D, a 4D float Tensor of shape `(batch, target_height, target_width, channels)`
- If `images` were 3D, a 3D float Tensor of shape `(target_height, target_width, channels)`

Examples

```
images <- random_uniform(c(15, 25, 3))
padded_images <- op_image_pad(
    images, 2, 3, target_height = 20, target_width = 30
)
shape(padded_images)

## shape(20, 30, 3)


batch_images <- random_uniform(c(2, 15, 25, 3))
padded_batch <- op_image_pad(batch_images, 2, 3,
                             target_height = 20,
                             target_width = 30)
shape(padded_batch)

## shape(2, 20, 30, 3)
```

See Also

Other image ops:
`op_image_affine_transform()`
`op_image_crop()`
`op_image_extract_patches()`
`op_image_map_coordinates()`

op_image_resize()

Other image utils:
image_array_save()
image_from_array()
image_load()
image_smart_resize()
image_to_array()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_resize()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()

op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()

op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()

op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()

op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

---

op_image_resize *Resize images to size using the specified interpolation method.*

Description

Resize images to size using the specified interpolation method.

Usage

```
op_image_resize(
  image,
  size,
  interpolation = "bilinear",
  antialias = FALSE,
  data_format = "channels_last"
)
```

Arguments

`image`	Input image or batch of images. Must be 3D or 4D.
`size`	Size of output image in `(height, width)` format.
`interpolation`	Interpolation method. Available methods are `"nearest"`, `"bilinear"`, and `"bicubic"`. Defaults to `"bilinear"`.
`antialias`	Whether to use an antialiasing filter when downsampling an image. Defaults to `FALSE`.
`data_format`	string, either `"channels_last"` or `"channels_first"`. The ordering of the dimensions in the inputs. `"channels_last"` corresponds to inputs with shape `(batch, height, width, channels)` while `"channels_first"` corresponds to inputs with shape `(batch, channels, height, weight)`. It defaults to the `image_data_format` value found in your Keras config file at `~/.keras/keras.json`. If you never set it, then it will be `"channels_last"`.

Value

Resized image or batch of images.

Examples

```
x <- random_uniform(c(2, 4, 4, 3)) # batch of 2 RGB images
y <- op_image_resize(x, c(2, 2))
shape(y)

## shape(2, 2, 2, 3)


x <- random_uniform(c(4, 4, 3)) # single RGB image
y <- op_image_resize(x, c(2, 2))
shape(y)

## shape(2, 2, 3)


x <- random_uniform(c(2, 3, 4, 4)) # batch of 2 RGB images
y <- op_image_resize(x, c(2, 2), data_format = "channels_first")
shape(y)

## shape(2, 3, 2, 2)
```

See Also

- https://keras.io/api/ops/image#resize-function

Other image ops:
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()

Other image utils:
image_array_save()
image_from_array()
image_load()
image_smart_resize()
image_to_array()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()

op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()

op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()

op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()

op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

op_inv *Computes the inverse of a square tensor.*

Description

Computes the inverse of a square tensor.

Usage

```
op_inv(x)
```

Arguments

`x`	Input tensor of shape `(..., M, M)`.

Value

A tensor of shape `(..., M, M)` representing the inverse of `x`.

See Also

Other linear algebra ops:
`op_cholesky()`
`op_det()`
`op_eig()`
`op_lu_factor()`
`op_norm()`
`op_solve_triangular()`
`op_svd()`

Other ops:
`op_abs()`
`op_add()`
`op_all()`
`op_any()`
`op_append()`
`op_arange()`
`op_arccos()`
`op_arccosh()`
`op_arcsin()`
`op_arcsinh()`
`op_arctan()`
`op_arctan2()`
`op_arctanh()`
`op_argmax()`
`op_argmin()`
`op_argsort()`
`op_array()`
`op_average()`
`op_average_pool()`
`op_batch_normalization()`
`op_binary_crossentropy()`

op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()

op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()

op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()

op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

op_in_top_k *Checks if the targets are in the top-k predictions.*

Description

Checks if the targets are in the top-k predictions.

Usage

```
op_in_top_k(targets, predictions, k)
```

Arguments

targets	A tensor of true labels.
predictions	A tensor of predicted labels.
k	An integer representing the number of predictions to consider.

Value

A boolean tensor of the same shape as `targets`, where each element indicates whether the corresponding target is in the top-k predictions.

Examples

```
targets <- op_array(c(2, 5, 3), "int32")
predictions <- op_array(dtype = "float32", rbind(
  c(0.1, 0.4, 0.6, 0.9, 0.5),
  c(0.1, 0.7, 0.9, 0.8, 0.3),
  c(0.1, 0.6, 0.9, 0.9, 0.5)
))
op_in_top_k(targets, predictions, k = 3L)

## tf.Tensor([ True False  True], shape=(3), dtype=bool)
```

See Also

- https://keras.io/api/ops/core#intopk-function

Other math ops:
`op_erf()`
`op_erfinv()`
`op_extract_sequences()`
`op_fft()`
`op_fft2()`
`op_irfft()`
`op_istft()`

op_logsumexp()
op_qr()
op_rfft()
op_rsqrt()
op_segment_max()
op_segment_sum()
op_solve()
op_stft()
op_top_k()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()

op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()

op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()

op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()

op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

---

op_irfft *Inverse real-valued Fast Fourier transform along the last axis.*

Description

Computes the inverse 1D Discrete Fourier Transform of a real-valued signal over the inner-most dimension of input.

The inner-most dimension of the input is assumed to be the result of RFFT: the `fft_length / 2 + 1` unique components of the DFT of a real-valued signal. If `fft_length` is not provided, it is computed from the size of the inner-most dimension of the input (`fft_length = 2 * (inner - 1)`). If the FFT length used to compute is odd, it should be provided since it cannot be inferred properly.

Along the axis IRFFT is computed on, if `fft_length / 2 + 1` is smaller than the corresponding dimension of the input, the dimension is cropped. If it is larger, the dimension is padded with zeros.

Usage

```
op_irfft(x, fft_length = NULL)
```

Arguments

x	List of the real and imaginary parts of the input tensor. Both tensors in the list should be of floating type.
fft_length	An integer representing the number of the fft length. If not specified, it is inferred from the length of the last axis of x. Defaults to NULL.

Value

A tensor containing the inverse real-valued Fast Fourier Transform along the last axis of x.

Examples

```
real <- op_array(c(0, 1, 2, 3, 4))
imag <- op_array(c(0, 1, 2, 3, 4))
op_irfft(c(real, imag))

#> tf.Tensor(
#> [ 2.         -2.0606601   0.5        -0.35355338  0.          0.06066012
#>  -0.5         0.35355338], shape=(8), dtype=float32)


all.equal(op_irfft(op_rfft(real, 5), 5), real)

#> [1] TRUE
```

See Also

- https://keras.io/api/ops/fft#irfft-function

Other math ops:
op_erf()
op_erfinv()
op_extract_sequences()
op_fft()
op_fft2()
op_in_top_k()
op_istft()
op_logsumexp()
op_qr()
op_rfft()
op_rsqrt()
op_segment_max()
op_segment_sum()
op_solve()

op_stft()
op_top_k()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()

op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_is_tensor()

op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()

op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()

op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

---

op_isclose *Return whether two tensors are element-wise almost equal.*

Description

Return whether two tensors are element-wise almost equal.

Usage

```
op_isclose(x1, x2)
```

Arguments

x1	First input tensor.
x2	Second input tensor.

Value

Output boolean tensor.

See Also

- https://keras.io/api/ops/numpy#isclose-function

Other numpy ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_bincount()
op_broadcast_to()
op_ceil()
op_clip()
op_concatenate()
op_conj()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_cumprod()
op_cumsum()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_einsum()
op_empty()
op_equal()
op_exp()
op_expand_dims()
op_expm1()

op_eye()
op_flip()
op_floor()
op_floor_divide()
op_full()
op_full_like()
op_get_item()
op_greater()
op_greater_equal()
op_hstack()
op_identity()
op_imag()
op_isfinite()
op_isinf()
op_isnan()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_matmul()
op_max()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moveaxis()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_not_equal()
op_ones()
op_ones_like()
op_outer()
op_pad()

op_power()
op_prod()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_repeat()
op_reshape()
op_roll()
op_round()
op_sign()
op_sin()
op_sinh()
op_size()
op_sort()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_subtract()
op_sum()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_var()
op_vdot()
op_vstack()
op_where()
op_zeros()
op_zeros_like()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()

op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()

op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()

op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()

op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()

op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

---

op_isfinite *Return whether a tensor is finite, element-wise.*

Description

Real values are finite when they are not NaN, not positive infinity, and not negative infinity. Complex values are finite when both their real and imaginary parts are finite.

Usage

```
op_isfinite(x)
```

Arguments

x	Input tensor.

Value

Output boolean tensor.

See Also

- https://keras.io/api/ops/numpy#isfinite-function

Other numpy ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()

op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_bincount()
op_broadcast_to()
op_ceil()
op_clip()
op_concatenate()
op_conj()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_cumprod()
op_cumsum()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_einsum()
op_empty()
op_equal()
op_exp()
op_expand_dims()
op_expm1()
op_eye()
op_flip()
op_floor()
op_floor_divide()
op_full()
op_full_like()
op_get_item()
op_greater()
op_greater_equal()
op_hstack()

op_identity()
op_imag()
op_isclose()
op_isinf()
op_isnan()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_matmul()
op_max()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moveaxis()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_not_equal()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_repeat()
op_reshape()
op_roll()
op_round()

op_sign()
op_sin()
op_sinh()
op_size()
op_sort()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_subtract()
op_sum()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_var()
op_vdot()
op_vstack()
op_where()
op_zeros()
op_zeros_like()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()

op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()

op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()

op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()

op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()

op_where()
op_while_loop()
op_zeros()
op_zeros_like()

---

op_isinf *Test element-wise for positive or negative infinity.*

Description

Test element-wise for positive or negative infinity.

Usage

```
op_isinf(x)
```

Arguments

x	Input tensor.

Value

Output boolean tensor.

See Also

- https://keras.io/api/ops/numpy#isinf-function

Other numpy ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()

op_bincount()
op_broadcast_to()
op_ceil()
op_clip()
op_concatenate()
op_conj()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_cumprod()
op_cumsum()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_einsum()
op_empty()
op_equal()
op_exp()
op_expand_dims()
op_expm1()
op_eye()
op_flip()
op_floor()
op_floor_divide()
op_full()
op_full_like()
op_get_item()
op_greater()
op_greater_equal()
op_hstack()
op_identity()
op_imag()
op_isclose()
op_isfinite()
op_isnan()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()

op_log2()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_matmul()
op_max()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moveaxis()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_not_equal()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_repeat()
op_reshape()
op_roll()
op_round()
op_sign()
op_sin()
op_sinh()
op_size()
op_sort()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()

op_subtract()
op_sum()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_var()
op_vdot()
op_vstack()
op_where()
op_zeros()
op_zeros_like()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()

op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()

op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()

op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()

op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

op_isnan *Test element-wise for NaN and return result as a boolean tensor.*

Description

Test element-wise for NaN and return result as a boolean tensor.

Usage

```
op_isnan(x)
```

Arguments

`x`	Input tensor.

Value

Output boolean tensor.

See Also

- `https://keras.io/api/ops/numpy#isnan-function`

Other numpy ops:
`op_abs()`
`op_add()`
`op_all()`
`op_any()`
`op_append()`
`op_arange()`
`op_arccos()`
`op_arccosh()`
`op_arcsin()`
`op_arcsinh()`
`op_arctan()`
`op_arctan2()`
`op_arctanh()`
`op_argmax()`
`op_argmin()`
`op_argsort()`
`op_array()`
`op_average()`
`op_bincount()`
`op_broadcast_to()`
`op_ceil()`
`op_clip()`
`op_concatenate()`
`op_conj()`
`op_copy()`
`op_correlate()`
`op_cos()`
`op_cosh()`
`op_count_nonzero()`

op_cross()
op_cumprod()
op_cumsum()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_einsum()
op_empty()
op_equal()
op_exp()
op_expand_dims()
op_expm1()
op_eye()
op_flip()
op_floor()
op_floor_divide()
op_full()
op_full_like()
op_get_item()
op_greater()
op_greater_equal()
op_hstack()
op_identity()
op_imag()
op_isclose()
op_isfinite()
op_isinf()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_matmul()
op_max()
op_maximum()
op_mean()

op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moveaxis()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_not_equal()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_repeat()
op_reshape()
op_roll()
op_round()
op_sign()
op_sin()
op_sinh()
op_size()
op_sort()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_subtract()
op_sum()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_trace()
op_transpose()

op_tri()
op_tril()
op_triu()
op_var()
op_vdot()
op_vstack()
op_where()
op_zeros()
op_zeros_like()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()

op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()

op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()

op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()

op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

---

op_istft — *Inverse Short-Time Fourier Transform along the last axis of the input.*

Description

To reconstruct an original waveform, the parameters should be the same in `stft`.

Usage

```
op_istft(
  x,
  sequence_length,
  sequence_stride,
  fft_length,
  length = NULL,
  window = "hann",
  center = TRUE
)
```

Arguments

`x`	Tuple of the real and imaginary parts of the input tensor. Both tensors in the list should be of floating type.
`sequence_length`	An integer representing the sequence length.
`sequence_stride`	An integer representing the sequence hop size.
`fft_length`	An integer representing the size of the FFT that produced `stft`.
`length`	An integer representing the output is clipped to exactly length. If not specified, no padding or clipping take place. Defaults to `NULL`.
`window`	A string, a tensor of the window or `NULL`. If `window` is a string, available values are `"hann"` and `"hamming"`. If `window` is a tensor, it will be used directly as the window and its length must be `sequence_length`. If `window` is `NULL`, no windowing is used. Defaults to `"hann"`.
`center`	Whether `x` was padded on both sides so that the t-th sequence is centered at time `t * sequence_stride`. Defaults to `TRUE`.

Value

A tensor containing the inverse Short-Time Fourier Transform along the last axis of `x`.

Examples

```
x <- op_convert_to_tensor(c(0, 1, 2, 3, 4))
op_istft(op_stft(x, 1, 1, 1), 1, 1, 1)

## tf.Tensor([], shape=(0), dtype=float32)


# array([0.0, 1.0, 2.0, 3.0, 4.0])
```

See Also

- https://keras.io/api/ops/fft#istft-function

Other math ops:
`op_erf()`
`op_erfinv()`
`op_extract_sequences()`
`op_fft()`
`op_fft2()`
`op_in_top_k()`
`op_irfft()`

op_logsumexp()
op_qr()
op_rfft()
op_rsqrt()
op_segment_max()
op_segment_sum()
op_solve()
op_stft()
op_top_k()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()

op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()

op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()

op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()

op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

---

op_is_tensor *Check whether the given object is a tensor.*

Description

Check whether the given object is a tensor.

Usage

```
op_is_tensor(x)
```

Arguments

x	A variable.

Value

TRUE if x is a tensor, otherwise FALSE.

Note

This checks for backend specific tensors so passing a TensorFlow tensor would return FALSE if your backend is PyTorch or JAX.

See Also

Other core ops:
op_cast()
op_cond()
op_convert_to_numpy()
op_convert_to_tensor()
op_custom_gradient()
op_fori_loop()
op_scatter()
op_scatter_update()
op_shape()
op_slice()
op_slice_update()
op_stop_gradient()
op_unstack()
op_vectorized_map()
op_while_loop()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()

op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()

op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()

op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()

op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

op_leaky_relu *Leaky version of a Rectified Linear Unit activation function.*

Description

It allows a small gradient when the unit is not active, it is defined as:
`f(x) = alpha * x for x < 0` or `f(x) = x for x >= 0`.

Usage

```
op_leaky_relu(x, negative_slope = 0.2)
```

Arguments

`x`	Input tensor.
`negative_slope`	Slope of the activation function at x < 0. Defaults to `0.2`.

Value

A tensor with the same shape as `x`.

Examples

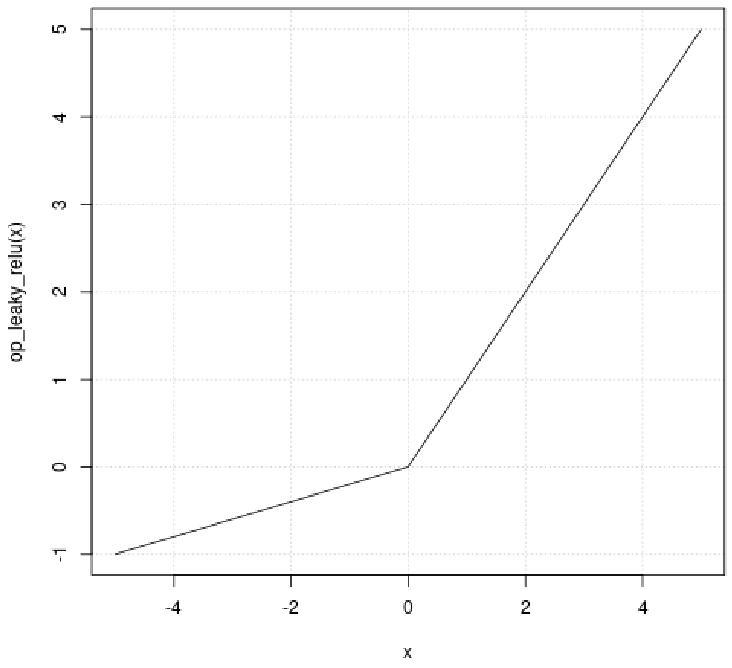
```
x <- op_array(c(-1., 0., 1.))
op_leaky_relu(x)


## tf.Tensor([-0.2  0.   1. ], shape=(3), dtype=float32)



# array([-0.2,  0. ,  1. ], shape=(3,), dtype=float64)


x <- seq(-5, 5, .1)
plot(x, op_leaky_relu(x),
     type = 'l', panel.first = grid())
```



See Also

- https://keras.io/api/ops/nn#leakyrelu-function

Other nn ops:
`op_average_pool()`
`op_batch_normalization()`
`op_binary_crossentropy()`
`op_categorical_crossentropy()`
`op_conv()`
`op_conv_transpose()`
`op_ctc_loss()`
`op_depthwise_conv()`
`op_elu()`
`op_gelu()`
`op_hard_sigmoid()`
`op_hard_silu()`
`op_log_sigmoid()`
`op_log_softmax()`
`op_max_pool()`
`op_moments()`
`op_multi_hot()`
`op_normalize()`
`op_one_hot()`

op_relu()
op_relu6()
op_selu()
op_separable_conv()
op_sigmoid()
op_silu()
op_softmax()
op_softplus()
op_softsign()
op_sparse_categorical_crossentropy()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()

op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()

op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()

op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()

op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

---

op_less *Return the truth value of* `x1 < x2` *element-wise.*

Description

Note that this function is automatically called when using the R operator < with a tensor.

```
(x <- op_arange(4))

## tf.Tensor([0. 1. 2. 3.], shape=(4), dtype=float64)


op_less(x, 2)
```

```
## tf.Tensor([ True  True False False], shape=(4), dtype=bool)
```

```
x < 2
```

```
## tf.Tensor([ True  True False False], shape=(4), dtype=bool)
```

Usage

```
op_less(x1, x2)
```

Arguments

x1	First input tensor.
x2	Second input tensor.

Value

Output tensor, element-wise comparison of x1 and x2.

See Also

- https://keras.io/api/ops/numpy#less-function

Other numpy ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_bincount()
op_broadcast_to()
op_ceil()
op_clip()
op_concatenate()
op_conj()

op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_cumprod()
op_cumsum()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_einsum()
op_empty()
op_equal()
op_exp()
op_expand_dims()
op_expm1()
op_eye()
op_flip()
op_floor()
op_floor_divide()
op_full()
op_full_like()
op_get_item()
op_greater()
op_greater_equal()
op_hstack()
op_identity()
op_imag()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()

op_logspace()
op_matmul()
op_max()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moveaxis()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_not_equal()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_repeat()
op_reshape()
op_roll()
op_round()
op_sign()
op_sin()
op_sinh()
op_size()
op_sort()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_subtract()
op_sum()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()

op_tanh()
op_tensordot()
op_tile()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_var()
op_vdot()
op_vstack()
op_where()
op_zeros()
op_zeros_like()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()

op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()

op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()

op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()

op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

---

op_less_equal *Return the truth value of* x1 <= x2 *element-wise.*

Description

Note that this function is automatically called when using the R operator <= with a tensor.

```
(x <- op_arange(4))
```

```
## tf.Tensor([0. 1. 2. 3.], shape=(4), dtype=float64)
```

```
op_less_equal(x, 2)

## tf.Tensor([ True  True  True False], shape=(4), dtype=bool)


x <= 2

## tf.Tensor([ True  True  True False], shape=(4), dtype=bool)
```

Usage

```
op_less_equal(x1, x2)
```

Arguments

x1	First input tensor.
x2	Second input tensor.

Value

Output tensor, element-wise comparison of x1 and x2.

See Also

- https://keras.io/api/ops/numpy#lessequal-function

Other numpy ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_bincount()
op_broadcast_to()
op_ceil()

op_clip()
op_concatenate()
op_conj()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_cumprod()
op_cumsum()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_einsum()
op_empty()
op_equal()
op_exp()
op_expand_dims()
op_expm1()
op_eye()
op_flip()
op_floor()
op_floor_divide()
op_full()
op_full_like()
op_get_item()
op_greater()
op_greater_equal()
op_hstack()
op_identity()
op_imag()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_less()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_logaddexp()
op_logical_and()

op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_matmul()
op_max()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moveaxis()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_not_equal()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_repeat()
op_reshape()
op_roll()
op_round()
op_sign()
op_sin()
op_sinh()
op_size()
op_sort()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_subtract()
op_sum()
op_swapaxes()

op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_var()
op_vdot()
op_vstack()
op_where()
op_zeros()
op_zeros_like()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()

op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()

op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()

op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()

op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

`op_linspace` *Return evenly spaced numbers over a specified interval.*

Description

Returns `num` evenly spaced samples, calculated over the interval `[start, stop]`.

The endpoint of the interval can optionally be excluded.

Usage

```
op_linspace(
  start,
  stop,
  num = 50L,
  endpoint = TRUE,
  retstep = FALSE,
  dtype = NULL,
  axis = 1L
)
```

Arguments

`start`	The starting value of the sequence.
`stop`	The end value of the sequence, unless `endpoint` is set to `FALSE`. In that case, the sequence consists of all but the last of `num + 1` evenly spaced samples, so that `stop` is excluded. Note that the step size changes when `endpoint` is `FALSE`.
`num`	Number of samples to generate. Defaults to `50`. Must be non-negative.
`endpoint`	If `TRUE`, `stop` is the last sample. Otherwise, it is not included. Defaults to`TRUE`.
`retstep`	If `TRUE`, return `(samples, step)`, where `step` is the spacing between samples.
`dtype`	The type of the output tensor.
`axis`	The axis in the result to store the samples. Relevant only if start or stop are array-like. Defaults to `1`, the first axis.

Value

A tensor of evenly spaced numbers. If `retstep` is `TRUE`, returns `(samples, step)`

Note

Torch backend does not support `axis` argument.

See Also

- `https://keras.io/api/ops/numpy#linspace-function`

Other numpy ops:
`op_abs()`
`op_add()`
`op_all()`
`op_any()`
`op_append()`
`op_arange()`
`op_arccos()`
`op_arccosh()`
`op_arcsin()`
`op_arcsinh()`
`op_arctan()`

op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_bincount()
op_broadcast_to()
op_ceil()
op_clip()
op_concatenate()
op_conj()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_cumprod()
op_cumsum()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_einsum()
op_empty()
op_equal()
op_exp()
op_expand_dims()
op_expm1()
op_eye()
op_flip()
op_floor()
op_floor_divide()
op_full()
op_full_like()
op_get_item()
op_greater()
op_greater_equal()
op_hstack()
op_identity()
op_imag()
op_isclose()
op_isfinite()

op_isinf()
op_isnan()
op_less()
op_less_equal()
op_log()
op_log10()
op_log1p()
op_log2()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_matmul()
op_max()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moveaxis()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_not_equal()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_repeat()
op_reshape()
op_roll()
op_round()
op_sign()
op_sin()
op_sinh()
op_size()

op_sort()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_subtract()
op_sum()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_var()
op_vdot()
op_vstack()
op_where()
op_zeros()
op_zeros_like()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()

op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()

op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()

op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()

op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

op_log	*Natural logarithm, element-wise.*

Description

Natural logarithm, element-wise.

Usage

```
op_log(x)
```

Arguments

x	Input tensor.

Value

Output tensor, element-wise natural logarithm of x.

See Also

- https://keras.io/api/ops/numpy#log-function

Other numpy ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_bincount()
op_broadcast_to()
op_ceil()
op_clip()

op_concatenate()
op_conj()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_cumprod()
op_cumsum()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_einsum()
op_empty()
op_equal()
op_exp()
op_expand_dims()
op_expm1()
op_eye()
op_flip()
op_floor()
op_floor_divide()
op_full()
op_full_like()
op_get_item()
op_greater()
op_greater_equal()
op_hstack()
op_identity()
op_imag()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_less()
op_less_equal()
op_linspace()
op_log10()
op_log1p()
op_log2()
op_logaddexp()
op_logical_and()
op_logical_not()

op_logical_or()
op_logical_xor()
op_logspace()
op_matmul()
op_max()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moveaxis()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_not_equal()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_repeat()
op_reshape()
op_roll()
op_round()
op_sign()
op_sin()
op_sinh()
op_size()
op_sort()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_subtract()
op_sum()
op_swapaxes()
op_take()

op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_var()
op_vdot()
op_vstack()
op_where()
op_zeros()
op_zeros_like()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()

op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()

op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()

op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()

op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

op_log10 *Return the base 10 logarithm of the input tensor, element-wise.*

Description

Return the base 10 logarithm of the input tensor, element-wise.

Usage

```
op_log10(x)
```

Arguments

x	Input tensor.

Value

Output tensor, element-wise base 10 logarithm of x.

See Also

- https://keras.io/api/ops/numpy#log10-function

Other numpy ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_bincount()
op_broadcast_to()
op_ceil()
op_clip()
op_concatenate()
op_conj()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_cumprod()
op_cumsum()
op_diag()

op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_einsum()
op_empty()
op_equal()
op_exp()
op_expand_dims()
op_expm1()
op_eye()
op_flip()
op_floor()
op_floor_divide()
op_full()
op_full_like()
op_get_item()
op_greater()
op_greater_equal()
op_hstack()
op_identity()
op_imag()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log1p()
op_log2()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_matmul()
op_max()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()

op_mod()
op_moveaxis()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_not_equal()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_repeat()
op_reshape()
op_roll()
op_round()
op_sign()
op_sin()
op_sinh()
op_size()
op_sort()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_subtract()
op_sum()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_var()

op_vdot()
op_vstack()
op_where()
op_zeros()
op_zeros_like()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()

op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()

op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()

op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()

op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

---

op_log1p *Returns the natural logarithm of one plus the x, element-wise.*

Description

Calculates `log(1 + x)`.

Usage

```
op_log1p(x)
```

Arguments

x	Input tensor.

Value

Output tensor, element-wise natural logarithm of `1 + x`.

See Also

- https://keras.io/api/ops/numpy#log1p-function

Other numpy ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_bincount()
op_broadcast_to()
op_ceil()
op_clip()
op_concatenate()
op_conj()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_cumprod()
op_cumsum()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_einsum()
op_empty()
op_equal()
op_exp()
op_expand_dims()
op_expm1()

op_eye()
op_flip()
op_floor()
op_floor_divide()
op_full()
op_full_like()
op_get_item()
op_greater()
op_greater_equal()
op_hstack()
op_identity()
op_imag()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log2()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_matmul()
op_max()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moveaxis()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_not_equal()
op_ones()
op_ones_like()
op_outer()
op_pad()

`op_power()`
`op_prod()`
`op_quantile()`
`op_ravel()`
`op_real()`
`op_reciprocal()`
`op_repeat()`
`op_reshape()`
`op_roll()`
`op_round()`
`op_sign()`
`op_sin()`
`op_sinh()`
`op_size()`
`op_sort()`
`op_split()`
`op_sqrt()`
`op_square()`
`op_squeeze()`
`op_stack()`
`op_std()`
`op_subtract()`
`op_sum()`
`op_swapaxes()`
`op_take()`
`op_take_along_axis()`
`op_tan()`
`op_tanh()`
`op_tensordot()`
`op_tile()`
`op_trace()`
`op_transpose()`
`op_tri()`
`op_tril()`
`op_triu()`
`op_var()`
`op_vdot()`
`op_vstack()`
`op_where()`
`op_zeros()`
`op_zeros_like()`

Other ops:
`op_abs()`
`op_add()`
`op_all()`
`op_any()`
`op_append()`

op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()

op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()

op_linspace()
op_log()
op_log10()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()

op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()

op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

---

op_log2 *Base-2 logarithm of* x*, element-wise.*

Description

Base-2 logarithm of x, element-wise.

Usage

```
op_log2(x)
```

Arguments

x	Input tensor.

Value

Output tensor, element-wise base-2 logarithm of x.

See Also

- https://keras.io/api/ops/numpy#log2-function

Other numpy ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()

op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_bincount()
op_broadcast_to()
op_ceil()
op_clip()
op_concatenate()
op_conj()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_cumprod()
op_cumsum()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_einsum()
op_empty()
op_equal()
op_exp()
op_expand_dims()
op_expm1()
op_eye()
op_flip()
op_floor()
op_floor_divide()
op_full()
op_full_like()
op_get_item()
op_greater()
op_greater_equal()
op_hstack()
op_identity()

op_imag()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_matmul()
op_max()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moveaxis()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_not_equal()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_repeat()
op_reshape()
op_roll()
op_round()
op_sign()

op_sin()
op_sinh()
op_size()
op_sort()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_subtract()
op_sum()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_var()
op_vdot()
op_vstack()
op_where()
op_zeros()
op_zeros_like()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()

op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()

op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()

op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()

op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()

op_while_loop()
op_zeros()
op_zeros_like()

---

op_logaddexp *Logarithm of the sum of exponentiations of the inputs.*

Description

Calculates `log(exp(x1) + exp(x2))`.

Usage

```
op_logaddexp(x1, x2)
```

Arguments

x1	Input tensor.
x2	Input tensor.

Value

Output tensor, element-wise logarithm of the sum of exponentiations of the inputs.

See Also

- https://keras.io/api/ops/numpy#logaddexp-function

Other numpy ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()

op_bincount()
op_broadcast_to()
op_ceil()
op_clip()
op_concatenate()
op_conj()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_cumprod()
op_cumsum()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_einsum()
op_empty()
op_equal()
op_exp()
op_expand_dims()
op_expm1()
op_eye()
op_flip()
op_floor()
op_floor_divide()
op_full()
op_full_like()
op_get_item()
op_greater()
op_greater_equal()
op_hstack()
op_identity()
op_imag()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()

op_log1p()
op_log2()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_matmul()
op_max()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moveaxis()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_not_equal()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_repeat()
op_reshape()
op_roll()
op_round()
op_sign()
op_sin()
op_sinh()
op_size()
op_sort()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()

op_subtract()
op_sum()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_var()
op_vdot()
op_vstack()
op_where()
op_zeros()
op_zeros_like()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()

op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()

op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()

op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()

op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

op_logical_and *Computes the element-wise logical AND of the given input tensors.*

Description

Zeros are treated as `FALSE` and non-zeros are treated as `TRUE`.

Usage

```
op_logical_and(x1, x2)
```

Arguments

`x1`	Input tensor.
`x2`	Input tensor.

Details

Note that this function is automatically called when using the R operator `&` with a tensor.

Value

Output tensor, element-wise logical AND of the inputs.

See Also

- `https://keras.io/api/ops/numpy#logicaland-function`

Other numpy ops:
`op_abs()`
`op_add()`
`op_all()`
`op_any()`
`op_append()`
`op_arange()`
`op_arccos()`
`op_arccosh()`
`op_arcsin()`
`op_arcsinh()`
`op_arctan()`
`op_arctan2()`
`op_arctanh()`
`op_argmax()`
`op_argmin()`
`op_argsort()`
`op_array()`
`op_average()`
`op_bincount()`
`op_broadcast_to()`
`op_ceil()`
`op_clip()`
`op_concatenate()`
`op_conj()`

op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_cumprod()
op_cumsum()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_einsum()
op_empty()
op_equal()
op_exp()
op_expand_dims()
op_expm1()
op_eye()
op_flip()
op_floor()
op_floor_divide()
op_full()
op_full_like()
op_get_item()
op_greater()
op_greater_equal()
op_hstack()
op_identity()
op_imag()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_logaddexp()
op_logical_not()
op_logical_or()
op_logical_xor()

op_logspace()
op_matmul()
op_max()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moveaxis()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_not_equal()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_repeat()
op_reshape()
op_roll()
op_round()
op_sign()
op_sin()
op_sinh()
op_size()
op_sort()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_subtract()
op_sum()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()

op_tanh()
op_tensordot()
op_tile()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_var()
op_vdot()
op_vstack()
op_where()
op_zeros()
op_zeros_like()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()

op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()

op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()

op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()

op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

---

op_logical_not *Computes the element-wise NOT of the given input tensor.*

Description

Zeros are treated as `FALSE` and non-zeros are treated as `TRUE`.

Note that this function is automatically called when using the R operator `!` with a tensor.

Usage

```
op_logical_not(x)
```

Arguments

x	Input tensor.

Value

Output tensor, element-wise logical NOT of the input.

See Also

- https://keras.io/api/ops/numpy#logicalnot-function

Other numpy ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_bincount()
op_broadcast_to()
op_ceil()
op_clip()
op_concatenate()
op_conj()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_cumprod()
op_cumsum()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()

op_divide_no_nan()
op_dot()
op_einsum()
op_empty()
op_equal()
op_exp()
op_expand_dims()
op_expm1()
op_eye()
op_flip()
op_floor()
op_floor_divide()
op_full()
op_full_like()
op_get_item()
op_greater()
op_greater_equal()
op_hstack()
op_identity()
op_imag()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_logaddexp()
op_logical_and()
op_logical_or()
op_logical_xor()
op_logspace()
op_matmul()
op_max()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moveaxis()
op_multiply()
op_nan_to_num()

op_ndim()
op_negative()
op_nonzero()
op_not_equal()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_repeat()
op_reshape()
op_roll()
op_round()
op_sign()
op_sin()
op_sinh()
op_size()
op_sort()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_subtract()
op_sum()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_var()
op_vdot()
op_vstack()
op_where()
op_zeros()

op_zeros_like()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()

op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()

op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()

op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()

op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

---

op_logical_or *Computes the element-wise logical OR of the given input tensors.*

Description

Zeros are treated as FALSE and non-zeros are treated as TRUE.

Note that this function is automatically called when using the R operator | with a tensor.

Usage

```
op_logical_or(x1, x2)
```

Arguments

x1	Input tensor.
x2	Input tensor.

Value

Output tensor, element-wise logical OR of the inputs.

See Also

- https://keras.io/api/ops/numpy#logicalor-function

Other numpy ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_bincount()
op_broadcast_to()
op_ceil()
op_clip()
op_concatenate()
op_conj()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_cumprod()
op_cumsum()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_einsum()
op_empty()
op_equal()
op_exp()
op_expand_dims()
op_expm1()

op_eye()
op_flip()
op_floor()
op_floor_divide()
op_full()
op_full_like()
op_get_item()
op_greater()
op_greater_equal()
op_hstack()
op_identity()
op_imag()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_xor()
op_logspace()
op_matmul()
op_max()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moveaxis()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_not_equal()
op_ones()
op_ones_like()
op_outer()
op_pad()

op_power()
op_prod()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_repeat()
op_reshape()
op_roll()
op_round()
op_sign()
op_sin()
op_sinh()
op_size()
op_sort()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_subtract()
op_sum()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_var()
op_vdot()
op_vstack()
op_where()
op_zeros()
op_zeros_like()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()

op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()

op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()

op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()

op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()

op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

---

op_logical_xor *Compute the truth value of* x1 XOR x2*, element-wise.*

Description

Compute the truth value of x1 XOR x2, element-wise.

Usage

```
op_logical_xor(x1, x2)
```

Arguments

x1	First input tensor.
x2	Second input tensor.

Value

Output boolean tensor.

See Also

Other numpy ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()

op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_bincount()
op_broadcast_to()
op_ceil()
op_clip()
op_concatenate()
op_conj()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_cumprod()
op_cumsum()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_einsum()
op_empty()
op_equal()
op_exp()
op_expand_dims()
op_expm1()
op_eye()
op_flip()
op_floor()
op_floor_divide()
op_full()
op_full_like()
op_get_item()
op_greater()
op_greater_equal()
op_hstack()
op_identity()

op_imag()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logspace()
op_matmul()
op_max()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moveaxis()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_not_equal()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_repeat()
op_reshape()
op_roll()
op_round()
op_sign()

op_sin()
op_sinh()
op_size()
op_sort()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_subtract()
op_sum()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_var()
op_vdot()
op_vstack()
op_where()
op_zeros()
op_zeros_like()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()

op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()

op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()

op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()

op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()

op_while_loop()
op_zeros()
op_zeros_like()

op_logspace *Returns numbers spaced evenly on a log scale.*

Description

In linear space, the sequence starts at base ** start and ends with base ** stop (see endpoint below).

Usage

```
op_logspace(
  start,
  stop,
  num = 50L,
  endpoint = TRUE,
  base = 10L,
  dtype = NULL,
  axis = 1L
)
```

Arguments

start	The starting value of the sequence.
stop	The final value of the sequence, unless endpoint is FALSE. In that case, num + 1 values are spaced over the interval in log-space, of which all but the last (a sequence of length num) are returned.
num	Number of samples to generate. Defaults to 50.
endpoint	If TRUE, stop is the last sample. Otherwise, it is not included. Defaults toTRUE.
base	The base of the log space. Defaults to 10.
dtype	The type of the output tensor.
axis	The axis in the result to store the samples. Relevant only if start or stop are array-like.

Value

A tensor of evenly spaced samples on a log scale.

Note

Torch backend does not support axis argument.

See Also

- https://keras.io/api/ops/numpy#logspace-function

Other numpy ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_bincount()
op_broadcast_to()
op_ceil()
op_clip()
op_concatenate()
op_conj()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_cumprod()
op_cumsum()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_einsum()
op_empty()
op_equal()
op_exp()
op_expand_dims()
op_expm1()

op_eye()
op_flip()
op_floor()
op_floor_divide()
op_full()
op_full_like()
op_get_item()
op_greater()
op_greater_equal()
op_hstack()
op_identity()
op_imag()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_matmul()
op_max()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moveaxis()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_not_equal()
op_ones()
op_ones_like()
op_outer()
op_pad()

op_power()
op_prod()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_repeat()
op_reshape()
op_roll()
op_round()
op_sign()
op_sin()
op_sinh()
op_size()
op_sort()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_subtract()
op_sum()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_var()
op_vdot()
op_vstack()
op_where()
op_zeros()
op_zeros_like()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()

op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()

op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()

op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()

op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()

op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

op_logsumexp *Computes the logarithm of sum of exponentials of elements in a tensor.*

Description

Computes the logarithm of sum of exponentials of elements in a tensor.

Usage

```
op_logsumexp(x, axis = NULL, keepdims = FALSE)
```

Arguments

x	Input tensor.
axis	An integer or a list of integers specifying the axis/axes along which to compute the sum. If NULL, the sum is computed over all elements. Defaults toNULL.
keepdims	A boolean indicating whether to keep the dimensions of the input tensor when computing the sum. Defaults toFALSE.

Value

A tensor containing the logarithm of the sum of exponentials of elements in x.

Examples

```
x <- op_convert_to_tensor(c(1, 2, 3))
op_logsumexp(x)

## tf.Tensor(3.407606, shape=(), dtype=float32)
```

See Also

- https://keras.io/api/ops/core#logsumexp-function

Other math ops:
op_erf()
op_erfinv()
op_extract_sequences()
op_fft()
op_fft2()
op_in_top_k()
op_irfft()
op_istft()
op_qr()
op_rfft()
op_rsqrt()
op_segment_max()
op_segment_sum()
op_solve()
op_stft()
op_top_k()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()

op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()

op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()

op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()

op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

---

op_log_sigmoid *Logarithm of the sigmoid activation function.*

Description

It is defined as `f(x) = log(1 / (1 + exp(-x)))`.

Usage

```
op_log_sigmoid(x)
```

Arguments

x	Input tensor.

Value

A tensor with the same shape as x.

Examples

```
x <- op_convert_to_tensor(c(-0.541391, 0.0, 0.50, 5.0))
op_log_sigmoid(x)

## tf.Tensor([-1.0000418  -0.6931472  -0.474077   -0.00671535], shape=(4), dtype=float32)
```

See Also

- https://keras.io/api/ops/nn#logsigmoid-function

Other nn ops:
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_categorical_crossentropy()
op_conv()
op_conv_transpose()
op_ctc_loss()
op_depthwise_conv()
op_elu()
op_gelu()
op_hard_sigmoid()
op_hard_silu()
op_leaky_relu()
op_log_softmax()
op_max_pool()
op_moments()
op_multi_hot()
op_normalize()
op_one_hot()
op_relu()
op_relu6()
op_selu()

op_separable_conv()
op_sigmoid()
op_silu()
op_softmax()
op_softplus()
op_softsign()
op_sparse_categorical_crossentropy()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()

op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()

op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()

op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()

op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

---

op_log_softmax *Log-softmax activation function.*

Description

It is defined as: `f(x) = x - max(x) - log(sum(exp(x - max(x))))`

Usage

```
op_log_softmax(x, axis = -1L)
```

Arguments

x	Input tensor.
axis	Integer, axis along which the log-softmax is applied. Defaults to -1.

Value

A tensor with the same shape as x.

Examples

```
x <- op_array(c(-1., 0., 1.))
op_log_softmax(x)

## tf.Tensor([-2.407606   -1.4076059  -0.40760595], shape=(3), dtype=float32)
```

See Also

- https://keras.io/api/ops/nn#logsoftmax-function

Other nn ops:
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_categorical_crossentropy()
op_conv()
op_conv_transpose()
op_ctc_loss()
op_depthwise_conv()
op_elu()
op_gelu()
op_hard_sigmoid()
op_hard_silu()
op_leaky_relu()
op_log_sigmoid()
op_max_pool()
op_moments()
op_multi_hot()
op_normalize()
op_one_hot()
op_relu()
op_relu6()
op_selu()
op_separable_conv()
op_sigmoid()
op_silu()
op_softmax()
op_softplus()
op_softsign()
op_sparse_categorical_crossentropy()

Other ops:
op_abs()
op_add()

op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()

op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()

op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()

op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()

op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

op_lu_factor *Computes the lower-upper decomposition of a square matrix.*

Description

Computes the lower-upper decomposition of a square matrix.

Usage

```
op_lu_factor(x)
```

Arguments

x	A tensor of shape `(..., M, M)`.

Value

A tuple of two tensors: a tensor of shape `(..., M, M)` containing the lower and upper triangular matrices and a tensor of shape `(..., M)` containing the pivots.

See Also

Other linear algebra ops:
op_cholesky()
op_det()
op_eig()
op_inv()
op_norm()
op_solve_triangular()

op_svd()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()

op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()

op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()

op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()

op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

op_matmul *Matrix product of two tensors.*

Description

- If both tensors are 1-dimensional, the dot product (scalar) is returned.
- If either tensor is N-D, N > 2, it is treated as a stack of matrices residing in the last two indexes and broadcast accordingly.
- If the first tensor is 1-D, it is promoted to a matrix by prepending a 1 to its dimensions. After matrix multiplication the prepended 1 is removed.
- If the second tensor is 1-D, it is promoted to a matrix by appending a 1 to its dimensions. After matrix multiplication the appended 1 is removed.

Usage

```
op_matmul(x1, x2)
```

Arguments

x1	First tensor.
x2	Second tensor.

Value

Output tensor, matrix product of the inputs.

See Also

- https://keras.io/api/ops/numpy#matmul-function

Other numpy ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_bincount()
op_broadcast_to()
op_ceil()
op_clip()
op_concatenate()
op_conj()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_cumprod()
op_cumsum()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_einsum()
op_empty()

op_equal()
op_exp()
op_expand_dims()
op_expm1()
op_eye()
op_flip()
op_floor()
op_floor_divide()
op_full()
op_full_like()
op_get_item()
op_greater()
op_greater_equal()
op_hstack()
op_identity()
op_imag()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_max()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moveaxis()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_not_equal()

op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_repeat()
op_reshape()
op_roll()
op_round()
op_sign()
op_sin()
op_sinh()
op_size()
op_sort()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_subtract()
op_sum()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_var()
op_vdot()
op_vstack()
op_where()
op_zeros()
op_zeros_like()

Other ops:
op_abs()

op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()

op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()

op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()

op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()

op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

---

op_max *Return the maximum of a tensor or maximum along an axis.*

Description

Return the maximum of a tensor or maximum along an axis.

Usage

```
op_max(x, axis = NULL, keepdims = FALSE, initial = NULL)
```

Arguments

x	Input tensor.
axis	Axis or axes along which to operate. By default, flattened input is used.
keepdims	If this is set to TRUE, the axes which are reduced are left in the result as dimensions with size one. Defaults toFALSE.
initial	The minimum value of an output element. Defaults toNULL.

Value

Maximum of x.

Examples

```
(x <- op_convert_to_tensor(rbind(c(1, 3, 5), c(1, 5, 2))))

## tf.Tensor(
## [[1. 3. 5.]
##  [1. 5. 2.]], shape=(2, 3), dtype=float64)


op_max(x)

## tf.Tensor(5.0, shape=(), dtype=float64)


op_max(x, axis = 1)

## tf.Tensor([1. 5. 5.], shape=(3), dtype=float64)


op_max(x, axis = 1, keepdims = TRUE)

## tf.Tensor([[1. 5. 5.]], shape=(1, 3), dtype=float64)
```

See Also

- https://keras.io/api/ops/numpy#max-function

Other numpy ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_bincount()
op_broadcast_to()

op_ceil()
op_clip()
op_concatenate()
op_conj()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_cumprod()
op_cumsum()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_einsum()
op_empty()
op_equal()
op_exp()
op_expand_dims()
op_expm1()
op_eye()
op_flip()
op_floor()
op_floor_divide()
op_full()
op_full_like()
op_get_item()
op_greater()
op_greater_equal()
op_hstack()
op_identity()
op_imag()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()

op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_matmul()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moveaxis()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_not_equal()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_repeat()
op_reshape()
op_roll()
op_round()
op_sign()
op_sin()
op_sinh()
op_size()
op_sort()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_subtract()
op_sum()

op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_var()
op_vdot()
op_vstack()
op_where()
op_zeros()
op_zeros_like()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()

op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()

op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()

op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()

op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

---

op_maximum *Element-wise maximum of* x1 *and* x2.

Description

Element-wise maximum of x1 and x2.

Usage

```
op_maximum(x1, x2)

op_pmax(x1, x2)
```

Arguments

x1	First tensor.
x2	Second tensor.

Value

Output tensor, element-wise maximum of x1 and x2.

See Also

- https://keras.io/api/ops/numpy#maximum-function

Other numpy ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_bincount()
op_broadcast_to()
op_ceil()
op_clip()
op_concatenate()
op_conj()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()

op_cumprod()
op_cumsum()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_einsum()
op_empty()
op_equal()
op_exp()
op_expand_dims()
op_expm1()
op_eye()
op_flip()
op_floor()
op_floor_divide()
op_full()
op_full_like()
op_get_item()
op_greater()
op_greater_equal()
op_hstack()
op_identity()
op_imag()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_matmul()
op_max()
op_mean()
op_median()

op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moveaxis()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_not_equal()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_repeat()
op_reshape()
op_roll()
op_round()
op_sign()
op_sin()
op_sinh()
op_size()
op_sort()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_subtract()
op_sum()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_trace()
op_transpose()
op_tri()

op_tril()
op_triu()
op_var()
op_vdot()
op_vstack()
op_where()
op_zeros()
op_zeros_like()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()

op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()

op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()

op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()

op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

---

op_max_pool *Max pooling operation.*

Description

Max pooling operation.

Usage

```
op_max_pool(
  inputs,
  pool_size,
  strides = NULL,
  padding = "valid",
  data_format = NULL
)
```

Arguments

`inputs`	Tensor of rank N+2. `inputs` has shape `(batch_size,) + inputs_spatial_shape + (num_channels,)` if `data_format = "channels_last"`, or `(batch_size, num_channels) + inputs_spatial_shape` if `data_format = "channels_first"`. Pooling happens over the spatial dimensions only.
`pool_size`	int or tuple/list of integers of size `len(inputs_spatial_shape)`, specifying the size of the pooling window for each spatial dimension of the input tensor. If `pool_size` is int, then every spatial dimension shares the same `pool_size`.
`strides`	int or tuple/list of integers of size `len(inputs_spatial_shape)`. The stride of the sliding window for each spatial dimension of the input tensor. If `strides` is int, then every spatial dimension shares the same `strides`.
`padding`	string, either `"valid"` or `"same"`. `"valid"` means no padding is applied, and `"same"` results in padding evenly to the left/right or up/down of the input such that output has the same height/width dimension as the input when `strides = 1`.
`data_format`	A string, either `"channels_last"` or `"channels_first"`. `data_format` determines the ordering of the dimensions in the inputs. If `data_format = "channels_last"`, `inputs` is of shape `(batch_size, ..., channels)` while if `data_format = "channels_first"`, `inputs` is of shape `(batch_size, channels, ...)`.

Value

A tensor of rank N+2, the result of the max pooling operation.

See Also

- `https://keras.io/api/ops/nn#maxpool-function`

Other nn ops:
`op_average_pool()`
`op_batch_normalization()`
`op_binary_crossentropy()`
`op_categorical_crossentropy()`
`op_conv()`
`op_conv_transpose()`
`op_ctc_loss()`
`op_depthwise_conv()`
`op_elu()`
`op_gelu()`
`op_hard_sigmoid()`
`op_hard_silu()`
`op_leaky_relu()`
`op_log_sigmoid()`
`op_log_softmax()`
`op_moments()`
`op_multi_hot()`
`op_normalize()`
`op_one_hot()`

op_relu()
op_relu6()
op_selu()
op_separable_conv()
op_sigmoid()
op_silu()
op_softmax()
op_softplus()
op_softsign()
op_sparse_categorical_crossentropy()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()

op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()

op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()

op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()

op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

---

op_mean *Compute the arithmetic mean along the specified axes.*

Description

Compute the arithmetic mean along the specified axes.

Usage

```
op_mean(x, axis = NULL, keepdims = FALSE)
```

Arguments

x	Input tensor.

axis	Axis or axes along which the means are computed. The default is to compute the mean of the flattened tensor.
keepdims	If this is set to TRUE, the axes which are reduced are left in the result as dimensions with size one.

Value

Output tensor containing the mean values.

See Also

- https://keras.io/api/ops/numpy#mean-function

Other numpy ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_bincount()
op_broadcast_to()
op_ceil()
op_clip()
op_concatenate()
op_conj()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_cumprod()
op_cumsum()
op_diag()
op_diagonal()
op_diff()

op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_einsum()
op_empty()
op_equal()
op_exp()
op_expand_dims()
op_expm1()
op_eye()
op_flip()
op_floor()
op_floor_divide()
op_full()
op_full_like()
op_get_item()
op_greater()
op_greater_equal()
op_hstack()
op_identity()
op_imag()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_matmul()
op_max()
op_maximum()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moveaxis()

op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_not_equal()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_repeat()
op_reshape()
op_roll()
op_round()
op_sign()
op_sin()
op_sinh()
op_size()
op_sort()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_subtract()
op_sum()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_var()
op_vdot()
op_vstack()

op_where()
op_zeros()
op_zeros_like()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()

op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()

op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()

op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()

op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

---

op_median *Compute the median along the specified axis.*

Description

Compute the median along the specified axis.

Usage

```
op_median(x, axis = NULL, keepdims = FALSE)
```

Arguments

x	Input tensor.
axis	Axis or axes along which the medians are computed. Defaults to axis = NULL which is to compute the median(s) along a flattened version of the array.
keepdims	If this is set to TRUE, the axes which are reduce are left in the result as dimensions with size one.

Value

The output tensor.

See Also

Other numpy ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_bincount()
op_broadcast_to()
op_ceil()
op_clip()
op_concatenate()
op_conj()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_cumprod()
op_cumsum()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_einsum()
op_empty()
op_equal()
op_exp()
op_expand_dims()
op_expm1()
op_eye()

op_flip()
op_floor()
op_floor_divide()
op_full()
op_full_like()
op_get_item()
op_greater()
op_greater_equal()
op_hstack()
op_identity()
op_imag()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_matmul()
op_max()
op_maximum()
op_mean()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moveaxis()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_not_equal()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()

op_prod()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_repeat()
op_reshape()
op_roll()
op_round()
op_sign()
op_sin()
op_sinh()
op_size()
op_sort()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_subtract()
op_sum()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_var()
op_vdot()
op_vstack()
op_where()
op_zeros()
op_zeros_like()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()

op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()

op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()

op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()

op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()

op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

---

op_meshgrid *Creates grids of coordinates from coordinate vectors.*

Description

Given N 1-D tensors T0, T1, ..., TN-1 as inputs with corresponding lengths S0, S1, ..., SN-1, this creates an N N-dimensional tensors G0, G1, ..., GN-1 each with shape (S0, ..., SN-1) where the output Gi is constructed by expanding Ti to the result shape.

Usage

```
op_meshgrid(..., indexing = "xy")
```

Arguments

...	1-D tensors representing the coordinates of a grid.
indexing	"xy" or "ij". "xy" is cartesian; "ij" is matrix indexing of output. Defaults to "xy".

Value

Sequence of N tensors.

Examples

```
x <- op_array(c(1, 2, 3), "int32")
y <- op_array(c(4, 5, 6), "int32")

c(grid_x, grid_y) %<-% op_meshgrid(x, y, indexing = "ij")
grid_x
```

```
## tf.Tensor(
## [[1 1 1]
##  [2 2 2]
##  [3 3 3]], shape=(3, 3), dtype=int32)


# array([[1, 1, 1],
#        [2, 2, 2],
#        [3, 3, 3]])
grid_y

## tf.Tensor(
## [[4 5 6]
##  [4 5 6]
##  [4 5 6]], shape=(3, 3), dtype=int32)


# array([[4, 5, 6],
#        [4, 5, 6],
#        [4, 5, 6]])
```

See Also

- https://keras.io/api/ops/numpy#meshgrid-function

Other numpy ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_bincount()
op_broadcast_to()
op_ceil()
op_clip()
op_concatenate()

op_conj()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_cumprod()
op_cumsum()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_einsum()
op_empty()
op_equal()
op_exp()
op_expand_dims()
op_expm1()
op_eye()
op_flip()
op_floor()
op_floor_divide()
op_full()
op_full_like()
op_get_item()
op_greater()
op_greater_equal()
op_hstack()
op_identity()
op_imag()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_logaddexp()
op_logical_and()
op_logical_not()

op_logical_or()
op_logical_xor()
op_logspace()
op_matmul()
op_max()
op_maximum()
op_mean()
op_median()
op_min()
op_minimum()
op_mod()
op_moveaxis()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_not_equal()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_repeat()
op_reshape()
op_roll()
op_round()
op_sign()
op_sin()
op_sinh()
op_size()
op_sort()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_subtract()
op_sum()
op_swapaxes()
op_take()
op_take_along_axis()

op_tan()
op_tanh()
op_tensordot()
op_tile()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_var()
op_vdot()
op_vstack()
op_where()
op_zeros()
op_zeros_like()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()

op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()

op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()

op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()

op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

---

op_min *Return the minimum of a tensor or minimum along an axis.*

Description

Return the minimum of a tensor or minimum along an axis.

Usage

```
op_min(x, axis = NULL, keepdims = FALSE, initial = NULL)
```

Arguments

x	Input tensor.
axis	Axis or axes along which to operate. By default, flattened input is used.
keepdims	If this is set to TRUE, the axes which are reduced are left in the result as dimensions with size one. Defaults toFALSE.
initial	The maximum value of an output element. Defaults toNULL.

Value

Minimum of x.

Examples

```
(x <- op_convert_to_tensor(rbind(c(1, 3, 5), c(1, 5, 2))))

## tf.Tensor(
## [[1. 3. 5.]
##  [1. 5. 2.]], shape=(2, 3), dtype=float64)


op_min(x)

## tf.Tensor(1.0, shape=(), dtype=float64)


op_min(x, axis = 1)

## tf.Tensor([1. 3. 2.], shape=(3), dtype=float64)


op_min(x, axis = 1, keepdims = TRUE)

## tf.Tensor([[1. 3. 2.]], shape=(1, 3), dtype=float64)
```

See Also

- https://keras.io/api/ops/numpy#min-function

Other numpy ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()

op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_bincount()
op_broadcast_to()
op_ceil()
op_clip()
op_concatenate()
op_conj()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_cumprod()
op_cumsum()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_einsum()
op_empty()
op_equal()
op_exp()
op_expand_dims()
op_expm1()
op_eye()
op_flip()
op_floor()
op_floor_divide()
op_full()
op_full_like()
op_get_item()
op_greater()
op_greater_equal()
op_hstack()
op_identity()

op_imag()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_matmul()
op_max()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_minimum()
op_mod()
op_moveaxis()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_not_equal()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_repeat()
op_reshape()
op_roll()
op_round()
op_sign()

op_sin()
op_sinh()
op_size()
op_sort()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_subtract()
op_sum()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_var()
op_vdot()
op_vstack()
op_where()
op_zeros()
op_zeros_like()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()

op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()

op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()

op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()

op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()

op_while_loop()
op_zeros()
op_zeros_like()

---

op_minimum *Element-wise minimum of* x1 *and* x2.

Description

Element-wise minimum of x1 and x2.

Usage

```
op_minimum(x1, x2)

op_pmin(x1, x2)
```

Arguments

x1	First tensor.
x2	Second tensor.

Value

Output tensor, element-wise minimum of x1 and x2.

See Also

- https://keras.io/api/ops/numpy#minimum-function

Other numpy ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()

op_array()
op_average()
op_bincount()
op_broadcast_to()
op_ceil()
op_clip()
op_concatenate()
op_conj()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_cumprod()
op_cumsum()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_einsum()
op_empty()
op_equal()
op_exp()
op_expand_dims()
op_expm1()
op_eye()
op_flip()
op_floor()
op_floor_divide()
op_full()
op_full_like()
op_get_item()
op_greater()
op_greater_equal()
op_hstack()
op_identity()
op_imag()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_less()
op_less_equal()
op_linspace()

op_log()
op_log10()
op_log1p()
op_log2()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_matmul()
op_max()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_mod()
op_moveaxis()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_not_equal()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_repeat()
op_reshape()
op_roll()
op_round()
op_sign()
op_sin()
op_sinh()
op_size()
op_sort()
op_split()
op_sqrt()
op_square()
op_squeeze()

op_stack()
op_std()
op_subtract()
op_sum()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_var()
op_vdot()
op_vstack()
op_where()
op_zeros()
op_zeros_like()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()

op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()

op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()

op_mean()
op_median()
op_meshgrid()
op_min()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()

op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

op_mod *Returns the element-wise remainder of division.*

Description

Note that this function is automatically called when using the R operator %% with a tensor.

```
(x <- op_arange(10))

## tf.Tensor([0. 1. 2. 3. 4. 5. 6. 7. 8. 9.], shape=(10), dtype=float64)


op_mod(x, 3)

## tf.Tensor([0. 1. 2. 0. 1. 2. 0. 1. 2. 0.], shape=(10), dtype=float64)


x %% 3

## tf.Tensor([0. 1. 2. 0. 1. 2. 0. 1. 2. 0.], shape=(10), dtype=float64)
```

Usage

```
op_mod(x1, x2)
```

Arguments

x1	First tensor.
x2	Second tensor.

Value

Output tensor, element-wise remainder of division.

See Also

- https://keras.io/api/ops/numpy#mod-function

Other numpy ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()

op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_bincount()
op_broadcast_to()
op_ceil()
op_clip()
op_concatenate()
op_conj()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_cumprod()
op_cumsum()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_einsum()
op_empty()
op_equal()
op_exp()
op_expand_dims()
op_expm1()
op_eye()
op_flip()
op_floor()
op_floor_divide()
op_full()
op_full_like()
op_get_item()
op_greater()
op_greater_equal()
op_hstack()
op_identity()
op_imag()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()

op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_matmul()
op_max()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_moveaxis()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_not_equal()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_repeat()
op_reshape()
op_roll()
op_round()
op_sign()
op_sin()
op_sinh()
op_size()
op_sort()
op_split()

op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_subtract()
op_sum()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_var()
op_vdot()
op_vstack()
op_where()
op_zeros()
op_zeros_like()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()

op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()

op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()

op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()

op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

op_moments *Calculates the mean and variance of* x.

Description

The mean and variance are calculated by aggregating the contents of x across axes. If x is 1-D and axes = c(1) this is just the mean and variance of a vector.

Usage

```
op_moments(x, axes, keepdims = FALSE, synchronized = FALSE)
```

Arguments

x	Input tensor.
axes	A list of axes which to compute mean and variance.
keepdims	If this is set to TRUE, the axes which are reduced are left in the result as dimensions with size one.
synchronized	Only applicable with the TensorFlow backend. If TRUE, synchronizes the global batch statistics (mean and variance) across all devices at each training step in a distributed training strategy. If FALSE, each replica uses its own local batch statistics.

Value

A list containing two tensors - mean and variance.

Examples

```
x <- op_convert_to_tensor(c(0, 1, 2, 3, 100), dtype = "float32")
op_moments(x, axes = c(1))

## [[1]]
## tf.Tensor(21.2, shape=(), dtype=float32)
##
## [[2]]
## tf.Tensor(1553.3601, shape=(), dtype=float32)
```

See Also

Other nn ops:
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_categorical_crossentropy()

op_conv()
op_conv_transpose()
op_ctc_loss()
op_depthwise_conv()
op_elu()
op_gelu()
op_hard_sigmoid()
op_hard_silu()
op_leaky_relu()
op_log_sigmoid()
op_log_softmax()
op_max_pool()
op_multi_hot()
op_normalize()
op_one_hot()
op_relu()
op_relu6()
op_selu()
op_separable_conv()
op_sigmoid()
op_silu()
op_softmax()
op_softplus()
op_softsign()
op_sparse_categorical_crossentropy()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()

op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()

op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()

op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()

op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

op_moveaxis *Move axes of a tensor to new positions.*

Description

Other axes remain in their original order.

Usage

```
op_moveaxis(x, source, destination)
```

Arguments

x	Tensor whose axes should be reordered.
source	Original positions of the axes to move. These must be unique.
destination	Destinations positions for each of the original axes. These must also be unique.

Value

Tensor with moved axes.

See Also

- https://keras.io/api/ops/numpy#moveaxis-function

Other numpy ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_bincount()
op_broadcast_to()
op_ceil()

op_clip()
op_concatenate()
op_conj()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_cumprod()
op_cumsum()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_einsum()
op_empty()
op_equal()
op_exp()
op_expand_dims()
op_expm1()
op_eye()
op_flip()
op_floor()
op_floor_divide()
op_full()
op_full_like()
op_get_item()
op_greater()
op_greater_equal()
op_hstack()
op_identity()
op_imag()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_logaddexp()

op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_matmul()
op_max()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_not_equal()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_repeat()
op_reshape()
op_roll()
op_round()
op_sign()
op_sin()
op_sinh()
op_size()
op_sort()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_subtract()
op_sum()
op_swapaxes()

op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_var()
op_vdot()
op_vstack()
op_where()
op_zeros()
op_zeros_like()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()

op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()

op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()

op_mod()
op_moments()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()

op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

`op_multiply` *Multiply arguments element-wise.*

Description

Note that this function is automatically called when using the R operator `*` with a tensor.

```
(x <- op_arange(4))

## tf.Tensor([0. 1. 2. 3.], shape=(4), dtype=float64)


op_multiply(x, x)

## tf.Tensor([0. 1. 4. 9.], shape=(4), dtype=float64)


x * x

## tf.Tensor([0. 1. 4. 9.], shape=(4), dtype=float64)
```

Usage

```
op_multiply(x1, x2)
```

Arguments

x1	First input tensor.
x2	Second input tensor.

Value

Output tensor, element-wise product of x1 and x2.

See Also

- https://keras.io/api/ops/numpy#multiply-function

Other numpy ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()

op_array()
op_average()
op_bincount()
op_broadcast_to()
op_ceil()
op_clip()
op_concatenate()
op_conj()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_cumprod()
op_cumsum()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_einsum()
op_empty()
op_equal()
op_exp()
op_expand_dims()
op_expm1()
op_eye()
op_flip()
op_floor()
op_floor_divide()
op_full()
op_full_like()
op_get_item()
op_greater()
op_greater_equal()
op_hstack()
op_identity()
op_imag()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_less()
op_less_equal()
op_linspace()

op_log()
op_log10()
op_log1p()
op_log2()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_matmul()
op_max()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moveaxis()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_not_equal()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_repeat()
op_reshape()
op_roll()
op_round()
op_sign()
op_sin()
op_sinh()
op_size()
op_sort()
op_split()
op_sqrt()
op_square()
op_squeeze()

op_stack()
op_std()
op_subtract()
op_sum()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_var()
op_vdot()
op_vstack()
op_where()
op_zeros()
op_zeros_like()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()

op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()

op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()

op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()

op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

op_multi_hot *Encodes integer labels as multi-hot vectors.*

Description

This function encodes integer labels as multi-hot vectors, where each label is mapped to a binary value in the resulting vector.

Usage

```
op_multi_hot(
  inputs,
  num_classes,
  axis = -1L,
  dtype = NULL,
  sparse = FALSE,
  ...
)
```

Arguments

`inputs`	Tensor of integer labels to be converted to multi-hot vectors.
`num_classes`	Integer, the total number of unique classes.
`axis`	(optional) Axis along which the multi-hot encoding should be added. Defaults to `-1`, which corresponds to the last dimension.
`dtype`	(optional) The data type of the resulting tensor. Default is backend's float type.
`sparse`	Whether to return a sparse tensor; for backends that support sparse tensors.
`...`	For forward/backwards compatability

Value

Tensor: The multi-hot encoded tensor.

Examples

```
data <- op_convert_to_tensor(c(0, 4))
op_multi_hot(data, num_classes = 5)

## tf.Tensor([1. 0. 0. 0. 1.], shape=(5), dtype=float32)
```

See Also

Other nn ops:
`op_average_pool()`
`op_batch_normalization()`
`op_binary_crossentropy()`
`op_categorical_crossentropy()`
`op_conv()`
`op_conv_transpose()`
`op_ctc_loss()`

op_depthwise_conv()
op_elu()
op_gelu()
op_hard_sigmoid()
op_hard_silu()
op_leaky_relu()
op_log_sigmoid()
op_log_softmax()
op_max_pool()
op_moments()
op_normalize()
op_one_hot()
op_relu()
op_relu6()
op_selu()
op_separable_conv()
op_sigmoid()
op_silu()
op_softmax()
op_softplus()
op_softsign()
op_sparse_categorical_crossentropy()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()

op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()

op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()

op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()

op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

---

op_nan_to_num *Replace NaN with zero and infinity with large finite numbers.*

Description

Replace NaN with zero and infinity with large finite numbers.

Usage

```
op_nan_to_num(x)
```

Arguments

x	Input data.

Value

x, with non-finite values replaced.

See Also

- https://keras.io/api/ops/numpy#nantonum-function

Other numpy ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_bincount()
op_broadcast_to()
op_ceil()
op_clip()
op_concatenate()
op_conj()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()

op_cross()
op_cumprod()
op_cumsum()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_einsum()
op_empty()
op_equal()
op_exp()
op_expand_dims()
op_expm1()
op_eye()
op_flip()
op_floor()
op_floor_divide()
op_full()
op_full_like()
op_get_item()
op_greater()
op_greater_equal()
op_hstack()
op_identity()
op_imag()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_matmul()
op_max()
op_maximum()

op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moveaxis()
op_multiply()
op_ndim()
op_negative()
op_nonzero()
op_not_equal()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_repeat()
op_reshape()
op_roll()
op_round()
op_sign()
op_sin()
op_sinh()
op_size()
op_sort()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_subtract()
op_sum()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_trace()
op_transpose()

op_tri()
op_tril()
op_triu()
op_var()
op_vdot()
op_vstack()
op_where()
op_zeros()
op_zeros_like()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()

op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()

op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_ndim()
op_negative()
op_nonzero()

op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()

op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

---

op_ndim *Return the number of dimensions of a tensor.*

Description

Return the number of dimensions of a tensor.

Usage

```
op_ndim(x)
```

Arguments

x	Input tensor.

Value

The number of dimensions in x.

See Also

- https://keras.io/api/ops/numpy#ndim-function

Other numpy ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_bincount()
op_broadcast_to()
op_ceil()
op_clip()
op_concatenate()
op_conj()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_cumprod()
op_cumsum()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_einsum()
op_empty()

op_equal()
op_exp()
op_expand_dims()
op_expm1()
op_eye()
op_flip()
op_floor()
op_floor_divide()
op_full()
op_full_like()
op_get_item()
op_greater()
op_greater_equal()
op_hstack()
op_identity()
op_imag()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_matmul()
op_max()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moveaxis()
op_multiply()
op_nan_to_num()
op_negative()
op_nonzero()
op_not_equal()

op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_repeat()
op_reshape()
op_roll()
op_round()
op_sign()
op_sin()
op_sinh()
op_size()
op_sort()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_subtract()
op_sum()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_var()
op_vdot()
op_vstack()
op_where()
op_zeros()
op_zeros_like()

Other ops:
op_abs()

op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()

op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()

op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()

op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()

op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

---

op_negative *Numerical negative, element-wise.*

Description

Note that this function is automatically called when using the unary R operator - with a tensor.

```
(x <- op_arange(4))
```

```
## tf.Tensor([0. 1. 2. 3.], shape=(4), dtype=float64)
```

```
op_negative(x)
```

```
## tf.Tensor([-0. -1. -2. -3.], shape=(4), dtype=float64)
```

```
-x
```

```
## tf.Tensor([-0. -1. -2. -3.], shape=(4), dtype=float64)
```

Usage

```
op_negative(x)
```

Arguments

x	Input tensor.

Value

Output tensor, y = -x.

See Also

- https://keras.io/api/ops/numpy#negative-function

Other numpy ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_bincount()
op_broadcast_to()
op_ceil()
op_clip()
op_concatenate()
op_conj()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_cumprod()
op_cumsum()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()

op_divide_no_nan()
op_dot()
op_einsum()
op_empty()
op_equal()
op_exp()
op_expand_dims()
op_expm1()
op_eye()
op_flip()
op_floor()
op_floor_divide()
op_full()
op_full_like()
op_get_item()
op_greater()
op_greater_equal()
op_hstack()
op_identity()
op_imag()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_matmul()
op_max()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moveaxis()
op_multiply()

op_nan_to_num()
op_ndim()
op_nonzero()
op_not_equal()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_repeat()
op_reshape()
op_roll()
op_round()
op_sign()
op_sin()
op_sinh()
op_size()
op_sort()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_subtract()
op_sum()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_var()
op_vdot()
op_vstack()
op_where()
op_zeros()

op_zeros_like()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()

op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()

op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()

op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()

op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

op_nonzero *Return the indices of the elements that are non-zero.*

Description

Return the indices of the elements that are non-zero.

Usage

```
op_nonzero(x)
```

Arguments

x	Input tensor.

Value

Indices of elements that are non-zero.

See Also

- https://keras.io/api/ops/numpy#nonzero-function

Other numpy ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_bincount()
op_broadcast_to()
op_ceil()
op_clip()
op_concatenate()
op_conj()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_cumprod()
op_cumsum()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_einsum()
op_empty()
op_equal()
op_exp()
op_expand_dims()
op_expm1()

op_eye()
op_flip()
op_floor()
op_floor_divide()
op_full()
op_full_like()
op_get_item()
op_greater()
op_greater_equal()
op_hstack()
op_identity()
op_imag()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_matmul()
op_max()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moveaxis()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_not_equal()
op_ones()
op_ones_like()
op_outer()
op_pad()

op_power()
op_prod()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_repeat()
op_reshape()
op_roll()
op_round()
op_sign()
op_sin()
op_sinh()
op_size()
op_sort()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_subtract()
op_sum()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_var()
op_vdot()
op_vstack()
op_where()
op_zeros()
op_zeros_like()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()

op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()

op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()

op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()

op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()

op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

---

op_norm *Matrix or vector norm.*

Description

This function is able to return one of eight different matrix norms, or one of an infinite number of vector norms (described below), depending on the value of the `ord` parameter.

Usage

```
op_norm(x, ord = NULL, axis = NULL, keepdims = FALSE)
```

Arguments

`x`	Input tensor.
`ord`	Order of the norm (see table under Notes). The default is `NULL`.
`axis`	If `axis` is an integer, it specifies the axis of `x` along which to compute the vector norms. If `axis` is a length 2 vector, it specifies the axes that hold 2-D matrices, and the matrix norms of these matrices are computed.
`keepdims`	If this is set to `TRUE`, the axes which are reduced are left in the result as dimensions with size one.

Value

Norm of the matrix or vector(s).

Note

For values of `ord` < 1, the result is, strictly speaking, not a mathematical 'norm', but it may still be useful for various numerical purposes. The following norms can be calculated:

- For matrices:
  - `ord=NULL`: Frobenius norm
  - `ord="fro"`: Frobenius norm
  - `ord="nuc"`: nuclear norm
  - `ord=Inf`: `max(sum(abs(x), axis=2))`
  - `ord=-Inf`: `min(sum(abs(x), axis=2))`
  - `ord=0`: not supported
  - `ord=1`: `max(sum(abs(x), axis=1))`
  - `ord=-1`: `min(sum(abs(x), axis=1))`
  - `ord=2`: 2-norm (largest sing. value)
  - `ord=-2`: smallest singular value
  - other: not supported
- For vectors:
  - `ord=NULL`: 2-norm
  - `ord="fro"`: not supported
  - `ord="nuc"`: not supported
  - `ord=Inf`: `max(abs(x))`
  - `ord=-Inf`: `min(abs(x))`
  - `ord=0`: `sum(x != 0)`
  - `ord=1`: as below
  - `ord=-1`: as below
  - `ord=2`: as below
  - `ord=-2`: as below
  - other: `sum(abs(x)^ord)^(1/ord)`

Examples

```
x <- op_reshape(op_arange(9, dtype="float32") - 4, c(3, 3))
op_norm(x)

## tf.Tensor(7.745967, shape=(), dtype=float32)


# 7.7459664
```

See Also

Other linear algebra ops:
`op_cholesky()`
`op_det()`

op_eig()
op_inv()
op_lu_factor()
op_solve_triangular()
op_svd()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()

op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()

op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_normalize()
op_not_equal()
op_one_hot()

op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()

op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

---

op_normalize *Normalizes* `x` *over the specified axis.*

Description

It is defined as: `normalize(x) = x / max(norm(x), epsilon)`.

Usage

```
op_normalize(x, axis = -1L, order = 2L)
```

Arguments

x	Input tensor.
axis	The axis or axes along which to perform normalization. Default to -1.
order	The exponent value in the norm formulation. Defaults to 2.

Value

The normalized array.

Examples

```
x <- op_convert_to_tensor(rbind(c(1, 2, 3), c(4, 5, 6)))
x_norm <- op_normalize(x)
x_norm

## tf.Tensor(
## [[0.26726124 0.53452248 0.80178373]
##  [0.45584231 0.56980288 0.68376346]], shape=(2, 3), dtype=float64)
```

See Also

- https://www.tensorflow.org/api_docs/python/tf/keras/ops/normalize

Other nn ops:
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_categorical_crossentropy()
op_conv()
op_conv_transpose()
op_ctc_loss()
op_depthwise_conv()
op_elu()
op_gelu()
op_hard_sigmoid()
op_hard_silu()
op_leaky_relu()
op_log_sigmoid()
op_log_softmax()
op_max_pool()
op_moments()
op_multi_hot()
op_one_hot()
op_relu()
op_relu6()
op_selu()
op_separable_conv()
op_sigmoid()
op_silu()
op_softmax()
op_softplus()
op_softsign()
op_sparse_categorical_crossentropy()

Other ops:
op_abs()
op_add()

op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()

op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()

op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()

op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()

op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

---

op_not_equal *Return* (x1 != x2) *element-wise.*

Description

Note that this function is automatically called when using the R operator != with a tensor.

```
(x <- op_arange(4))

## tf.Tensor([0. 1. 2. 3.], shape=(4), dtype=float64)


op_not_equal(x, 2)

## tf.Tensor([ True  True False  True], shape=(4), dtype=bool)


x != 2

## tf.Tensor([ True  True False  True], shape=(4), dtype=bool)
```

Usage

```
op_not_equal(x1, x2)
```

Arguments

x1	First input tensor.
x2	Second input tensor.

Value

Output tensor, element-wise comparsion of x1 and x2.

See Also

- https://keras.io/api/ops/numpy#notequal-function

Other numpy ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_bincount()
op_broadcast_to()
op_ceil()
op_clip()
op_concatenate()
op_conj()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_cumprod()
op_cumsum()
op_diag()
op_diagonal()
op_diff()
op_digitize()

op_divide()
op_divide_no_nan()
op_dot()
op_einsum()
op_empty()
op_equal()
op_exp()
op_expand_dims()
op_expm1()
op_eye()
op_flip()
op_floor()
op_floor_divide()
op_full()
op_full_like()
op_get_item()
op_greater()
op_greater_equal()
op_hstack()
op_identity()
op_imag()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_matmul()
op_max()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moveaxis()

op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_repeat()
op_reshape()
op_roll()
op_round()
op_sign()
op_sin()
op_sinh()
op_size()
op_sort()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_subtract()
op_sum()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_var()
op_vdot()
op_vstack()
op_where()

op_zeros()
op_zeros_like()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()

op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()

op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_one_hot()
op_ones()
op_ones_like()
op_outer()

op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()

op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

op_ones *Return a new tensor of given shape and type, filled with ones.*

Description

Return a new tensor of given shape and type, filled with ones.

Usage

```
op_ones(shape, dtype = NULL)
```

Arguments

shape	Shape of the new tensor.
dtype	Desired data type of the tensor.

Value

Tensor of ones with the given shape and dtype.

See Also

- https://keras.io/api/ops/numpy#ones-function

Other numpy ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_bincount()
op_broadcast_to()
op_ceil()
op_clip()
op_concatenate()
op_conj()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_cumprod()
op_cumsum()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_einsum()
op_empty()
op_equal()
op_exp()
op_expand_dims()
op_expm1()

op_eye()
op_flip()
op_floor()
op_floor_divide()
op_full()
op_full_like()
op_get_item()
op_greater()
op_greater_equal()
op_hstack()
op_identity()
op_imag()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_matmul()
op_max()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moveaxis()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_not_equal()
op_ones_like()
op_outer()
op_pad()

op_power()
op_prod()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_repeat()
op_reshape()
op_roll()
op_round()
op_sign()
op_sin()
op_sinh()
op_size()
op_sort()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_subtract()
op_sum()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_var()
op_vdot()
op_vstack()
op_where()
op_zeros()
op_zeros_like()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()

op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()

op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()

op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()

op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()

op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

op_ones_like *Return a tensor of ones with the same shape and type of* x.

Description

Return a tensor of ones with the same shape and type of x.

Usage

```
op_ones_like(x, dtype = NULL)
```

Arguments

x	Input tensor.
dtype	Overrides the data type of the result.

Value

A tensor of ones with the same shape and type as x.

See Also

- https://keras.io/api/ops/numpy#oneslike-function

Other numpy ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()

op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_bincount()
op_broadcast_to()
op_ceil()
op_clip()
op_concatenate()
op_conj()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_cumprod()
op_cumsum()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_einsum()
op_empty()
op_equal()
op_exp()
op_expand_dims()
op_expm1()
op_eye()
op_flip()
op_floor()
op_floor_divide()
op_full()
op_full_like()
op_get_item()
op_greater()
op_greater_equal()
op_hstack()

op_identity()
op_imag()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_matmul()
op_max()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moveaxis()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_not_equal()
op_ones()
op_outer()
op_pad()
op_power()
op_prod()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_repeat()
op_reshape()
op_roll()
op_round()

op_sign()
op_sin()
op_sinh()
op_size()
op_sort()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_subtract()
op_sum()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_var()
op_vdot()
op_vstack()
op_where()
op_zeros()
op_zeros_like()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()

op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()

op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()

op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()

op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()

op_where()
op_while_loop()
op_zeros()
op_zeros_like()

op_one_hot *Converts integer tensor* x *into a one-hot tensor.*

Description

The one-hot encoding is a representation where each integer value is converted into a binary vector with a length equal to num_classes, and the index corresponding to the integer value is marked as 1, while all other indices are marked as 0.

Usage

```
op_one_hot(x, num_classes, axis = -1L, dtype = NULL, sparse = FALSE)
```

Arguments

x	Integer tensor to be encoded. The shape can be arbitrary, but the dtype should be integer. R factors are coerced to integer and offset to be 0-based, i.e., as.integer(x) - 1L.
num_classes	Number of classes for the one-hot encoding.
axis	Axis along which the encoding is performed. Defaults to -1, which represents the last axis.
dtype	(Optional) Data type of the output tensor. If not provided, it defaults to the default data type of the backend.
sparse	Whether to return a sparse tensor; for backends that support sparse tensors.

Value

Integer tensor: One-hot encoded tensor with the same shape as x except for the specified axis dimension, which will have a length of num_classes. The dtype of the output tensor is determined by dtype or the default data type of the backend.

Examples

```
x <- op_array(c(1, 3, 2, 0), "int32")
op_one_hot(x, num_classes = 4)

## tf.Tensor(
## [[0. 1. 0. 0.]
##  [0. 0. 0. 1.]
##  [0. 0. 1. 0.]
##  [1. 0. 0. 0.]], shape=(4, 4), dtype=float32)
```

```
# array([[0. 1. 0. 0.]
#        [0. 0. 0. 1.]
#        [0. 0. 1. 0.]
#        [1. 0. 0. 0.]], shape=(4, 4), dtype=float32)
```

See Also

- https://keras.io/api/ops/nn#onehot-function

Other nn ops:
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_categorical_crossentropy()
op_conv()
op_conv_transpose()
op_ctc_loss()
op_depthwise_conv()
op_elu()
op_gelu()
op_hard_sigmoid()
op_hard_silu()
op_leaky_relu()
op_log_sigmoid()
op_log_softmax()
op_max_pool()
op_moments()
op_multi_hot()
op_normalize()
op_relu()
op_relu6()
op_selu()
op_separable_conv()
op_sigmoid()
op_silu()
op_softmax()
op_softplus()
op_softsign()
op_sparse_categorical_crossentropy()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()

op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()

op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()

op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()

op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()

op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

---

op_outer *Compute the outer product of two vectors.*

Description

Given two vectors x1 and x2, the outer product is:

```
out[i, j] = x1[i] * x2[j]
```

Usage

```
op_outer(x1, x2)
```

Arguments

x1	First input tensor.
x2	Second input tensor.

Value

Outer product of x1 and x2.

See Also

- https://keras.io/api/ops/numpy#outer-function

Other numpy ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()

op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_bincount()
op_broadcast_to()
op_ceil()
op_clip()
op_concatenate()
op_conj()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_cumprod()
op_cumsum()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_einsum()
op_empty()
op_equal()
op_exp()
op_expand_dims()
op_expm1()
op_eye()
op_flip()
op_floor()
op_floor_divide()
op_full()
op_full_like()
op_get_item()
op_greater()
op_greater_equal()
op_hstack()
op_identity()

op_imag()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_matmul()
op_max()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moveaxis()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_not_equal()
op_ones()
op_ones_like()
op_pad()
op_power()
op_prod()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_repeat()
op_reshape()
op_roll()
op_round()
op_sign()

op_sin()
op_sinh()
op_size()
op_sort()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_subtract()
op_sum()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_var()
op_vdot()
op_vstack()
op_where()
op_zeros()
op_zeros_like()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()

op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()

op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()

op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()

op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()

op_while_loop()
op_zeros()
op_zeros_like()

op_pad *Pad a tensor.*

Description

Pad a tensor.

Usage

```
op_pad(x, pad_width, mode = "constant", constant_values = NULL)
```

Arguments

x	Tensor to pad.
pad_width	Number of values padded to the edges of each axis. `((before_1, after_1), ...(before_N, after_N))` unique pad widths for each axis. `((before, after),)` yields same before and after pad for each axis. `(pad,)` or `int` is a shortcut for `before = after = pad` width for all axes.
mode	One of `"constant"`, `"edge"`, `"linear_ramp"`, `"maximum"`, `"mean"`, `"median"`, `"minimum"`, `"reflect"`, `"symmetric"`, `"wrap"`, `"empty"`, `"circular"`. Defaults to `"constant"`.
constant_values	Value to pad with if `mode == "constant"`. Defaults to `0`. A `ValueError` is raised if not `NULL` and `mode != "constant"`.

Value

Padded tensor.

Note

Torch backend only supports modes `"constant"`, `"reflect"`, `"symmetric"` and `"circular"`. Only Torch backend supports `"circular"` mode.

Note: Tensorflow backend only supports modes `"constant"`, `"reflect"` and `"symmetric"`.

See Also

- https://keras.io/api/ops/numpy#pad-function

Other numpy ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_bincount()
op_broadcast_to()
op_ceil()
op_clip()
op_concatenate()
op_conj()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_cumprod()
op_cumsum()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_einsum()
op_empty()
op_equal()
op_exp()
op_expand_dims()
op_expm1()

op_eye()
op_flip()
op_floor()
op_floor_divide()
op_full()
op_full_like()
op_get_item()
op_greater()
op_greater_equal()
op_hstack()
op_identity()
op_imag()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_matmul()
op_max()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moveaxis()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_not_equal()
op_ones()
op_ones_like()
op_outer()

op_power()
op_prod()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_repeat()
op_reshape()
op_roll()
op_round()
op_sign()
op_sin()
op_sinh()
op_size()
op_sort()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_subtract()
op_sum()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_var()
op_vdot()
op_vstack()
op_where()
op_zeros()
op_zeros_like()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()

op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()

op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()

op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()

op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()

op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

---

op_power *First tensor elements raised to powers from second tensor, element-wise.*

Description

Note that this function is automatically called when using the R operator ^ with a tensor.

```
(x <- op_arange(4))

## tf.Tensor([0. 1. 2. 3.], shape=(4), dtype=float64)


op_power(2, x)

## tf.Tensor([1. 2. 4. 8.], shape=(4), dtype=float64)


2 ^ x

## tf.Tensor([1. 2. 4. 8.], shape=(4), dtype=float64)
```

Usage

```
op_power(x1, x2)
```

Arguments

x1	The bases.
x2	The exponents.

Value

Output tensor, the bases in x1 raised to the exponents in x2.

See Also

- https://keras.io/api/ops/numpy#power-function

Other numpy ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_bincount()
op_broadcast_to()
op_ceil()
op_clip()
op_concatenate()
op_conj()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_cumprod()
op_cumsum()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_einsum()
op_empty()

op_equal()
op_exp()
op_expand_dims()
op_expm1()
op_eye()
op_flip()
op_floor()
op_floor_divide()
op_full()
op_full_like()
op_get_item()
op_greater()
op_greater_equal()
op_hstack()
op_identity()
op_imag()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_matmul()
op_max()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moveaxis()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()

op_not_equal()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_prod()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_repeat()
op_reshape()
op_roll()
op_round()
op_sign()
op_sin()
op_sinh()
op_size()
op_sort()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_subtract()
op_sum()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_var()
op_vdot()
op_vstack()
op_where()
op_zeros()
op_zeros_like()

Other ops:
op_abs()

op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()

op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()

op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_prod()
op_qr()
op_quantile()

op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()

op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

---

op_prod *Return the product of tensor elements over a given axis.*

Description

Return the product of tensor elements over a given axis.

Usage

```
op_prod(x, axis = NULL, keepdims = FALSE, dtype = NULL)
```

Arguments

x	Input tensor.
axis	Axis or axes along which a product is performed. The default, axis = NULL, will compute the product of all elements in the input tensor.
keepdims	If this is set to TRUE, the axes which are reduce are left in the result as dimensions with size one.
dtype	Data type of the returned tensor.

Value

Product of elements of x over the given axis or axes.

See Also

- https://keras.io/api/ops/numpy#prod-function

Other numpy ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_bincount()
op_broadcast_to()
op_ceil()
op_clip()
op_concatenate()
op_conj()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_cumprod()
op_cumsum()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_einsum()
op_empty()
op_equal()
op_exp()
op_expand_dims()
op_expm1()

op_eye()
op_flip()
op_floor()
op_floor_divide()
op_full()
op_full_like()
op_get_item()
op_greater()
op_greater_equal()
op_hstack()
op_identity()
op_imag()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_matmul()
op_max()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moveaxis()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_not_equal()
op_ones()
op_ones_like()
op_outer()

op_pad()
op_power()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_repeat()
op_reshape()
op_roll()
op_round()
op_sign()
op_sin()
op_sinh()
op_size()
op_sort()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_subtract()
op_sum()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_var()
op_vdot()
op_vstack()
op_where()
op_zeros()
op_zeros_like()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()

op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()

op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()

op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()

op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()

op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

`op_qr` *Computes the QR decomposition of a tensor.*

Description

Computes the QR decomposition of a tensor.

Usage

```
op_qr(x, mode = "reduced")
```

Arguments

`x`	Input tensor of shape `(..., M, N)`.
`mode`	A string specifying the mode of the QR decomposition. • 'reduced': Returns the reduced QR decomposition. (default) • 'complete': Returns the complete QR decomposition.

Value

A list containing two tensors. The first tensor of shape `(..., M, K)` is the orthogonal matrix `q` and the second tensor of shape (..., K, N)`is the upper triangular matrixr, where` K = min(M, N)‘.

Examples

```
x <- op_convert_to_tensor(rbind(c(1, 2), c(3, 4), c(5, 6)))
op_qr(x)
```

```
## $q
## tf.Tensor(
## [[-0.16903085  0.89708523]
##  [-0.50709255  0.27602622]
##  [-0.84515425 -0.34503278]], shape=(3, 2), dtype=float64)
##
## $r
## tf.Tensor(
## [[-5.91607978 -7.43735744]
##  [ 0.          0.82807867]], shape=(2, 2), dtype=float64)
```

```
c(q, r) %<-% op_qr(x)
```

See Also

- `https://keras.io/api/ops/core#qr-function`

Other math ops:
`op_erf()`
`op_erfinv()`
`op_extract_sequences()`
`op_fft()`
`op_fft2()`
`op_in_top_k()`
`op_irfft()`
`op_istft()`
`op_logsumexp()`
`op_rfft()`
`op_rsqrt()`
`op_segment_max()`
`op_segment_sum()`
`op_solve()`
`op_stft()`
`op_top_k()`

Other ops:
`op_abs()`
`op_add()`
`op_all()`
`op_any()`
`op_append()`
`op_arange()`
`op_arccos()`
`op_arccosh()`
`op_arcsin()`
`op_arcsinh()`
`op_arctan()`
`op_arctan2()`

op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()

op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()

op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()

op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()

op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

---

op_quantile *Compute the q-th quantile(s) of the data along the specified axis.*

Description

Compute the q-th quantile(s) of the data along the specified axis.

Usage

```
op_quantile(x, q, axis = NULL, method = "linear", keepdims = FALSE)
```

Arguments

x	Input tensor.
q	Probability or sequence of probabilities for the quantiles to compute. Values must be between 0 and 1 inclusive.
axis	Axis or axes along which the quantiles are computed. Defaults to `axis=NULL` which is to compute the quantile(s) along a flattened version of the array.
method	A string specifies the method to use for estimating the quantile. Available methods are `"linear"`, `"lower"`, `"higher"`, `"midpoint"`, and `"nearest"`. Defaults to `"linear"`. If the desired quantile lies between two data points `i < j`: • `"linear"`: `i + (j - i) * fraction`, where fraction is the fractional part of the index surrounded by `i` and `j`. • `"lower"`: `i`. • `"higher"`: `j`. • `"midpoint"`: `(i + j) / 2` • `"nearest"`: `i` or `j`, whichever is nearest.
keepdims	If this is set to `TRUE`, the axes which are reduce are left in the result as dimensions with size one.

Value

The quantile(s). If `q` is a single probability and `axis=NULL`, then the result is a scalar. If multiple probabilies levels are given, first axis of the result corresponds to the quantiles. The other axes are the axes that remain after the reduction of `x`.

See Also

Other numpy ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_bincount()
op_broadcast_to()
op_ceil()
op_clip()
op_concatenate()
op_conj()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_cumprod()
op_cumsum()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_einsum()
op_empty()
op_equal()
op_exp()
op_expand_dims()
op_expm1()
op_eye()

op_flip()
op_floor()
op_floor_divide()
op_full()
op_full_like()
op_get_item()
op_greater()
op_greater_equal()
op_hstack()
op_identity()
op_imag()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_matmul()
op_max()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moveaxis()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_not_equal()
op_ones()
op_ones_like()
op_outer()
op_pad()

op_power()
op_prod()
op_ravel()
op_real()
op_reciprocal()
op_repeat()
op_reshape()
op_roll()
op_round()
op_sign()
op_sin()
op_sinh()
op_size()
op_sort()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_subtract()
op_sum()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_var()
op_vdot()
op_vstack()
op_where()
op_zeros()
op_zeros_like()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()

op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()

op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()

op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()

op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()

op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

---

op_ravel *Return a contiguous flattened tensor.*

Description

A 1-D tensor, containing the elements of the input, is returned.

Usage

```
op_ravel(x)
```

Arguments

x	Input tensor.

Value

Output tensor.

See Also

- https://keras.io/api/ops/numpy#ravel-function

Other numpy ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()

op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_bincount()
op_broadcast_to()
op_ceil()
op_clip()
op_concatenate()
op_conj()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_cumprod()
op_cumsum()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_einsum()
op_empty()
op_equal()
op_exp()
op_expand_dims()
op_expm1()
op_eye()
op_flip()
op_floor()
op_floor_divide()
op_full()
op_full_like()
op_get_item()
op_greater()
op_greater_equal()
op_hstack()
op_identity()
op_imag()

op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_matmul()
op_max()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moveaxis()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_not_equal()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_quantile()
op_real()
op_reciprocal()
op_repeat()
op_reshape()
op_roll()
op_round()
op_sign()
op_sin()

op_sinh()
op_size()
op_sort()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_subtract()
op_sum()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_var()
op_vdot()
op_vstack()
op_where()
op_zeros()
op_zeros_like()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()

op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()

op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()

op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()

op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()

op_zeros()
op_zeros_like()

op_real	*Return the real part of the complex argument.*

Description

Return the real part of the complex argument.

Usage

```
op_real(x)
```

Arguments

x	Input tensor.

Value

The real component of the complex argument.

See Also

- https://keras.io/api/ops/numpy#real-function

Other numpy ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_bincount()
op_broadcast_to()

op_ceil()
op_clip()
op_concatenate()
op_conj()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_cumprod()
op_cumsum()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_einsum()
op_empty()
op_equal()
op_exp()
op_expand_dims()
op_expm1()
op_eye()
op_flip()
op_floor()
op_floor_divide()
op_full()
op_full_like()
op_get_item()
op_greater()
op_greater_equal()
op_hstack()
op_identity()
op_imag()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()

op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_matmul()
op_max()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moveaxis()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_not_equal()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_quantile()
op_ravel()
op_reciprocal()
op_repeat()
op_reshape()
op_roll()
op_round()
op_sign()
op_sin()
op_sinh()
op_size()
op_sort()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_subtract()
op_sum()

op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_var()
op_vdot()
op_vstack()
op_where()
op_zeros()
op_zeros_like()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()

op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()

op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()

op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()

op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

`op_reciprocal` *Return the reciprocal of the argument, element-wise.*

Description

Calculates `1/x`.

Usage

```
op_reciprocal(x)
```

Arguments

x	Input tensor.

Value

Output tensor, element-wise reciprocal of x.

See Also

- https://keras.io/api/ops/numpy#reciprocal-function

Other numpy ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_bincount()
op_broadcast_to()
op_ceil()
op_clip()
op_concatenate()
op_conj()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_cumprod()
op_cumsum()
op_diag()

op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_einsum()
op_empty()
op_equal()
op_exp()
op_expand_dims()
op_expm1()
op_eye()
op_flip()
op_floor()
op_floor_divide()
op_full()
op_full_like()
op_get_item()
op_greater()
op_greater_equal()
op_hstack()
op_identity()
op_imag()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_matmul()
op_max()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()

op_minimum()
op_mod()
op_moveaxis()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_not_equal()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_quantile()
op_ravel()
op_real()
op_repeat()
op_reshape()
op_roll()
op_round()
op_sign()
op_sin()
op_sinh()
op_size()
op_sort()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_subtract()
op_sum()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_var()

op_vdot()
op_vstack()
op_where()
op_zeros()
op_zeros_like()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()

op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()

op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()

op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()

op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

`op_relu` *Rectified linear unit activation function.*

Description

It is defined as `f(x) = max(0, x)`.

Usage

```
op_relu(x)
```

Arguments

x	Input tensor.

Value

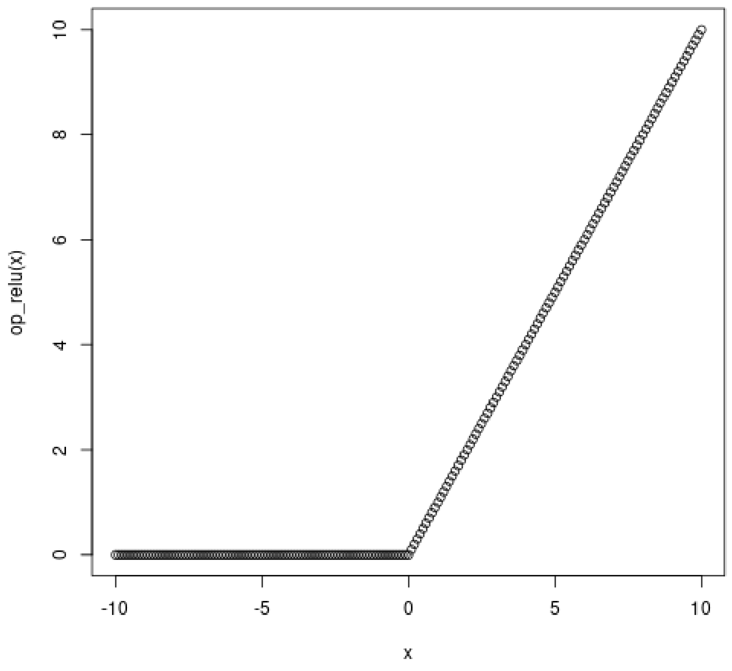
A tensor with the same shape as `x`.

Examples

```
x1 <- op_convert_to_tensor(c(-1, 0, 1, 0.2))
op_relu(x1)

## tf.Tensor([0.  0.  1.  0.2], shape=(4), dtype=float32)


x <- seq(-10, 10, .1)
plot(x, op_relu(x))
```



See Also

- https://keras.io/api/ops/nn#relu-function

Other nn ops:
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_categorical_crossentropy()
op_conv()
op_conv_transpose()
op_ctc_loss()
op_depthwise_conv()

op_elu()
op_gelu()
op_hard_sigmoid()
op_hard_silu()
op_leaky_relu()
op_log_sigmoid()
op_log_softmax()
op_max_pool()
op_moments()
op_multi_hot()
op_normalize()
op_one_hot()
op_relu6()
op_selu()
op_separable_conv()
op_sigmoid()
op_silu()
op_softmax()
op_softplus()
op_softsign()
op_sparse_categorical_crossentropy()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()

op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()

op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()

op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()

op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

---

op_relu6 *Rectified linear unit activation function with upper bound of 6.*

Description

It is defined as `f(x) = op_clip(x, 0, 6)`.

Usage

```
op_relu6(x)
```

Arguments

x	Input tensor.

Value

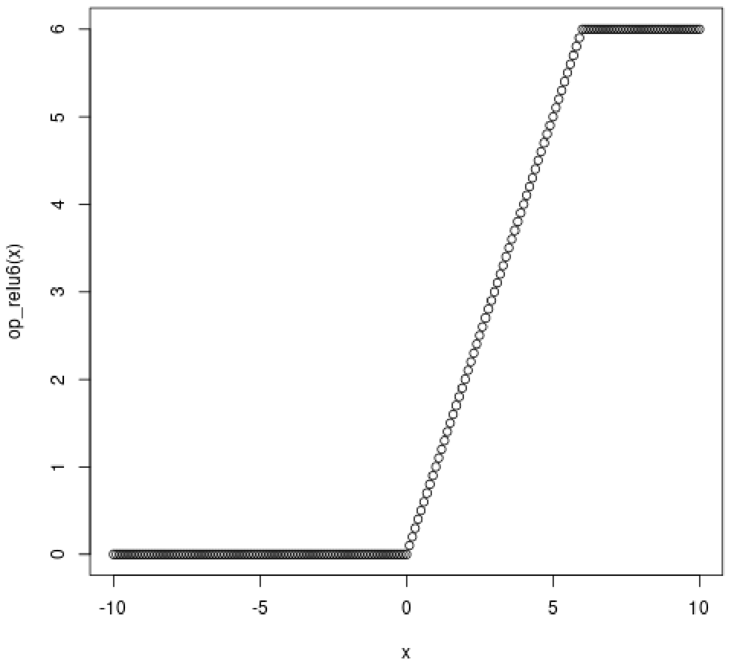
A tensor with the same shape as `x`.

Examples

```
x <- op_convert_to_tensor(c(-3, -2, 0.1, 0.2, 6, 8))
op_relu6(x)
```

```
## tf.Tensor([0.  0.  0.1 0.2 6.  6. ], shape=(6), dtype=float32)
```

```
x <- seq(-10, 10, .1)
plot(x, op_relu6(x))
```



See Also

- https://keras.io/api/ops/nn#relu6-function

Other nn ops:
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_categorical_crossentropy()
op_conv()
op_conv_transpose()
op_ctc_loss()
op_depthwise_conv()
op_elu()
op_gelu()
op_hard_sigmoid()
op_hard_silu()
op_leaky_relu()
op_log_sigmoid()
op_log_softmax()
op_max_pool()
op_moments()
op_multi_hot()
op_normalize()
op_one_hot()
op_relu()
op_selu()
op_separable_conv()
op_sigmoid()
op_silu()
op_softmax()
op_softplus()
op_softsign()
op_sparse_categorical_crossentropy()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()

op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()

op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()

op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()

op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()

op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

---

op_repeat *Repeat each element of a tensor after themselves.*

Description

Repeat each element of a tensor after themselves.

Usage

```
op_repeat(x, repeats, axis = NULL)
```

Arguments

x	Input tensor.
repeats	The number of repetitions for each element.
axis	The axis along which to repeat values. By default, use the flattened input array, and return a flat output array.

Value

Output tensor.

See Also

- https://keras.io/api/ops/numpy#repeat-function

Other numpy ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()

op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_bincount()
op_broadcast_to()
op_ceil()
op_clip()
op_concatenate()
op_conj()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_cumprod()
op_cumsum()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_einsum()
op_empty()
op_equal()
op_exp()
op_expand_dims()
op_expm1()
op_eye()
op_flip()
op_floor()
op_floor_divide()
op_full()
op_full_like()
op_get_item()
op_greater()
op_greater_equal()
op_hstack()
op_identity()
op_imag()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()

op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_matmul()
op_max()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moveaxis()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_not_equal()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_reshape()
op_roll()
op_round()
op_sign()
op_sin()
op_sinh()
op_size()
op_sort()
op_split()

op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_subtract()
op_sum()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_var()
op_vdot()
op_vstack()
op_where()
op_zeros()
op_zeros_like()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()

op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()

op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()

op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()

op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

op_reshape	*Gives a new shape to a tensor without changing its data.*

Description

Gives a new shape to a tensor without changing its data.

Usage

```
op_reshape(x, newshape)
```

Arguments

x	Input tensor.
newshape	The new shape should be compatible with the original shape. One shape dimension can be -1 in which case the value is inferred from the length of the array and remaining dimensions.

Value

The reshaped tensor.

See Also

- https://keras.io/api/ops/numpy#reshape-function

Other numpy ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_bincount()
op_broadcast_to()
op_ceil()

op_clip()
op_concatenate()
op_conj()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_cumprod()
op_cumsum()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_einsum()
op_empty()
op_equal()
op_exp()
op_expand_dims()
op_expm1()
op_eye()
op_flip()
op_floor()
op_floor_divide()
op_full()
op_full_like()
op_get_item()
op_greater()
op_greater_equal()
op_hstack()
op_identity()
op_imag()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_logaddexp()

op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_matmul()
op_max()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moveaxis()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_not_equal()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_repeat()
op_roll()
op_round()
op_sign()
op_sin()
op_sinh()
op_size()
op_sort()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_subtract()
op_sum()
op_swapaxes()

op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_var()
op_vdot()
op_vstack()
op_where()
op_zeros()
op_zeros_like()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()

op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()

op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()

op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()

op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

`op_rfft` *Real-valued Fast Fourier Transform along the last axis of the input.*

Description

Computes the 1D Discrete Fourier Transform of a real-valued signal over the inner-most dimension of input.

Since the Discrete Fourier Transform of a real-valued signal is Hermitian-symmetric, RFFT only returns the `fft_length / 2 + 1` unique components of the FFT: the zero-frequency term, followed by the `fft_length / 2` positive-frequency terms.

Along the axis RFFT is computed on, if `fft_length` is smaller than the corresponding dimension of the input, the dimension is cropped. If it is larger, the dimension is padded with zeros.

Usage

```
op_rfft(x, fft_length = NULL)
```

Arguments

`x`	Input tensor.
`fft_length`	An integer representing the number of the fft length. If not specified, it is inferred from the length of the last axis of `x`. Defaults to `NULL`.

Value

A list containing two tensors - the real and imaginary parts of the output.

Examples

```
x <- op_convert_to_tensor(c(0, 1, 2, 3, 4))
op_rfft(x)

## [[1]]
## tf.Tensor([10.  -2.5 -2.5], shape=(3), dtype=float32)
##
## [[2]]
## tf.Tensor([0.        3.440955  0.8122992], shape=(3), dtype=float32)


op_rfft(x, 3)

## [[1]]
## tf.Tensor([ 3.  -1.5], shape=(2), dtype=float32)
##
## [[2]]
## tf.Tensor([0.        0.8660254], shape=(2), dtype=float32)
```

See Also

- https://keras.io/api/ops/fft#rfft-function

Other math ops:
`op_erf()`
`op_erfinv()`
`op_extract_sequences()`

op_fft()
op_fft2()
op_in_top_k()
op_irfft()
op_istft()
op_logsumexp()
op_qr()
op_rsqrt()
op_segment_max()
op_segment_sum()
op_solve()
op_stft()
op_top_k()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()

op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()

op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()

op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()

op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

---

op_roll *Roll tensor elements along a given axis.*

Description

Elements that roll beyond the last position are re-introduced at the first.

Usage

```
op_roll(x, shift, axis = NULL)
```

Arguments

x	Input tensor.
shift	The number of places by which elements are shifted.
axis	The axis along which elements are shifted. By default, the array is flattened before shifting, after which the original shape is restored.

Value

Output tensor.

See Also

- https://keras.io/api/ops/numpy#roll-function

Other numpy ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_bincount()
op_broadcast_to()
op_ceil()
op_clip()
op_concatenate()
op_conj()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_cumprod()
op_cumsum()
op_diag()

op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_einsum()
op_empty()
op_equal()
op_exp()
op_expand_dims()
op_expm1()
op_eye()
op_flip()
op_floor()
op_floor_divide()
op_full()
op_full_like()
op_get_item()
op_greater()
op_greater_equal()
op_hstack()
op_identity()
op_imag()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_matmul()
op_max()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()

op_minimum()
op_mod()
op_moveaxis()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_not_equal()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_repeat()
op_reshape()
op_round()
op_sign()
op_sin()
op_sinh()
op_size()
op_sort()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_subtract()
op_sum()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_var()

op_vdot()
op_vstack()
op_where()
op_zeros()
op_zeros_like()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()

op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()

op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()

op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()

op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

---

op_round *Evenly round to the given number of decimals.*

Description

Evenly round to the given number of decimals.

Usage

```
op_round(x, decimals = 0L)
```

Arguments

x	Input tensor.
decimals	Number of decimal places to round to. Defaults to 0.

Value

Output tensor.

See Also

- https://keras.io/api/ops/numpy#round-function

Other numpy ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_bincount()
op_broadcast_to()
op_ceil()
op_clip()
op_concatenate()
op_conj()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_cumprod()
op_cumsum()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_einsum()
op_empty()
op_equal()
op_exp()
op_expand_dims()
op_expm1()

op_eye()
op_flip()
op_floor()
op_floor_divide()
op_full()
op_full_like()
op_get_item()
op_greater()
op_greater_equal()
op_hstack()
op_identity()
op_imag()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_matmul()
op_max()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moveaxis()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_not_equal()
op_ones()
op_ones_like()
op_outer()

op_pad()
op_power()
op_prod()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_repeat()
op_reshape()
op_roll()
op_sign()
op_sin()
op_sinh()
op_size()
op_sort()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_subtract()
op_sum()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_var()
op_vdot()
op_vstack()
op_where()
op_zeros()
op_zeros_like()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()

op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()

op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()

op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()

op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()

op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

op_rsqrt *Computes reciprocal of square root of x element-wise.*

Description

Computes reciprocal of square root of x element-wise.

Usage

```
op_rsqrt(x)
```

Arguments

x	input tensor

Value

A tensor with the same dtype as x.

Examples

```
x <- op_convert_to_tensor(c(1, 10, 100))
op_rsqrt(x)

## tf.Tensor([1.         0.31622776 0.1       ], shape=(3), dtype=float32)


# array([1, 0.31622776, 0.1], dtype=float32)
```

See Also

- https://keras.io/api/ops/core#rsqrt-function

Other math ops:
op_erf()
op_erfinv()
op_extract_sequences()
op_fft()
op_fft2()
op_in_top_k()
op_irfft()
op_istft()
op_logsumexp()
op_qr()
op_rfft()
op_segment_max()
op_segment_sum()
op_solve()
op_stft()
op_top_k()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()

op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()

op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()

op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()

op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

op_scatter *Returns a tensor of shape* shape *where* indices *are set to* values.

Description

At a high level, this operation does zeros[indices] = updates and returns the output. It is equivalent to:

```
output <- op_scatter_update(op_zeros(shape), indices, values)
```

Usage

```
op_scatter(indices, values, shape)
```

Arguments

indices	A tensor or list specifying indices for the values in values.
values	A tensor, the values to be set at indices.
shape	Shape of the output tensor.

Value

A tensor of shape shape where indices are set to values.

Examples

```
indices <- rbind(c(1, 2), c(2, 2))
values <- op_array(c(1, 1))
op_scatter(indices, values, shape= c(2, 2))

## tf.Tensor(
## [[0. 1.]
##  [0. 1.]], shape=(2, 2), dtype=float32)
```

See Also

- https://keras.io/api/ops/core#scatter-function

Other core ops:
op_cast()
op_cond()
op_convert_to_numpy()
op_convert_to_tensor()
op_custom_gradient()
op_fori_loop()
op_is_tensor()
op_scatter_update()
op_shape()
op_slice()
op_slice_update()
op_stop_gradient()
op_unstack()

op_vectorized_map()
op_while_loop()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()

op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()

op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()

op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()

op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

---

op_scatter_update *Update inputs via updates at scattered (sparse) indices.*

Description

At a high level, this operation does `inputs[indices] <- updates`. Assume `inputs` is a tensor of shape `(D1, D2, ..., Dn)`, there are 2 main usages of `scatter_update`.

1. `indices` is a 2D tensor of shape `(num_updates, n)`, where `num_updates` is the number of updates to perform, and `updates` is a 1D tensor of shape `(num_updates)`. For example, if `inputs` is `op_zeros(c(4, 4, 4))`, and we want to update `inputs[2, 3, 4]` and `inputs[1, 2, 4]` as 1, then we can use:

```
inputs <- op_zeros(c(4, 4, 4))
indices <- rbind(c(2, 3, 4),
                 c(1, 2, 4))
updates <- op_array(c(1, 1), "float32")
op_scatter_update(inputs, indices, updates)

## tf.Tensor(
## [[[0. 0. 0. 0.]
##   [0. 0. 0. 1.]
##   [0. 0. 0. 0.]
##   [0. 0. 0. 0.]]
##
##  [[0. 0. 0. 0.]
##   [0. 0. 0. 0.]
##   [0. 0. 0. 1.]
##   [0. 0. 0. 0.]]
##
##  [[0. 0. 0. 0.]
##   [0. 0. 0. 0.]
##   [0. 0. 0. 0.]
##   [0. 0. 0. 0.]]
##
##  [[0. 0. 0. 0.]
##   [0. 0. 0. 0.]
##   [0. 0. 0. 0.]
##   [0. 0. 0. 0.]]], shape=(4, 4, 4), dtype=float32)
```

2 `indices` is a 2D tensor of shape `(num_updates, k)`, where `num_updates` is the number of updates to perform, and `k` (`k <= n`) is the size of each index in `indices`. `updates` is a `n - k`-D tensor of shape `(num_updates, shape(inputs)[-(1:k)])`. For example, if `inputs <- op_zeros(c(4, 4, 4))`, and we want to update `inputs[1, 2, ]` and `inputs[2, 3, ]` as `[1, 1, 1, 1]`, then `indices` would have shape `(num_updates, 2)` (`k = 2`), and `updates` would have shape `(num_updates, 4)` (`shape(inputs)[3:4] == 4`). See the code below:

```
inputs <- op_zeros(c(4, 4, 4))
indices <- rbind(c(2, 3),
                 c(3, 4))
updates <- op_array(rbind(c(1, 1, 1, 1),
                          c(1, 1, 1, 1)),
                    "float32")
op_scatter_update(inputs, indices, updates)
```

```
## tf.Tensor(
## [[[0. 0. 0. 0.]
##   [0. 0. 0. 0.]
##   [0. 0. 0. 0.]
##   [0. 0. 0. 0.]]
##
##  [[0. 0. 0. 0.]
##   [0. 0. 0. 0.]
##   [1. 1. 1. 1.]
##   [0. 0. 0. 0.]]
##
##  [[0. 0. 0. 0.]
##   [0. 0. 0. 0.]
##   [0. 0. 0. 0.]
##   [1. 1. 1. 1.]]
##
##  [[0. 0. 0. 0.]
##   [0. 0. 0. 0.]
##   [0. 0. 0. 0.]
##   [0. 0. 0. 0.]]], shape=(4, 4, 4), dtype=float32)
```

Usage

```
op_scatter_update(inputs, indices, updates)
```

Arguments

inputs	A tensor, the tensor to be updated.
indices	A tensor or list of shape (N, inputs$ndim), specifying indices to update. N is the number of indices to update, must be equal to the first dimension of updates.
updates	A tensor, the new values to be put to inputs at indices.

Value

A tensor, has the same shape and dtype as inputs.

See Also

- https://keras.io/api/ops/core#scatterupdate-function

Other core ops:
op_cast()
op_cond()
op_convert_to_numpy()
op_convert_to_tensor()
op_custom_gradient()
op_fori_loop()
op_is_tensor()
op_scatter()
op_shape()
op_slice()
op_slice_update()
op_stop_gradient()
op_unstack()
op_vectorized_map()
op_while_loop()

Other ops:
op_abs()
op_add()
op_all()
op_any()

op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()

op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()

op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()

op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()

op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

---

op_segment_max *Computes the max of segments in a tensor.*

Description

Computes the max of segments in a tensor.

Usage

```
op_segment_max(data, segment_ids, num_segments = NULL, sorted = FALSE)
```

Arguments

data	Input tensor.
segment_ids	A 1-D tensor containing segment indices for each element in `data`.
num_segments	An integer representing the total number of segments. If not specified, it is inferred from the maximum value in `segment_ids`.
sorted	A boolean indicating whether `segment_ids` is sorted. Defaults to`FALSE`.

Value

A tensor containing the max of segments, where each element represents the max of the corresponding segment in `data`.

Examples

```
data <- op_convert_to_tensor(c(1, 2, 10, 20, 100, 200))
segment_ids <- op_array(c(1, 1, 2, 2, 3, 3), "int32")
num_segments <- 3
op_segment_max(data, segment_ids, num_segments)
```

```
## tf.Tensor([  2.  20. 200.], shape=(3), dtype=float32)


# array([2, 20, 200], dtype=int32)
```

See Also

- https://keras.io/api/ops/core#segmentmax-function

Other math ops:
op_erf()
op_erfinv()
op_extract_sequences()
op_fft()
op_fft2()
op_in_top_k()
op_irfft()
op_istft()
op_logsumexp()
op_qr()
op_rfft()
op_rsqrt()
op_segment_sum()
op_solve()
op_stft()
op_top_k()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()

op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()

op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()

op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()

op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

`op_segment_sum` *Computes the sum of segments in a tensor.*

Description

Computes the sum of segments in a tensor.

Usage

```
op_segment_sum(data, segment_ids, num_segments = NULL, sorted = FALSE)
```

Arguments

`data`	Input tensor.
`segment_ids`	A 1-D tensor containing segment indices for each element in `data`.
`num_segments`	An integer representing the total number of segments. If not specified, it is inferred from the maximum value in `segment_ids`.
`sorted`	A boolean indicating whether `segment_ids` is sorted. Defaults to`FALSE`.

Value

A tensor containing the sum of segments, where each element represents the sum of the corresponding segment in `data`.

Examples

```
data <- op_array(c(1, 2, 10, 20, 100, 200))
segment_ids <- op_array(c(1, 1, 2, 2, 3, 3), "int32")
num_segments <- 3
op_segment_sum(data, segment_ids, num_segments)

## tf.Tensor([  3.  30. 300.], shape=(3), dtype=float32)
```

See Also

- https://keras.io/api/ops/core#segmentsum-function

Other math ops:
`op_erf()`
`op_erfinv()`
`op_extract_sequences()`
`op_fft()`
`op_fft2()`
`op_in_top_k()`
`op_irfft()`
`op_istft()`

op_logsumexp()
op_qr()
op_rfft()
op_rsqrt()
op_segment_max()
op_solve()
op_stft()
op_top_k()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()

op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()

op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()

op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()

op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

op_selu *Scaled Exponential Linear Unit (SELU) activation function.*

Description

It is defined as:
`f(x) = scale * alpha * (exp(x) - 1.) for x < 0`, `f(x) = scale * x for x >= 0`.

Usage

```
op_selu(x)
```

Arguments

x	Input tensor.

Value

A tensor with the same shape as x.

Examples

```
x <- op_array(c(-1, 0, 1))
op_selu(x)

## tf.Tensor([-1.1113307  0.         1.050701 ], shape=(3), dtype=float32)
```

See Also

- https://keras.io/api/ops/nn#selu-function

Other nn ops:
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_categorical_crossentropy()
op_conv()
op_conv_transpose()
op_ctc_loss()
op_depthwise_conv()
op_elu()
op_gelu()
op_hard_sigmoid()
op_hard_silu()
op_leaky_relu()
op_log_sigmoid()
op_log_softmax()
op_max_pool()
op_moments()
op_multi_hot()
op_normalize()
op_one_hot()
op_relu()
op_relu6()
op_separable_conv()
op_sigmoid()
op_silu()
op_softmax()
op_softplus()
op_softsign()
op_sparse_categorical_crossentropy()

Other ops:
op_abs()
op_add()

op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()

op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()

op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()

op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()

op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

op_separable_conv *General N-D separable convolution.*

Description

This ops supports 1D and 2D separable convolution. `separable_conv` is a depthwise conv followed by a pointwise conv.

Usage

```
op_separable_conv(
  inputs,
  depthwise_kernel,
  pointwise_kernel,
  strides = 1L,
  padding = "valid",
  data_format = NULL,
  dilation_rate = 1L
)
```

Arguments

`inputs`	Tensor of rank N+2. `inputs` has shape `(batch_size,) + inputs_spatial_shape + (num_channels,)` if `data_format="channels_last"`, or `(batch_size, num_channels) + inputs_spatial_shape` if `data_format="channels_first"`.
`depthwise_kernel`	Tensor of rank N+2. `depthwise_kernel` has shape `[kernel_spatial_shape, num_input_channels, r` `num_input_channels` should match the number of channels in `inputs`.

pointwise_kernel
: Tensor of rank N+2. `pointwise_kernel` has shape `(*ones_like(kernel_spatial_shape), num_inpu`

strides
: int or int tuple/list of `len(inputs_spatial_shape)`, specifying the strides of the convolution along each spatial dimension. If `strides` is int, then every spatial dimension shares the same `strides`.

padding
: string, either `"valid"` or `"same"`. `"valid"` means no padding is applied, and `"same"` results in padding evenly to the left/right or up/down of the input such that output has the same height/width dimension as the input when `strides=1`.

data_format
: A string, either `"channels_last"` or `"channels_first"`. `data_format` determines the ordering of the dimensions in the inputs. If `data_format="channels_last"`, `inputs` is of shape `(batch_size, ..., channels)` while if `data_format="channels_first"`, `inputs` is of shape `(batch_size, channels, ...)`.

dilation_rate
: int or int tuple/list of `len(inputs_spatial_shape)`, specifying the dilation rate to use for dilated convolution. If `dilation_rate` is int, then every spatial dimension shares the same `dilation_rate`.

Value

A tensor of rank N+2, the result of the depthwise conv operation.

See Also

- https://keras.io/api/ops/nn#separableconv-function

Other nn ops:
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_categorical_crossentropy()
op_conv()
op_conv_transpose()
op_ctc_loss()
op_depthwise_conv()
op_elu()
op_gelu()
op_hard_sigmoid()
op_hard_silu()
op_leaky_relu()
op_log_sigmoid()
op_log_softmax()
op_max_pool()
op_moments()
op_multi_hot()
op_normalize()
op_one_hot()
op_relu()
op_relu6()
op_selu()
op_sigmoid()

op_silu()
op_softmax()
op_softplus()
op_softsign()
op_sparse_categorical_crossentropy()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()

op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()

op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()

op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()

op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

op_shape *Gets the shape of the tensor input.*

Description

Gets the shape of the tensor input.

Usage

```
op_shape(x)
```

Arguments

x	A tensor. This function will try to access the shape attribute of the input tensor.

Value

A list of integers or NULL values, indicating the shape of the input tensor.

Note

On the TensorFlow backend, when `x` is a `tf.Tensor` with dynamic shape, dimensions which are dynamic in the context of a compiled function will have a `tf.Tensor` value instead of a static integer value.

Examples

```
x <- op_zeros(c(8, 12))
op_shape(x)

## shape(8, 12)
```

See Also

- https://keras.io/api/ops/core#shape-function

Other core ops:
op_cast()
op_cond()
op_convert_to_numpy()
op_convert_to_tensor()
op_custom_gradient()
op_fori_loop()
op_is_tensor()
op_scatter()
op_scatter_update()
op_slice()
op_slice_update()
op_stop_gradient()
op_unstack()
op_vectorized_map()
op_while_loop()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()

op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()

op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()

op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()

op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()

op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

---

op_sigmoid *Sigmoid activation function.*

Description

It is defined as `f(x) = 1 / (1 + exp(-x))`.

Usage

```
op_sigmoid(x)
```

Arguments

x	Input tensor.

Value

A tensor with the same shape as `x`.

Examples

```
x <- op_convert_to_tensor(c(-6, 1, 0, 1, 6))
op_sigmoid(x)

## tf.Tensor([0.00247262 0.7310586  0.5       0.7310586  0.99752736], shape=(5), dtype=float32)
```

See Also

- https://keras.io/api/ops/nn#sigmoid-function

Other nn ops:
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_categorical_crossentropy()
op_conv()
op_conv_transpose()
op_ctc_loss()
op_depthwise_conv()
op_elu()

op_gelu()
op_hard_sigmoid()
op_hard_silu()
op_leaky_relu()
op_log_sigmoid()
op_log_softmax()
op_max_pool()
op_moments()
op_multi_hot()
op_normalize()
op_one_hot()
op_relu()
op_relu6()
op_selu()
op_separable_conv()
op_silu()
op_softmax()
op_softplus()
op_softsign()
op_sparse_categorical_crossentropy()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()

op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()

op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()

op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()

op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

op_sign *Returns a tensor with the signs of the elements of* x.

Description

Returns a tensor with the signs of the elements of x.

Usage

```
op_sign(x)
```

Arguments

x	Input tensor.

Value

Output tensor of same shape as x.

See Also

- https://keras.io/api/ops/numpy#sign-function

Other numpy ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_bincount()
op_broadcast_to()
op_ceil()
op_clip()
op_concatenate()
op_conj()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()

op_cross()
op_cumprod()
op_cumsum()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_einsum()
op_empty()
op_equal()
op_exp()
op_expand_dims()
op_expm1()
op_eye()
op_flip()
op_floor()
op_floor_divide()
op_full()
op_full_like()
op_get_item()
op_greater()
op_greater_equal()
op_hstack()
op_identity()
op_imag()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_matmul()
op_max()
op_maximum()

op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moveaxis()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_not_equal()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_repeat()
op_reshape()
op_roll()
op_round()
op_sin()
op_sinh()
op_size()
op_sort()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_subtract()
op_sum()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_trace()
op_transpose()

op_tri()
op_tril()
op_triu()
op_var()
op_vdot()
op_vstack()
op_where()
op_zeros()
op_zeros_like()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()

op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()

op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()

op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()

op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

---

op_silu *Sigmoid Linear Unit (SiLU) activation function, also known as Swish.*

Description

The SiLU activation function is computed by the sigmoid function multiplied by its input. It is defined as `f(x) = x * sigmoid(x)`.

Usage

```
op_silu(x)
```

Arguments

x	Input tensor.

Value

A tensor with the same shape as x.

Examples

```
x <- op_convert_to_tensor(c(-6, 1, 0, 1, 6))
op_sigmoid(x)

## tf.Tensor([0.00247262 0.7310586  0.5       0.7310586  0.99752736], shape=(5), dtype=float32)


op_silu(x)

## tf.Tensor([-0.01483574  0.7310586   0.         0.7310586   5.985164  ], shape=(5), dtype=float32)
```

See Also

- https://keras.io/api/ops/nn#silu-function

Other nn ops:
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_categorical_crossentropy()
op_conv()
op_conv_transpose()
op_ctc_loss()
op_depthwise_conv()
op_elu()
op_gelu()
op_hard_sigmoid()
op_hard_silu()
op_leaky_relu()
op_log_sigmoid()
op_log_softmax()
op_max_pool()
op_moments()
op_multi_hot()
op_normalize()
op_one_hot()
op_relu()
op_relu6()
op_selu()
op_separable_conv()
op_sigmoid()
op_softmax()
op_softplus()
op_softsign()

op_sparse_categorical_crossentropy()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()

op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()

op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()

op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()

op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

op_sin *Trigonometric sine, element-wise.*

Description

Trigonometric sine, element-wise.

Usage

```
op_sin(x)
```

Arguments

x	Input tensor.

Value

Output tensor of same shape as x.

See Also

- https://keras.io/api/ops/numpy#sin-function

Other numpy ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_bincount()
op_broadcast_to()
op_ceil()
op_clip()
op_concatenate()
op_conj()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_cumprod()
op_cumsum()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_einsum()
op_empty()
op_equal()
op_exp()
op_expand_dims()
op_expm1()

op_eye()
op_flip()
op_floor()
op_floor_divide()
op_full()
op_full_like()
op_get_item()
op_greater()
op_greater_equal()
op_hstack()
op_identity()
op_imag()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_matmul()
op_max()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moveaxis()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_not_equal()
op_ones()
op_ones_like()
op_outer()

op_pad()
op_power()
op_prod()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_repeat()
op_reshape()
op_roll()
op_round()
op_sign()
op_sinh()
op_size()
op_sort()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_subtract()
op_sum()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_var()
op_vdot()
op_vstack()
op_where()
op_zeros()
op_zeros_like()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()

op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()

op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()

op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()

op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()

op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

---

op_sinh *Hyperbolic sine, element-wise.*

Description

Hyperbolic sine, element-wise.

Usage

```
op_sinh(x)
```

Arguments

x	Input tensor.

Value

Output tensor of same shape as x.

See Also

- https://keras.io/api/ops/numpy#sinh-function

Other numpy ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()

op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_bincount()
op_broadcast_to()
op_ceil()
op_clip()
op_concatenate()
op_conj()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_cumprod()
op_cumsum()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_einsum()
op_empty()
op_equal()
op_exp()
op_expand_dims()
op_expm1()
op_eye()
op_flip()
op_floor()
op_floor_divide()
op_full()
op_full_like()
op_get_item()
op_greater()
op_greater_equal()
op_hstack()
op_identity()

op_imag()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_matmul()
op_max()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moveaxis()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_not_equal()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_repeat()
op_reshape()
op_roll()
op_round()

op_sign()
op_sin()
op_size()
op_sort()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_subtract()
op_sum()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_var()
op_vdot()
op_vstack()
op_where()
op_zeros()
op_zeros_like()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()

op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()

op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()

op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()

op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()

op_while_loop()
op_zeros()
op_zeros_like()

---

op_size *Return the number of elements in a tensor.*

Description

Return the number of elements in a tensor.

Usage

```
op_size(x)
```

Arguments

x	Input tensor.

Value

Number of elements in x.

See Also

- https://keras.io/api/ops/numpy#size-function

Other numpy ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_bincount()

op_broadcast_to()
op_ceil()
op_clip()
op_concatenate()
op_conj()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_cumprod()
op_cumsum()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_einsum()
op_empty()
op_equal()
op_exp()
op_expand_dims()
op_expm1()
op_eye()
op_flip()
op_floor()
op_floor_divide()
op_full()
op_full_like()
op_get_item()
op_greater()
op_greater_equal()
op_hstack()
op_identity()
op_imag()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()

op_log2()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_matmul()
op_max()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moveaxis()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_not_equal()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_repeat()
op_reshape()
op_roll()
op_round()
op_sign()
op_sin()
op_sinh()
op_sort()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_subtract()

op_sum()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_var()
op_vdot()
op_vstack()
op_where()
op_zeros()
op_zeros_like()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()

op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()

op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()

op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_slice()
op_slice_update()

op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

op_slice *Return a slice of an input tensor.*

Description

At a high level, this operation is an explicit replacement for array slicing e.g. `inputs[start_indices:(start_indices + shape)]`. Unlike slicing via brackets, this operation will accept tensor start indices on all backends, which is useful when indices dynamically computed via other tensor operations.

```
(inputs <- op_arange(5*5) |> op_reshape(c(5, 5)))
```

```
## tf.Tensor(
## [[ 0.  1.  2.  3.  4.]
##  [ 5.  6.  7.  8.  9.]
##  [10. 11. 12. 13. 14.]
##  [15. 16. 17. 18. 19.]
##  [20. 21. 22. 23. 24.]], shape=(5, 5), dtype=float64)
```

```
start_indices <- c(3, 3)
shape <- c(2, 2)
op_slice(inputs, start_indices, shape)
```

```
## tf.Tensor(
## [[12. 13.]
##  [17. 18.]], shape=(2, 2), dtype=float64)
```

Usage

```
op_slice(inputs, start_indices, shape)
```

Arguments

`inputs`	A tensor, the tensor to be sliced.
`start_indices`	A list of length `inputs$ndim`, specifying the starting indices for updating.
`shape`	The full shape of the returned slice.

Value

A tensor, has the same shape and dtype as `inputs`.

See Also

- `https://keras.io/api/ops/core#slice-function`

Other core ops:
`op_cast()`
`op_cond()`
`op_convert_to_numpy()`
`op_convert_to_tensor()`
`op_custom_gradient()`

op_fori_loop()
op_is_tensor()
op_scatter()
op_scatter_update()
op_shape()
op_slice_update()
op_stop_gradient()
op_unstack()
op_vectorized_map()
op_while_loop()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()

op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()

op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()

op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()

op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

---

op_slice_update *Update an input by slicing in a tensor of updated values.*

Description

At a high level, this operation does `inputs[start_indices: start_indices + updates.shape] = updates`. Assume inputs is a tensor of shape `(D1, D2, ..., Dn)`, `start_indices` must be a list of n integers, specifying the starting indices. `updates` must have the same rank as `inputs`, and the size of each dim must not exceed `Di - start_indices[i]`. For example, if we have 2D inputs `inputs = op_zeros(c(5, 5))`, and we want to update the intersection of last 2 rows and last 2 columns as 1, i.e., `inputs[4:5, 4:5] = op_ones(c(2, 2))`, then we can use the code below:

```
inputs <- op_zeros(c(5, 5))
start_indices <- c(3, 3)
```

```
updates <- op_ones(c(2, 2))
op_slice_update(inputs, start_indices, updates)

## tf.Tensor(
## [[0. 0. 0. 0. 0.]
##  [0. 0. 0. 0. 0.]
##  [0. 0. 1. 1. 0.]
##  [0. 0. 1. 1. 0.]
##  [0. 0. 0. 0. 0.]], shape=(5, 5), dtype=float32)
```

Usage

```
op_slice_update(inputs, start_indices, updates)
```

Arguments

`inputs`	A tensor, the tensor to be updated.
`start_indices`	A list of length `inputs$ndim`, specifying the starting indices for updating.
`updates`	A tensor, the new values to be put to `inputs` at `indices`. `updates` must have the same rank as `inputs`.

Value

A tensor, has the same shape and dtype as `inputs`.

See Also

- https://keras.io/api/ops/core#sliceupdate-function

Other core ops:
`op_cast()`
`op_cond()`
`op_convert_to_numpy()`
`op_convert_to_tensor()`
`op_custom_gradient()`
`op_fori_loop()`
`op_is_tensor()`
`op_scatter()`
`op_scatter_update()`
`op_shape()`
`op_slice()`
`op_stop_gradient()`
`op_unstack()`
`op_vectorized_map()`
`op_while_loop()`

Other ops:
`op_abs()`

op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()

op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()

op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()

op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()

op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

---

op_softmax *Softmax activation function.*

Description

The elements of the output vector lie within the range `(0, 1)`, and their total sum is exactly 1 (excluding the floating point rounding error).

Each vector is processed independently. The `axis` argument specifies the axis along which the function is applied within the input.

It is defined as: `f(x) = exp(x) / sum(exp(x))`

Usage

```
op_softmax(x, axis = -1L)
```

Arguments

x	Input tensor.
axis	Integer, axis along which the softmax is applied.

Value

A tensor with the same shape as `x`.

Examples

```
x <- op_array(c(-1, 0, 1))
op_softmax(x)

## tf.Tensor([0.09003057 0.24472848 0.66524094], shape=(3), dtype=float32)
```

See Also

- https://keras.io/api/ops/nn#softmax-function

Other nn ops:
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_categorical_crossentropy()
op_conv()
op_conv_transpose()
op_ctc_loss()
op_depthwise_conv()
op_elu()
op_gelu()
op_hard_sigmoid()
op_hard_silu()
op_leaky_relu()
op_log_sigmoid()
op_log_softmax()
op_max_pool()
op_moments()
op_multi_hot()
op_normalize()
op_one_hot()
op_relu()
op_relu6()
op_selu()
op_separable_conv()
op_sigmoid()
op_silu()
op_softplus()
op_softsign()
op_sparse_categorical_crossentropy()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()

op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()

op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()

op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()

op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()

op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

---

op_softplus *Softplus activation function.*

Description

It is defined as `f(x) = log(exp(x) + 1)`, where `log` is the natural logarithm and `exp` is the exponential function.

Usage

```
op_softplus(x)
```

Arguments

x	Input tensor.

Value

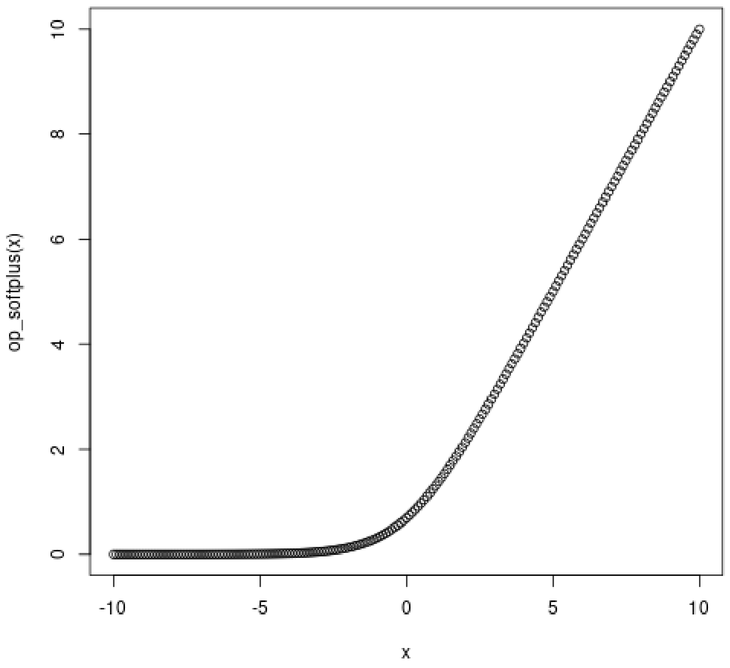
A tensor with the same shape as `x`.

Examples

```
x <- op_convert_to_tensor(c(-0.555, 0, 0.555))
op_softplus(x)

## tf.Tensor([0.45366603 0.6931472  1.008666  ], shape=(3), dtype=float32)


x <- seq(-10, 10, .1)
plot(x, op_softplus(x))
```



See Also

- https://keras.io/api/ops/nn#softplus-function

Other nn ops:
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_categorical_crossentropy()
op_conv()
op_conv_transpose()
op_ctc_loss()
op_depthwise_conv()
op_elu()
op_gelu()
op_hard_sigmoid()
op_hard_silu()
op_leaky_relu()
op_log_sigmoid()
op_log_softmax()
op_max_pool()
op_moments()
op_multi_hot()
op_normalize()

op_one_hot()
op_relu()
op_relu6()
op_selu()
op_separable_conv()
op_sigmoid()
op_silu()
op_softmax()
op_softsign()
op_sparse_categorical_crossentropy()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()

op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()

op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()

op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()

op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

---

op_softsign *Softsign activation function.*

Description

It is defined as `f(x) = x / (abs(x) + 1)`.

Usage

```
op_softsign(x)
```

Arguments

x	Input tensor.

Value

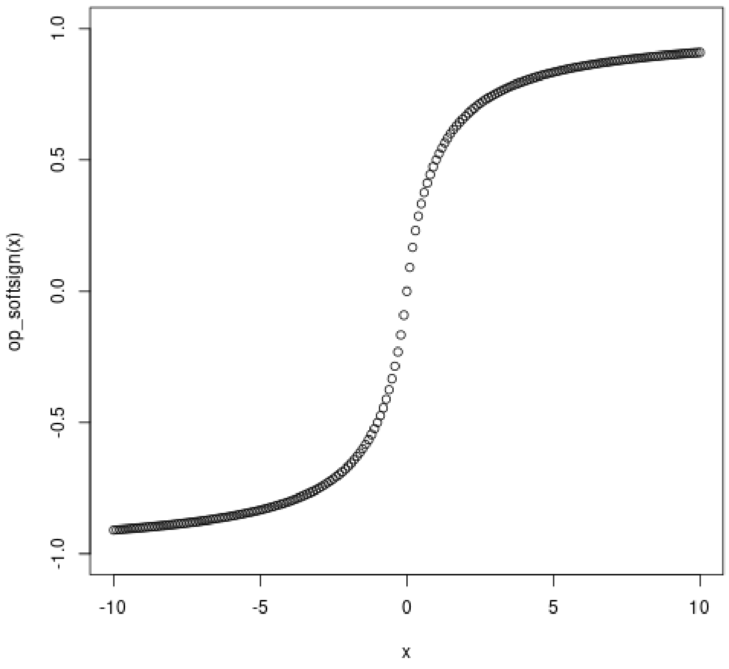
A tensor with the same shape as x.

Examples

```
x <- op_convert_to_tensor(c(-0.100, -10.0, 1.0, 0.0, 100.0))
op_softsign(x)

## tf.Tensor([-0.09090909 -0.90909094  0.5        0.         0.990099  ], shape=(5), dtype=float32)


x <- seq(-10, 10, .1)
plot(x, op_softsign(x), ylim = c(-1, 1))
```



See Also

- https://keras.io/api/ops/nn#softsign-function

Other nn ops:
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_categorical_crossentropy()

op_conv()
op_conv_transpose()
op_ctc_loss()
op_depthwise_conv()
op_elu()
op_gelu()
op_hard_sigmoid()
op_hard_silu()
op_leaky_relu()
op_log_sigmoid()
op_log_softmax()
op_max_pool()
op_moments()
op_multi_hot()
op_normalize()
op_one_hot()
op_relu()
op_relu6()
op_selu()
op_separable_conv()
op_sigmoid()
op_silu()
op_softmax()
op_softplus()
op_sparse_categorical_crossentropy()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()

op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()

op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()

op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()

op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

op_solve *Solves a linear system of equations given by* `a x = b`.

Description

Solves for x in the equation a %*% x == b.

Usage

```
op_solve(a, b)
```

Arguments

a	A tensor of shape `(..., M, M)` representing the coefficients matrix.
b	A tensor of shape `(..., M)` or `(..., M, N)` represeting the right-hand side or "dependent variable" matrix.

Value

A tensor of shape `(..., M)` or `(..., M, N)` representing the solution of the linear system. Returned shape is identical to b.

Examples

```
a <- op_array(c(1, 2, 4, 5), dtype="float32") |> op_reshape(c(2, 2))
b <- op_array(c(2, 4, 8, 10), dtype="float32") |> op_reshape(c(2, 2))
op_solve(a, b)

## tf.Tensor(
## [[2. 0.]
##  [0. 2.]], shape=(2, 2), dtype=float32)
```

See Also

Other math ops:
op_erf()
op_erfinv()
op_extract_sequences()
op_fft()
op_fft2()
op_in_top_k()
op_irfft()
op_istft()
op_logsumexp()
op_qr()
op_rfft()

op_rsqrt()
op_segment_max()
op_segment_sum()
op_stft()
op_top_k()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()

op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()

op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()

op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()

op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

---

op_solve_triangular *Solves a linear system of equations given by* a %*% x = b.

Description

Solves a linear system of equations given by a %*% x = b.

Usage

```
op_solve_triangular(a, b, lower = FALSE)
```

Arguments

a	A tensor of shape (..., M, M) representing the coefficients matrix.
b	A tensor of shape (..., M) or (..., M, N) represeting the right-hand side or "dependent variable" matrix.
lower	logical. Use only data contained in the lower triangle of a. Default is to use upper triangle.

Value

A tensor of shape `(..., M)` or `(..., M, N)` representing the solution of the linear system. Returned shape is identical to `b`.

See Also

Other linear algebra ops:
`op_cholesky()`
`op_det()`
`op_eig()`
`op_inv()`
`op_lu_factor()`
`op_norm()`
`op_svd()`

Other ops:
`op_abs()`
`op_add()`
`op_all()`
`op_any()`
`op_append()`
`op_arange()`
`op_arccos()`
`op_arccosh()`
`op_arcsin()`
`op_arcsinh()`
`op_arctan()`
`op_arctan2()`
`op_arctanh()`
`op_argmax()`
`op_argmin()`
`op_argsort()`
`op_array()`
`op_average()`
`op_average_pool()`
`op_batch_normalization()`
`op_binary_crossentropy()`
`op_bincount()`
`op_broadcast_to()`
`op_cast()`
`op_categorical_crossentropy()`
`op_ceil()`
`op_cholesky()`
`op_clip()`
`op_concatenate()`
`op_cond()`
`op_conj()`
`op_conv()`

op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()

op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()

op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()

op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

---

op_sort *Sorts the elements of* x *along a given axis in ascending order.*

Description

Sorts the elements of x along a given axis in ascending order.

Usage

```
op_sort(x, axis = -1L)
```

Arguments

x	Input tensor.
axis	Axis along which to sort. If NULL, the tensor is flattened before sorting. Defaults to -1; the last axis.

Value

Sorted tensor.

See Also

- https://keras.io/api/ops/numpy#sort-function

Other numpy ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_bincount()
op_broadcast_to()
op_ceil()
op_clip()
op_concatenate()
op_conj()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_cumprod()
op_cumsum()
op_diag()
op_diagonal()
op_diff()

op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_einsum()
op_empty()
op_equal()
op_exp()
op_expand_dims()
op_expm1()
op_eye()
op_flip()
op_floor()
op_floor_divide()
op_full()
op_full_like()
op_get_item()
op_greater()
op_greater_equal()
op_hstack()
op_identity()
op_imag()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_matmul()
op_max()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()

op_moveaxis()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_not_equal()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_repeat()
op_reshape()
op_roll()
op_round()
op_sign()
op_sin()
op_sinh()
op_size()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_subtract()
op_sum()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_var()
op_vdot()
op_vstack()

op_where()
op_zeros()
op_zeros_like()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()

op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()

op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()

op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()

op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

---

op_sparse_categorical_crossentropy
Computes sparse categorical cross-entropy loss.

Description

The sparse categorical cross-entropy loss is similar to categorical cross-entropy, but it is used when the target tensor contains integer class labels instead of one-hot encoded vectors. It measures the dissimilarity between the target and output probabilities or logits.

Usage

```
op_sparse_categorical_crossentropy(
  target,
  output,
  from_logits = FALSE,
  axis = -1L
)
```

Arguments

`target`	The target tensor representing the true class labels as integers. Its shape should match the shape of the `output` tensor except for the last dimension.
`output`	The output tensor representing the predicted probabilities or logits. Its shape should match the shape of the `target` tensor except for the last dimension.
`from_logits`	(optional) Whether `output` is a tensor of logits or probabilities. Set it to `TRUE` if `output` represents logits; otherwise, set it to `FALSE` if `output` represents probabilities. Defaults to`FALSE`.
`axis`	(optional) The axis along which the sparse categorical cross-entropy is computed. Defaults to `-1`, which corresponds to the last dimension of the tensors.

Value

Integer tensor: The computed sparse categorical cross-entropy loss between `target` and `output`.

Examples

```
target <- op_array(c(0, 1, 2), dtype="int32")
output <- op_array(rbind(c(0.9, 0.05, 0.05),
                         c(0.1, 0.8,  0.1),
                         c(0.2, 0.3,  0.5)))
op_sparse_categorical_crossentropy(target, output)

## tf.Tensor([0.10536052 0.22314355 0.69314718], shape=(3), dtype=float64)
```

See Also

- https://keras.io/api/ops/nn#sparsecategoricalcrossentropy-function

Other nn ops:
`op_average_pool()`
`op_batch_normalization()`
`op_binary_crossentropy()`
`op_categorical_crossentropy()`
`op_conv()`
`op_conv_transpose()`
`op_ctc_loss()`
`op_depthwise_conv()`
`op_elu()`
`op_gelu()`
`op_hard_sigmoid()`
`op_hard_silu()`
`op_leaky_relu()`
`op_log_sigmoid()`
`op_log_softmax()`
`op_max_pool()`
`op_moments()`

op_multi_hot()
op_normalize()
op_one_hot()
op_relu()
op_relu6()
op_selu()
op_separable_conv()
op_sigmoid()
op_silu()
op_softmax()
op_softplus()
op_softsign()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()

op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()

op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()

op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()

op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

---

op_split *Split a tensor into chunks.*

Description

Split a tensor into chunks.

Usage

```
op_split(x, indices_or_sections, axis = 1L)
```

Arguments

x	Input tensor.
indices_or_sections	If an integer, N, the tensor will be split into N equal sections along axis. If a 1-D array of sorted integers, the entries indicate indices at which the tensor will be split along axis.
axis	Axis along which to split. Defaults to 1, the first axis.

Value

A list of tensors.

Note

A split does not have to result in equal division when using Torch backend.

See Also

- https://keras.io/api/ops/numpy#split-function

Other numpy ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_bincount()
op_broadcast_to()
op_ceil()
op_clip()
op_concatenate()
op_conj()
op_copy()
op_correlate()
op_cos()
op_cosh()

op_count_nonzero()
op_cross()
op_cumprod()
op_cumsum()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_einsum()
op_empty()
op_equal()
op_exp()
op_expand_dims()
op_expm1()
op_eye()
op_flip()
op_floor()
op_floor_divide()
op_full()
op_full_like()
op_get_item()
op_greater()
op_greater_equal()
op_hstack()
op_identity()
op_imag()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_matmul()
op_max()

op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moveaxis()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_not_equal()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_repeat()
op_reshape()
op_roll()
op_round()
op_sign()
op_sin()
op_sinh()
op_size()
op_sort()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_subtract()
op_sum()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_trace()

op_transpose()
op_tri()
op_tril()
op_triu()
op_var()
op_vdot()
op_vstack()
op_where()
op_zeros()
op_zeros_like()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()

op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()

op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()

op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_sqrt()

op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

---

op_sqrt *Return the non-negative square root of a tensor, element-wise.*

Description

Return the non-negative square root of a tensor, element-wise.

Usage

```
op_sqrt(x)
```

Arguments

x	Input tensor.

Value

Output tensor, the non-negative square root of x.

See Also

- https://keras.io/api/ops/numpy#sqrt-function

Other numpy ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_bincount()
op_broadcast_to()
op_ceil()
op_clip()
op_concatenate()
op_conj()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_cumprod()
op_cumsum()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_einsum()
op_empty()

op_equal()
op_exp()
op_expand_dims()
op_expm1()
op_eye()
op_flip()
op_floor()
op_floor_divide()
op_full()
op_full_like()
op_get_item()
op_greater()
op_greater_equal()
op_hstack()
op_identity()
op_imag()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_matmul()
op_max()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moveaxis()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()

op_not_equal()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_repeat()
op_reshape()
op_roll()
op_round()
op_sign()
op_sin()
op_sinh()
op_size()
op_sort()
op_split()
op_square()
op_squeeze()
op_stack()
op_std()
op_subtract()
op_sum()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_var()
op_vdot()
op_vstack()
op_where()
op_zeros()
op_zeros_like()

Other ops:
op_abs()

op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()

op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()

op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()

op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()

op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

---

op_square *Return the element-wise square of the input.*

Description

Return the element-wise square of the input.

Usage

```
op_square(x)
```

Arguments

x	Input tensor.

Value

Output tensor, the square of x.

See Also

- https://keras.io/api/ops/numpy#square-function

Other numpy ops:
op_abs()
op_add()
op_all()
op_any()

op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_bincount()
op_broadcast_to()
op_ceil()
op_clip()
op_concatenate()
op_conj()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_cumprod()
op_cumsum()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_einsum()
op_empty()
op_equal()
op_exp()
op_expand_dims()
op_expm1()
op_eye()
op_flip()
op_floor()
op_floor_divide()
op_full()
op_full_like()
op_get_item()

op_greater()
op_greater_equal()
op_hstack()
op_identity()
op_imag()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_matmul()
op_max()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moveaxis()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_not_equal()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_quantile()
op_ravel()
op_real()
op_reciprocal()

op_repeat()
op_reshape()
op_roll()
op_round()
op_sign()
op_sin()
op_sinh()
op_size()
op_sort()
op_split()
op_sqrt()
op_squeeze()
op_stack()
op_std()
op_subtract()
op_sum()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_var()
op_vdot()
op_vstack()
op_where()
op_zeros()
op_zeros_like()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()

op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()

op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()

op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()

op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()

op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

op_squeeze *Remove axes of length one from* x.

Description

Remove axes of length one from x.

Usage

```
op_squeeze(x, axis = NULL)
```

Arguments

x	Input tensor.
axis	Select a subset of the entries of length one in the shape.

Value

The input tensor with all or a subset of the dimensions of length 1 removed.

See Also

- https://keras.io/api/ops/numpy#squeeze-function

Other numpy ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()

op_argmin()
op_argsort()
op_array()
op_average()
op_bincount()
op_broadcast_to()
op_ceil()
op_clip()
op_concatenate()
op_conj()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_cumprod()
op_cumsum()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_einsum()
op_empty()
op_equal()
op_exp()
op_expand_dims()
op_expm1()
op_eye()
op_flip()
op_floor()
op_floor_divide()
op_full()
op_full_like()
op_get_item()
op_greater()
op_greater_equal()
op_hstack()
op_identity()
op_imag()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_less()

op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_matmul()
op_max()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moveaxis()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_not_equal()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_repeat()
op_reshape()
op_roll()
op_round()
op_sign()
op_sin()
op_sinh()
op_size()
op_sort()
op_split()

op_sqrt()
op_square()
op_stack()
op_std()
op_subtract()
op_sum()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_var()
op_vdot()
op_vstack()
op_where()
op_zeros()
op_zeros_like()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()

op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()

op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()

op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()

op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

op_stack *Join a sequence of tensors along a new axis.*

Description

The axis parameter specifies the index of the new axis in the dimensions of the result.

Usage

```
op_stack(x, axis = 1L)
```

Arguments

x	A sequence of tensors.
axis	Axis along which to stack. Defaults to 1, the first axis.

Value

The stacked tensor.

See Also

- https://keras.io/api/ops/numpy#stack-function

Other numpy ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_bincount()
op_broadcast_to()
op_ceil()
op_clip()
op_concatenate()

op_conj()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_cumprod()
op_cumsum()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_einsum()
op_empty()
op_equal()
op_exp()
op_expand_dims()
op_expm1()
op_eye()
op_flip()
op_floor()
op_floor_divide()
op_full()
op_full_like()
op_get_item()
op_greater()
op_greater_equal()
op_hstack()
op_identity()
op_imag()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_logaddexp()
op_logical_and()
op_logical_not()

op_logical_or()
op_logical_xor()
op_logspace()
op_matmul()
op_max()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moveaxis()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_not_equal()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_repeat()
op_reshape()
op_roll()
op_round()
op_sign()
op_sin()
op_sinh()
op_size()
op_sort()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_std()
op_subtract()
op_sum()
op_swapaxes()
op_take()
op_take_along_axis()

op_tan()
op_tanh()
op_tensordot()
op_tile()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_var()
op_vdot()
op_vstack()
op_where()
op_zeros()
op_zeros_like()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()

op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()

op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()

op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()

op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

---

op_std *Compute the standard deviation along the specified axis.*

Description

Compute the standard deviation along the specified axis.

Usage

```
op_std(x, axis = NULL, keepdims = FALSE)
```

Arguments

`x`	Input tensor.
`axis`	Axis along which to compute standard deviation. Default is to compute the standard deviation of the flattened tensor.
`keepdims`	If this is set to `TRUE`, the axes which are reduced are left in the result as dimensions with size one.

Value

Output tensor containing the standard deviation values.

See Also

- `https://keras.io/api/ops/numpy#std-function`

Other numpy ops:
`op_abs()`
`op_add()`
`op_all()`
`op_any()`
`op_append()`
`op_arange()`
`op_arccos()`
`op_arccosh()`
`op_arcsin()`
`op_arcsinh()`
`op_arctan()`
`op_arctan2()`
`op_arctanh()`
`op_argmax()`
`op_argmin()`
`op_argsort()`
`op_array()`
`op_average()`
`op_bincount()`
`op_broadcast_to()`
`op_ceil()`
`op_clip()`
`op_concatenate()`
`op_conj()`
`op_copy()`
`op_correlate()`
`op_cos()`
`op_cosh()`
`op_count_nonzero()`
`op_cross()`
`op_cumprod()`
`op_cumsum()`

op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_einsum()
op_empty()
op_equal()
op_exp()
op_expand_dims()
op_expm1()
op_eye()
op_flip()
op_floor()
op_floor_divide()
op_full()
op_full_like()
op_get_item()
op_greater()
op_greater_equal()
op_hstack()
op_identity()
op_imag()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_matmul()
op_max()
op_maximum()
op_mean()
op_median()
op_meshgrid()

op_min()
op_minimum()
op_mod()
op_moveaxis()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_not_equal()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_repeat()
op_reshape()
op_roll()
op_round()
op_sign()
op_sin()
op_sinh()
op_size()
op_sort()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_subtract()
op_sum()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()

op_var()
op_vdot()
op_vstack()
op_where()
op_zeros()
op_zeros_like()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()

op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()

op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()

op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()

op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

op_stft — *Short-Time Fourier Transform along the last axis of the input.*

Description

The STFT computes the Fourier transform of short overlapping windows of the input. This giving frequency components of the signal as they change over time.

Usage

```
op_stft(
  x,
  sequence_length,
  sequence_stride,
  fft_length,
  window = "hann",
  center = TRUE
)
```

Arguments

x	Input tensor.
sequence_length	An integer representing the sequence length.
sequence_stride	An integer representing the sequence hop size.
fft_length	An integer representing the size of the FFT to apply. If not specified, uses the smallest power of 2 enclosing sequence_length.
window	A string, a tensor of the window or NULL. If window is a string, available values are "hann" and "hamming". If window is a tensor, it will be used directly as the window and its length must be sequence_length. If window is NULL, no windowing is used. Defaults to "hann".
center	Whether to pad x on both sides so that the t-th sequence is centered at time t * sequence_stride. Otherwise, the t-th sequence begins at time t * sequence_stride. Defaults to TRUE.

Value

A list containing two tensors - the real and imaginary parts of the STFT output.

Examples

```
x <- op_array(c(0, 1, 2, 3, 4))
op_stft(x, 3, 2, 3)

## [[1]]
## tf.Tensor(
## [[ 0.  0.]
##  [ 2. -1.]
##  [ 4. -2.]], shape=(3, 2), dtype=float32)
##
## [[2]]
## tf.Tensor(
## [[ 0.         0.       ]
##  [ 0.        -1.7320508]
##  [ 0.        -3.4641016]], shape=(3, 2), dtype=float32)
```

See Also

- https://keras.io/api/ops/fft#stft-function

Other math ops:
op_erf()
op_erfinv()
op_extract_sequences()
op_fft()
op_fft2()

op_in_top_k()
op_irfft()
op_istft()
op_logsumexp()
op_qr()
op_rfft()
op_rsqrt()
op_segment_max()
op_segment_sum()
op_solve()
op_top_k()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()

op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()

op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()

op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()

op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

---

op_stop_gradient *Stops gradient computation.*

Description

Stops gradient computation.

Usage

```
op_stop_gradient(variable)
```

Arguments

variable	A tensor variable for which the gradient computation is to be disabled.

Value

The variable with gradient computation disabled.

Examples

```
var <- op_convert_to_tensor(c(1, 2, 3), dtype="float32")
var <- op_stop_gradient(var)
```

See Also

- https://keras.io/api/ops/core#stopgradient-function

Other core ops:
op_cast()
op_cond()
op_convert_to_numpy()
op_convert_to_tensor()
op_custom_gradient()
op_fori_loop()
op_is_tensor()
op_scatter()
op_scatter_update()
op_shape()
op_slice()
op_slice_update()
op_unstack()
op_vectorized_map()
op_while_loop()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()

op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()

op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()

op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()

op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

op_subtract	*Subtract arguments element-wise.*

Description

Note that this function is automatically called when using the R operator - with a tensor.

```
x <- op_ones(c(3))
op_subtract(x, x)

## tf.Tensor([0. 0. 0.], shape=(3), dtype=float32)
```

```
x - x

## tf.Tensor([0. 0. 0.], shape=(3), dtype=float32)
```

Usage

```
op_subtract(x1, x2)
```

Arguments

x1	First input tensor.
x2	Second input tensor.

Value

Output tensor, element-wise difference of x1 and x2.

See Also

- https://keras.io/api/ops/numpy#subtract-function

Other numpy ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()

op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_bincount()
op_broadcast_to()
op_ceil()
op_clip()
op_concatenate()
op_conj()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_cumprod()
op_cumsum()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_einsum()
op_empty()
op_equal()
op_exp()
op_expand_dims()
op_expm1()
op_eye()
op_flip()
op_floor()
op_floor_divide()
op_full()
op_full_like()
op_get_item()
op_greater()
op_greater_equal()
op_hstack()
op_identity()
op_imag()
op_isclose()

op_isfinite()
op_isinf()
op_isnan()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_matmul()
op_max()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moveaxis()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_not_equal()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_repeat()
op_reshape()
op_roll()
op_round()
op_sign()
op_sin()

op_sinh()
op_size()
op_sort()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_sum()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_var()
op_vdot()
op_vstack()
op_where()
op_zeros()
op_zeros_like()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()

op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()

op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()

op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()

op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()

op_zeros_like()

op_sum *Sum of a tensor over the given axes.*

Description

Sum of a tensor over the given axes.

Usage

```
op_sum(x, axis = NULL, keepdims = FALSE)
```

Arguments

x	Input tensor.
axis	Axis or axes along which the sum is computed. The default is to compute the sum of the flattened tensor.
keepdims	If this is set to TRUE, the axes which are reduced are left in the result as dimensions with size one.

Value

Output tensor containing the sum.

See Also

- https://keras.io/api/ops/numpy#sum-function

Other numpy ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()

op_average()
op_bincount()
op_broadcast_to()
op_ceil()
op_clip()
op_concatenate()
op_conj()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_cumprod()
op_cumsum()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_einsum()
op_empty()
op_equal()
op_exp()
op_expand_dims()
op_expm1()
op_eye()
op_flip()
op_floor()
op_floor_divide()
op_full()
op_full_like()
op_get_item()
op_greater()
op_greater_equal()
op_hstack()
op_identity()
op_imag()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_less()
op_less_equal()
op_linspace()
op_log()

op_log10()
op_log1p()
op_log2()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_matmul()
op_max()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moveaxis()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_not_equal()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_repeat()
op_reshape()
op_roll()
op_round()
op_sign()
op_sin()
op_sinh()
op_size()
op_sort()
op_split()
op_sqrt()
op_square()
op_squeeze()

op_stack()
op_std()
op_subtract()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_var()
op_vdot()
op_vstack()
op_where()
op_zeros()
op_zeros_like()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()

op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()

op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()

op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()

op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

---

op_svd *Computes the singular value decomposition of a matrix.*

Description

Computes the singular value decomposition of a matrix.

Usage

```
op_svd(x, full_matrices = TRUE, compute_uv = TRUE)
```

Arguments

`x`	Input tensor of shape `(..., M, N)`.
`full_matrices`	Logical
`compute_uv`	Logical

Value

A list of three tensors:

- a tensor of shape `(..., M, M)` containing the left singular vectors,
- a tensor of shape `(..., M, N)` containing the singular values and
- a tensor of shape `(..., N, N)` containing the right singular vectors.

See Also

- `https://www.tensorflow.org/api_docs/python/tf/keras/ops/svd`

Other linear algebra ops:
`op_cholesky()`
`op_det()`
`op_eig()`
`op_inv()`
`op_lu_factor()`
`op_norm()`
`op_solve_triangular()`

Other ops:
`op_abs()`
`op_add()`
`op_all()`
`op_any()`
`op_append()`
`op_arange()`
`op_arccos()`
`op_arccosh()`
`op_arcsin()`
`op_arcsinh()`
`op_arctan()`
`op_arctan2()`
`op_arctanh()`

op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()

op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()

op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()

op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()

op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

---

op_swapaxes *Interchange two axes of a tensor.*

Description

Interchange two axes of a tensor.

Usage

```
op_swapaxes(x, axis1, axis2)
```

Arguments

x	Input tensor.
axis1	First axis.
axis2	Second axis.

Value

A tensor with the axes swapped.

See Also

- https://keras.io/api/ops/numpy#swapaxes-function

Other numpy ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()

op_argmin()
op_argsort()
op_array()
op_average()
op_bincount()
op_broadcast_to()
op_ceil()
op_clip()
op_concatenate()
op_conj()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_cumprod()
op_cumsum()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_einsum()
op_empty()
op_equal()
op_exp()
op_expand_dims()
op_expm1()
op_eye()
op_flip()
op_floor()
op_floor_divide()
op_full()
op_full_like()
op_get_item()
op_greater()
op_greater_equal()
op_hstack()
op_identity()
op_imag()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_less()

op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_matmul()
op_max()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moveaxis()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_not_equal()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_repeat()
op_reshape()
op_roll()
op_round()
op_sign()
op_sin()
op_sinh()
op_size()
op_sort()
op_split()

op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_subtract()
op_sum()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_var()
op_vdot()
op_vstack()
op_where()
op_zeros()
op_zeros_like()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()

op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()

op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()

op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()

op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

op_take	*Take elements from a tensor along an axis.*

Description

Take elements from a tensor along an axis.

Usage

```
op_take(x, indices, axis = NULL)
```

Arguments

`x`	Source tensor.
`indices`	The indices of the values to extract.
`axis`	The axis over which to select values. By default, the flattened input tensor is used.

Value

The corresponding tensor of values.

See Also

- https://keras.io/api/ops/numpy#take-function

Other numpy ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_bincount()
op_broadcast_to()

op_ceil()
op_clip()
op_concatenate()
op_conj()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_cumprod()
op_cumsum()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_einsum()
op_empty()
op_equal()
op_exp()
op_expand_dims()
op_expm1()
op_eye()
op_flip()
op_floor()
op_floor_divide()
op_full()
op_full_like()
op_get_item()
op_greater()
op_greater_equal()
op_hstack()
op_identity()
op_imag()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()

op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_matmul()
op_max()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moveaxis()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_not_equal()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_repeat()
op_reshape()
op_roll()
op_round()
op_sign()
op_sin()
op_sinh()
op_size()
op_sort()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_subtract()

op_sum()
op_swapaxes()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_var()
op_vdot()
op_vstack()
op_where()
op_zeros()
op_zeros_like()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()

op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()

op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()

op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()

op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

---

op_take_along_axis *Select values from* x *at the 1-D* indices *along the given axis.*

Description

Select values from x at the 1-D indices along the given axis.

Usage

```
op_take_along_axis(x, indices, axis = NULL)
```

Arguments

`x`	Source tensor.
`indices`	The indices of the values to extract.
`axis`	The axis over which to select values. By default, the flattened input tensor is used.

Value

The corresponding tensor of values.

See Also

- https://keras.io/api/ops/numpy#takealongaxis-function

Other numpy ops:
`op_abs()`
`op_add()`
`op_all()`
`op_any()`
`op_append()`
`op_arange()`
`op_arccos()`
`op_arccosh()`
`op_arcsin()`
`op_arcsinh()`
`op_arctan()`
`op_arctan2()`
`op_arctanh()`
`op_argmax()`
`op_argmin()`
`op_argsort()`
`op_array()`
`op_average()`
`op_bincount()`
`op_broadcast_to()`
`op_ceil()`
`op_clip()`
`op_concatenate()`
`op_conj()`
`op_copy()`
`op_correlate()`
`op_cos()`
`op_cosh()`
`op_count_nonzero()`
`op_cross()`

op_cumprod()
op_cumsum()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_einsum()
op_empty()
op_equal()
op_exp()
op_expand_dims()
op_expm1()
op_eye()
op_flip()
op_floor()
op_floor_divide()
op_full()
op_full_like()
op_get_item()
op_greater()
op_greater_equal()
op_hstack()
op_identity()
op_imag()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_matmul()
op_max()
op_maximum()
op_mean()

op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moveaxis()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_not_equal()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_repeat()
op_reshape()
op_roll()
op_round()
op_sign()
op_sin()
op_sinh()
op_size()
op_sort()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_subtract()
op_sum()
op_swapaxes()
op_take()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_trace()
op_transpose()
op_tri()

op_tril()
op_triu()
op_var()
op_vdot()
op_vstack()
op_where()
op_zeros()
op_zeros_like()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()

op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()

op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()

op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()

op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

---

op_tan *Compute tangent, element-wise.*

Description

Compute tangent, element-wise.

Usage

```
op_tan(x)
```

Arguments

x	Input tensor.

Value

Output tensor of same shape as x.

See Also

- https://keras.io/api/ops/numpy#tan-function

Other numpy ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_bincount()
op_broadcast_to()
op_ceil()
op_clip()
op_concatenate()
op_conj()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_cumprod()
op_cumsum()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_einsum()
op_empty()

op_equal()
op_exp()
op_expand_dims()
op_expm1()
op_eye()
op_flip()
op_floor()
op_floor_divide()
op_full()
op_full_like()
op_get_item()
op_greater()
op_greater_equal()
op_hstack()
op_identity()
op_imag()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_matmul()
op_max()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moveaxis()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()

op_not_equal()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_repeat()
op_reshape()
op_roll()
op_round()
op_sign()
op_sin()
op_sinh()
op_size()
op_sort()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_subtract()
op_sum()
op_swapaxes()
op_take()
op_take_along_axis()
op_tanh()
op_tensordot()
op_tile()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_var()
op_vdot()
op_vstack()
op_where()
op_zeros()
op_zeros_like()

Other ops:
op_abs()

op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()

op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()

op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()

op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()

op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

op_tanh *Hyperbolic tangent, element-wise.*

Description

Hyperbolic tangent, element-wise.

Usage

```
op_tanh(x)
```

Arguments

x	Input tensor.

Value

Output tensor of same shape as x.

See Also

- https://keras.io/api/ops/nn#tanh-function

Other numpy ops:
op_abs()
op_add()
op_all()
op_any()

op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_bincount()
op_broadcast_to()
op_ceil()
op_clip()
op_concatenate()
op_conj()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_cumprod()
op_cumsum()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_einsum()
op_empty()
op_equal()
op_exp()
op_expand_dims()
op_expm1()
op_eye()
op_flip()
op_floor()
op_floor_divide()
op_full()
op_full_like()
op_get_item()

op_greater()
op_greater_equal()
op_hstack()
op_identity()
op_imag()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_matmul()
op_max()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moveaxis()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_not_equal()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_quantile()
op_ravel()
op_real()
op_reciprocal()

op_repeat()
op_reshape()
op_roll()
op_round()
op_sign()
op_sin()
op_sinh()
op_size()
op_sort()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_subtract()
op_sum()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tensordot()
op_tile()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_var()
op_vdot()
op_vstack()
op_where()
op_zeros()
op_zeros_like()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()

op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()

op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()

op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()

op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()

op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

---

op_tensordot *Compute the tensor dot product along specified axes.*

Description

Compute the tensor dot product along specified axes.

Usage

```
op_tensordot(x1, x2, axes = 3L)
```

Arguments

x1	First tensor.
x2	Second tensor.
axes	• If an integer, N, sum over the last N axes of x1 and the first N axes of x2 in order. The sizes of the corresponding axes must match. • Or, a list of axes to be summed over, first sequence applying to x1, second to x2. Both sequences must be of the same length.

Value

The tensor dot product of the inputs.

See Also

- https://keras.io/api/ops/numpy#tensordot-function

Other numpy ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()

op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_bincount()
op_broadcast_to()
op_ceil()
op_clip()
op_concatenate()
op_conj()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_cumprod()
op_cumsum()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_einsum()
op_empty()
op_equal()
op_exp()
op_expand_dims()
op_expm1()
op_eye()
op_flip()
op_floor()
op_floor_divide()
op_full()
op_full_like()
op_get_item()
op_greater()
op_greater_equal()
op_hstack()
op_identity()
op_imag()

op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_matmul()
op_max()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moveaxis()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_not_equal()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_repeat()
op_reshape()
op_roll()
op_round()
op_sign()

op_sin()
op_sinh()
op_size()
op_sort()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_subtract()
op_sum()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tile()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_var()
op_vdot()
op_vstack()
op_where()
op_zeros()
op_zeros_like()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()

op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()

op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()

op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()

op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()

op_zeros()
op_zeros_like()

op_tile — *Repeat* x *the number of times given by* repeats.

Description

If repeats has length d, the result will have dimension of max(d, x.ndim).

If x.ndim < d, x is promoted to be d-dimensional by prepending new axes.

If x.ndim > d, repeats is promoted to x.ndim by prepending 1's to it.

Usage

```
op_tile(x, repeats)
```

Arguments

x	Input tensor.
repeats	The number of repetitions of x along each axis.

Value

The tiled output tensor.

See Also

- https://keras.io/api/ops/numpy#tile-function

Other numpy ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()

op_array()
op_average()
op_bincount()
op_broadcast_to()
op_ceil()
op_clip()
op_concatenate()
op_conj()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_cumprod()
op_cumsum()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_einsum()
op_empty()
op_equal()
op_exp()
op_expand_dims()
op_expm1()
op_eye()
op_flip()
op_floor()
op_floor_divide()
op_full()
op_full_like()
op_get_item()
op_greater()
op_greater_equal()
op_hstack()
op_identity()
op_imag()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_less()
op_less_equal()
op_linspace()

op_log()
op_log10()
op_log1p()
op_log2()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_matmul()
op_max()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moveaxis()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_not_equal()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_repeat()
op_reshape()
op_roll()
op_round()
op_sign()
op_sin()
op_sinh()
op_size()
op_sort()
op_split()
op_sqrt()
op_square()

op_squeeze()
op_stack()
op_std()
op_subtract()
op_sum()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_var()
op_vdot()
op_vstack()
op_where()
op_zeros()
op_zeros_like()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()

op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()

op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()

op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()

op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

op_top_k *Finds the top-k values and their indices in a tensor.*

Description

Finds the top-k values and their indices in a tensor.

Usage

```
op_top_k(x, k, sorted = TRUE)
```

Arguments

`x`	Input tensor.
`k`	An integer representing the number of top elements to retrieve.
`sorted`	A boolean indicating whether to sort the output in descending order. Defaults to`TRUE`.

Value

A list containing two tensors. The first tensor contains the top-k values, and the second tensor contains the indices of the top-k values in the input tensor.

Examples

```
x <- op_array(c(5, 2, 7, 1, 9, 3), "int32")
op_top_k(x, k = 3)
```

```
## $values
## tf.Tensor([9 7 5], shape=(3), dtype=int32)
##
## $indices
## tf.Tensor([4 2 0], shape=(3), dtype=int32)
```

```
c(values, indices) %<-% op_top_k(x, k = 3)
values
```

```
## tf.Tensor([9 7 5], shape=(3), dtype=int32)
```

```
indices
```

```
## tf.Tensor([4 2 0], shape=(3), dtype=int32)
```

See Also

- https://keras.io/api/ops/core#topk-function

Other math ops:
op_erf()
op_erfinv()
op_extract_sequences()
op_fft()
op_fft2()
op_in_top_k()
op_irfft()
op_istft()
op_logsumexp()
op_qr()
op_rfft()
op_rsqrt()
op_segment_max()
op_segment_sum()
op_solve()
op_stft()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()

op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()

op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()

op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()

op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

op_trace *Return the sum along diagonals of the tensor.*

Description

If `x` is 2-D, the sum along its diagonal with the given offset is returned, i.e., the sum of elements `x[i, i+offset]` for all `i`.

If a has more than two dimensions, then the axes specified by `axis1` and `axis2` are used to determine the 2-D sub-arrays whose traces are returned.

The shape of the resulting tensor is the same as that of `x` with `axis1` and `axis2` removed.

Usage

```
op_trace(x, offset = 0L, axis1 = 1L, axis2 = 2L)
```

Arguments

`x`	Input tensor.
`offset`	Offset of the diagonal from the main diagonal. Can be both positive and negative. Defaults to `0`.
`axis1`	Axis to be used as the first axis of the 2-D sub-arrays. Defaults to `1`. (first axis).
`axis2`	Axis to be used as the second axis of the 2-D sub-arrays. Defaults to `2`. (second axis).

Value

If `x` is 2-D, the sum of the diagonal is returned. If `x` has larger dimensions, then a tensor of sums along diagonals is returned.

See Also

- `https://keras.io/api/ops/numpy#trace-function`

Other numpy ops:
`op_abs()`
`op_add()`
`op_all()`
`op_any()`
`op_append()`
`op_arange()`
`op_arccos()`
`op_arccosh()`
`op_arcsin()`
`op_arcsinh()`
`op_arctan()`
`op_arctan2()`
`op_arctanh()`
`op_argmax()`
`op_argmin()`
`op_argsort()`
`op_array()`
`op_average()`

op_bincount()
op_broadcast_to()
op_ceil()
op_clip()
op_concatenate()
op_conj()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_cumprod()
op_cumsum()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_einsum()
op_empty()
op_equal()
op_exp()
op_expand_dims()
op_expm1()
op_eye()
op_flip()
op_floor()
op_floor_divide()
op_full()
op_full_like()
op_get_item()
op_greater()
op_greater_equal()
op_hstack()
op_identity()
op_imag()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()

op_log1p()
op_log2()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_matmul()
op_max()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moveaxis()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_not_equal()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_repeat()
op_reshape()
op_roll()
op_round()
op_sign()
op_sin()
op_sinh()
op_size()
op_sort()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()

op_std()
op_subtract()
op_sum()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_transpose()
op_tri()
op_tril()
op_triu()
op_var()
op_vdot()
op_vstack()
op_where()
op_zeros()
op_zeros_like()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()

op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()

op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()

op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()

op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

op_transpose *Returns a tensor with* axes *transposed.*

Description

Returns a tensor with `axes` transposed.

Usage

```
op_transpose(x, axes = NULL)
```

Arguments

`x`	Input tensor.
`axes`	Sequence of integers. Permutation of the dimensions of `x`. By default, the order of the axes are reversed.

Value

`x` with its axes permuted.

See Also

- `https://keras.io/api/ops/numpy#transpose-function`

Other numpy ops:
`op_abs()`
`op_add()`
`op_all()`
`op_any()`
`op_append()`
`op_arange()`
`op_arccos()`
`op_arccosh()`
`op_arcsin()`
`op_arcsinh()`
`op_arctan()`
`op_arctan2()`
`op_arctanh()`
`op_argmax()`
`op_argmin()`
`op_argsort()`
`op_array()`
`op_average()`
`op_bincount()`
`op_broadcast_to()`
`op_ceil()`
`op_clip()`
`op_concatenate()`
`op_conj()`
`op_copy()`
`op_correlate()`
`op_cos()`

op_cosh()
op_count_nonzero()
op_cross()
op_cumprod()
op_cumsum()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_einsum()
op_empty()
op_equal()
op_exp()
op_expand_dims()
op_expm1()
op_eye()
op_flip()
op_floor()
op_floor_divide()
op_full()
op_full_like()
op_get_item()
op_greater()
op_greater_equal()
op_hstack()
op_identity()
op_imag()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_matmul()

op_max()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moveaxis()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_not_equal()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_repeat()
op_reshape()
op_roll()
op_round()
op_sign()
op_sin()
op_sinh()
op_size()
op_sort()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_subtract()
op_sum()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()

op_tile()
op_trace()
op_tri()
op_tril()
op_triu()
op_var()
op_vdot()
op_vstack()
op_where()
op_zeros()
op_zeros_like()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()

op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()

op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()

op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()

op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

---

op_tri *Return a tensor with ones at and below a diagonal and zeros elsewhere.*

Description

Return a tensor with ones at and below a diagonal and zeros elsewhere.

Usage

```
op_tri(N, M = NULL, k = 0L, dtype = NULL)
```

Arguments

N	Number of rows in the tensor.
M	Number of columns in the tensor.
k	The sub-diagonal at and below which the array is filled. k = 0 is the main diagonal, while k < 0 is below it, and k > 0 is above. The default is 0.
dtype	Data type of the returned tensor. The default is "float32".

Value

Tensor with its lower triangle filled with ones and zeros elsewhere. T[i, j] == 1 for j <= i + k, 0 otherwise.

See Also

- https://keras.io/api/ops/numpy#tri-function

Other numpy ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_bincount()
op_broadcast_to()
op_ceil()
op_clip()
op_concatenate()
op_conj()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_cumprod()

op_cumsum()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_einsum()
op_empty()
op_equal()
op_exp()
op_expand_dims()
op_expm1()
op_eye()
op_flip()
op_floor()
op_floor_divide()
op_full()
op_full_like()
op_get_item()
op_greater()
op_greater_equal()
op_hstack()
op_identity()
op_imag()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_matmul()
op_max()
op_maximum()
op_mean()
op_median()

op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moveaxis()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_not_equal()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_repeat()
op_reshape()
op_roll()
op_round()
op_sign()
op_sin()
op_sinh()
op_size()
op_sort()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_subtract()
op_sum()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_trace()
op_transpose()
op_tril()

op_triu()
op_var()
op_vdot()
op_vstack()
op_where()
op_zeros()
op_zeros_like()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()

op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()

op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()

op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()

op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

op_tril *Return lower triangle of a tensor.*

Description

For tensors with `ndim` exceeding 2, `tril` will apply to the final two axes.

Usage

```
op_tril(x, k = 0L)
```

Arguments

x	Input tensor.
k	Diagonal above which to zero elements. Defaults to `0`. the main diagonal. k < `0` is below it, and k > `0` is above it.

Value

Lower triangle of x, of same shape and data type as x.

See Also

- https://keras.io/api/ops/numpy#tril-function

Other numpy ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_bincount()
op_broadcast_to()
op_ceil()
op_clip()
op_concatenate()
op_conj()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_cumprod()
op_cumsum()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_einsum()
op_empty()

op_equal()
op_exp()
op_expand_dims()
op_expm1()
op_eye()
op_flip()
op_floor()
op_floor_divide()
op_full()
op_full_like()
op_get_item()
op_greater()
op_greater_equal()
op_hstack()
op_identity()
op_imag()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_matmul()
op_max()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moveaxis()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()

op_not_equal()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_repeat()
op_reshape()
op_roll()
op_round()
op_sign()
op_sin()
op_sinh()
op_size()
op_sort()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_subtract()
op_sum()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_trace()
op_transpose()
op_tri()
op_triu()
op_var()
op_vdot()
op_vstack()
op_where()
op_zeros()
op_zeros_like()

Other ops:
op_abs()

op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()

op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()

op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()

op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()

op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

---

op_triu *Return upper triangle of a tensor.*

Description

For tensors with ndim exceeding 2, triu will apply to the final two axes.

Usage

```
op_triu(x, k = 0L)
```

Arguments

x	Input tensor.
k	Diagonal below which to zero elements. Defaults to 0. the main diagonal. k < 0 is below it, and k > 0 is above it.

Value

Upper triangle of x, of same shape and data type as x.

See Also

- https://keras.io/api/ops/numpy#triu-function

Other numpy ops:
op_abs()
op_add()

op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_bincount()
op_broadcast_to()
op_ceil()
op_clip()
op_concatenate()
op_conj()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_cumprod()
op_cumsum()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_einsum()
op_empty()
op_equal()
op_exp()
op_expand_dims()
op_expm1()
op_eye()
op_flip()
op_floor()
op_floor_divide()
op_full()

op_full_like()
op_get_item()
op_greater()
op_greater_equal()
op_hstack()
op_identity()
op_imag()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_matmul()
op_max()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moveaxis()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_not_equal()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_quantile()
op_ravel()

op_real()
op_reciprocal()
op_repeat()
op_reshape()
op_roll()
op_round()
op_sign()
op_sin()
op_sinh()
op_size()
op_sort()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_subtract()
op_sum()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_trace()
op_transpose()
op_tri()
op_tril()
op_var()
op_vdot()
op_vstack()
op_where()
op_zeros()
op_zeros_like()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()

op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()

op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()

op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()

op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()

op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

---

op_unstack *Unpacks the given dimension of a rank-R tensor into rank-(R-1) tensors.*

Description

Unpacks the given dimension of a rank-R tensor into rank-(R-1) tensors.

Usage

```
op_unstack(x, num = NULL, axis = 1L)
```

Arguments

x	The input tensor.
num	The length of the dimension axis. Automatically inferred if NULL.
axis	The axis along which to unpack.

Value

A list of tensors unpacked along the given axis.

Examples

```
x <- op_array(rbind(c(1, 2),
                    c(3, 4)))
op_unstack(x, axis=1)

## [[1]]
## tf.Tensor([1. 2.], shape=(2), dtype=float64)
##
## [[2]]
## tf.Tensor([3. 4.], shape=(2), dtype=float64)


op_unstack(x, axis=2)
```

```
## [[1]]
## tf.Tensor([1. 3.], shape=(2), dtype=float64)
##
## [[2]]
## tf.Tensor([2. 4.], shape=(2), dtype=float64)
```

```
all.equal(op_unstack(x),
          op_unstack(x, axis = 1))
```

```
## [1] TRUE
```

```
all.equal(op_unstack(x, axis = -1),
          op_unstack(x, axis = 2))
```

```
## [1] TRUE
```

```
# [array([1, 2)), array([3, 4))]
```

[3, 4))]: R:3,%204))

See Also

Other core ops:
op_cast()
op_cond()
op_convert_to_numpy()
op_convert_to_tensor()
op_custom_gradient()
op_fori_loop()
op_is_tensor()
op_scatter()
op_scatter_update()
op_shape()
op_slice()
op_slice_update()
op_stop_gradient()
op_vectorized_map()
op_while_loop()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()

op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()

op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()

op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()

op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()

op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

op_var — *Compute the variance along the specified axes.*

Description

Compute the variance along the specified axes.

Usage

```
op_var(x, axis = NULL, keepdims = FALSE)
```

Arguments

x	Input tensor.
axis	Axis or axes along which the variance is computed. The default is to compute the variance of the flattened tensor.
keepdims	If this is set to TRUE, the axes which are reduced are left in the result as dimensions with size one.

Value

Output tensor containing the variance.

See Also

- https://keras.io/api/ops/numpy#var-function

Other numpy ops:
op_abs()
op_add()
op_all()
op_any()

op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_bincount()
op_broadcast_to()
op_ceil()
op_clip()
op_concatenate()
op_conj()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_cumprod()
op_cumsum()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_einsum()
op_empty()
op_equal()
op_exp()
op_expand_dims()
op_expm1()
op_eye()
op_flip()
op_floor()
op_floor_divide()
op_full()
op_full_like()
op_get_item()

op_greater()
op_greater_equal()
op_hstack()
op_identity()
op_imag()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_matmul()
op_max()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moveaxis()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_not_equal()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_quantile()
op_ravel()
op_real()
op_reciprocal()

op_repeat()
op_reshape()
op_roll()
op_round()
op_sign()
op_sin()
op_sinh()
op_size()
op_sort()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_subtract()
op_sum()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_vdot()
op_vstack()
op_where()
op_zeros()
op_zeros_like()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()

op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()

op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()

op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()

op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()

op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

---

op_vdot *Return the dot product of two vectors.*

Description

If the first argument is complex, the complex conjugate of the first argument is used for the calculation of the dot product.

Multidimensional tensors are flattened before the dot product is taken.

Usage

```
op_vdot(x1, x2)
```

Arguments

x1	First input tensor. If complex, its complex conjugate is taken before calculation of the dot product.
x2	Second input tensor.

Value

Output tensor.

See Also

- https://keras.io/api/ops/numpy#vdot-function

Other numpy ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()

op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_bincount()
op_broadcast_to()
op_ceil()
op_clip()
op_concatenate()
op_conj()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_cumprod()
op_cumsum()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_einsum()
op_empty()
op_equal()
op_exp()
op_expand_dims()
op_expm1()
op_eye()
op_flip()
op_floor()
op_floor_divide()
op_full()
op_full_like()
op_get_item()
op_greater()
op_greater_equal()
op_hstack()
op_identity()
op_imag()
op_isclose()
op_isfinite()

op_isinf()
op_isnan()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_matmul()
op_max()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moveaxis()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_not_equal()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_repeat()
op_reshape()
op_roll()
op_round()
op_sign()
op_sin()
op_sinh()

op_size()
op_sort()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_subtract()
op_sum()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_var()
op_vstack()
op_where()
op_zeros()
op_zeros_like()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()

op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()

op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()

op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()

op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

`op_vectorized_map` *Parallel map of function `f` on the first axis of tensor(s) `elements`.*

Description

Schematically, `op_vectorized_map()` maps over the first dimension of the provided tensors. If `elements` is a list of tensors, then each of the tensors are required to have the same size first dimension, and they are iterated over together.

Usage

```
op_vectorized_map(elements, f)
```

Arguments

`elements`	see description
`f`	A function taking either a tensor, or list of tensors.

Value

A tensor or list of tensors, the result of mapping `f` across `elements`.

Examples

```
(x <- op_arange(12L) |> op_reshape(c(3, 4)))

## tf.Tensor(
## [[ 0  1  2  3]
##  [ 4  5  6  7]
##  [ 8  9 10 11]], shape=(3, 4), dtype=int32)


x |> op_vectorized_map(\(row) {row + 10})

## tf.Tensor(
## [[10 11 12 13]
##  [14 15 16 17]
##  [18 19 20 21]], shape=(3, 4), dtype=int32)


list(x, x, x) |> op_vectorized_map(\(rows) Reduce(`+`, rows))

## tf.Tensor(
## [[ 0  3  6  9]
##  [12 15 18 21]
##  [24 27 30 33]], shape=(3, 4), dtype=int32)
```

Note that f may be traced and compiled. Meaning, the R function may only evaluated once with symbolic tensors if using Jax or TensorFlow backends, and not with eager tensors. See the output from `str()` in these examples:

```
# simplest case, map f over rows of x,
# where .x is 1 row of x
input <- x
output <- op_vectorized_map(input, function(.x) {
  str(.x)
  .x + 10
})

## <tf.Tensor 'loop_body/GatherV2:0' shape=(4) dtype=int32>


output

## tf.Tensor(
## [[10 11 12 13]
##  [14 15 16 17]
##  [18 19 20 21]], shape=(3, 4), dtype=int32)


# map f over two tensors simultaneously. Here, # `.x` is a list of two
# tensors. The return values from each call of `f(row)` are stacked to form the
# final output
input <- list(x, x)
output <- op_vectorized_map(input, function(.x) {
  str(.x)
  .x[[1]] + 10
})

## List of 2
##  $ :<tf.Tensor 'loop_body/GatherV2:0' shape=(4) dtype=int32>
##  $ :<tf.Tensor 'loop_body/GatherV2_1:0' shape=(4) dtype=int32>


output

## tf.Tensor(
## [[10 11 12 13]
##  [14 15 16 17]
##  [18 19 20 21]], shape=(3, 4), dtype=int32)


# same as above, but now returning two tensors in the final output
output <- op_vectorized_map(input, function(.x) {
  str(.x)
  c(.x1, .x2) %<-% .x
  list(.x1+10, .x2+20)
})

## List of 2
##  $ :<tf.Tensor 'loop_body/GatherV2:0' shape=(4) dtype=int32>
##  $ :<tf.Tensor 'loop_body/GatherV2_1:0' shape=(4) dtype=int32>


output

## [[1]]
## tf.Tensor(
## [[10 11 12 13]
##  [14 15 16 17]
##  [18 19 20 21]], shape=(3, 4), dtype=int32)
##
## [[2]]
## tf.Tensor(
## [[20 21 22 23]
##  [24 25 26 27]
##  [28 29 30 31]], shape=(3, 4), dtype=int32)


# passing named lists.
# WARNING: if passing a named list, the order of elements of `.x` supplied
# to `f` is not stable. Only retrieve elements by name.
input <- list(name1 = x, name2 = x)
output <- op_vectorized_map(input, function(.x) {
  str(.x)
  list(outname1 = .x$name1 + 10,
       outname2 = .x$name2 + 20)
})

## List of 2
##  $ name1:<tf.Tensor 'loop_body/GatherV2:0' shape=(4) dtype=int32>
##  $ name2:<tf.Tensor 'loop_body/GatherV2_1:0' shape=(4) dtype=int32>


output

## $outname1
## tf.Tensor(
## [[10 11 12 13]
##  [14 15 16 17]
##  [18 19 20 21]], shape=(3, 4), dtype=int32)
##
## $outname2
## tf.Tensor(
## [[20 21 22 23]
##  [24 25 26 27]
##  [28 29 30 31]], shape=(3, 4), dtype=int32)
```

```
# passing a tuple() is equivalent to passing an unnamed list()
input <- tuple(x, x)
output <- op_vectorized_map(input, function(.x) {
  str(.x)
  list(.x[[1]] + 10)
})
```

```
## List of 2
##  $ :<tf.Tensor 'loop_body/GatherV2:0' shape=(4) dtype=int32>
##  $ :<tf.Tensor 'loop_body/GatherV2_1:0' shape=(4) dtype=int32>
```

```
output
```

```
## [[1]]
## tf.Tensor(
## [[10 11 12 13]
##  [14 15 16 17]
##  [18 19 20 21]], shape=(3, 4), dtype=int32)
```

Debugging `f`

Even in eager contexts, `op_vectorized_map()` may trace `f`. In that case, if you want to eagerly debug `f` (e.g., with `browser()`), you can swap in a manual (slow) implementation of `op_vectorized_map()`. Note this example debug implementation does not handle all the same edge cases as `op_vectorized_map()`, in particular, if `f` returns a structure of multiple tensors.

```
op_vectorized_map_debug <- function(elements, fn) {

  if (!is.list(elements)) {
    # `elements` is a single tensor
    batch_size <- op_shape(elements)[[1]]
    out <- elements |>
      op_split(batch_size) |>
      lapply(fn) |>
      op_stack()
    return(out)
  }

  # `elements` is a list of tensors
  batch_size <- elements[[1]] |> op_shape() |> _[[1]]
  elements |>
    lapply(\(e) op_split(e, batch_size)) |>
    zip_lists() |>
    lapply(fn) |>
    op_stack()

}
```

See Also

Other core ops:
op_cast()
op_cond()
op_convert_to_numpy()
op_convert_to_tensor()
op_custom_gradient()
op_fori_loop()
op_is_tensor()
op_scatter()
op_scatter_update()
op_shape()
op_slice()
op_slice_update()
op_stop_gradient()
op_unstack()
op_while_loop()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()

op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()

op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()

op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()

op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vstack()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

op_vstack	*Stack tensors in sequence vertically (row wise).*

Description

Stack tensors in sequence vertically (row wise).

Usage

```
op_vstack(xs)
```

Arguments

xs	Sequence of tensors.

Value

Tensor formed by stacking the given tensors.

See Also

- https://keras.io/api/ops/numpy#vstack-function

Other numpy ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_bincount()
op_broadcast_to()
op_ceil()
op_clip()
op_concatenate()
op_conj()

op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_cumprod()
op_cumsum()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_einsum()
op_empty()
op_equal()
op_exp()
op_expand_dims()
op_expm1()
op_eye()
op_flip()
op_floor()
op_floor_divide()
op_full()
op_full_like()
op_get_item()
op_greater()
op_greater_equal()
op_hstack()
op_identity()
op_imag()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()

op_logical_xor()
op_logspace()
op_matmul()
op_max()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moveaxis()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_not_equal()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_repeat()
op_reshape()
op_roll()
op_round()
op_sign()
op_sin()
op_sinh()
op_size()
op_sort()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_subtract()
op_sum()
op_swapaxes()
op_take()
op_take_along_axis()

op_tan()
op_tanh()
op_tensordot()
op_tile()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_var()
op_vdot()
op_where()
op_zeros()
op_zeros_like()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()

op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()

op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()

op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()

op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_where()
op_while_loop()
op_zeros()
op_zeros_like()

op_where *Return elements chosen from* x1 *or* x2 *depending on* condition.

Description

Return elements chosen from x1 or x2 depending on condition.

Usage

```
op_where(condition, x1 = NULL, x2 = NULL)
```

Arguments

condition	Where TRUE, yield x1, otherwise yield x2.
x1	Values from which to choose when condition is TRUE.
x2	Values from which to choose when condition is FALSE.

Value

A tensor with elements from x1 where condition is TRUE, and elements from x2 where condition is FALSE.

See Also

- https://keras.io/api/ops/numpy#where-function

Other numpy ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_bincount()
op_broadcast_to()
op_ceil()
op_clip()
op_concatenate()
op_conj()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_cumprod()
op_cumsum()
op_diag()

op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_einsum()
op_empty()
op_equal()
op_exp()
op_expand_dims()
op_expm1()
op_eye()
op_flip()
op_floor()
op_floor_divide()
op_full()
op_full_like()
op_get_item()
op_greater()
op_greater_equal()
op_hstack()
op_identity()
op_imag()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_matmul()
op_max()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()

op_minimum()
op_mod()
op_moveaxis()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_not_equal()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_repeat()
op_reshape()
op_roll()
op_round()
op_sign()
op_sin()
op_sinh()
op_size()
op_sort()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_subtract()
op_sum()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()

op_var()
op_vdot()
op_vstack()
op_zeros()
op_zeros_like()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()

op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()

op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()

op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()

op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_while_loop()
op_zeros()
op_zeros_like()

---

op_while_loop *While loop implementation.*

Description

While loop implementation.

Usage

```
op_while_loop(cond, body, loop_vars, maximum_iterations = NULL)
```

Arguments

cond	A callable that represents the termination condition of the loop. Must accept a `loop_vars` like structure as an argument. If `loop_vars` is a tuple or unnamed list, each element of `loop_vars` will be passed positionally to the callable.
body	A callable that represents the loop body. Must accept a `loop_vars` like structure as an argument, and return update value with the same structure. If `loop_vars` is a tuple or unnamed list, each element of `loop_vars` will be passed positionally to the callable.

`loop_vars`	An arbitrary nested structure of tensor state to persist across loop iterations.
`maximum_iterations`	Optional maximum number of iterations of the while loop to run. If provided, the cond output is AND-ed with an additional condition ensuring the number of iterations executed is no greater than `maximum_iterations`.

Value

A list of tensors, has the same shape and dtype as `loop_vars`.

Examples

```
i <- 0
loop_vars <- list(i)

# cond() must return a scalar bool
cond <- function(i) i < 10L

# body must return same shape as loop_vars
body <- function(i) list(i + 1L)

op_while_loop(cond, body, loop_vars)

## [[1]]
## tf.Tensor(10.0, shape=(), dtype=float32)


x <- 0; y <- 1
cond <- \(x, y) x < 10
body <- \(x, y) list(x+1, y+1)
op_while_loop(cond, body, list(x, y))

## [[1]]
## tf.Tensor(10.0, shape=(), dtype=float32)
##
## [[2]]
## tf.Tensor(11.0, shape=(), dtype=float32)
```

See Also

- `https://keras.io/api/ops/core#whileloop-function`

Other core ops:
`op_cast()`
`op_cond()`
`op_convert_to_numpy()`
`op_convert_to_tensor()`
`op_custom_gradient()`

op_fori_loop()
op_is_tensor()
op_scatter()
op_scatter_update()
op_shape()
op_slice()
op_slice_update()
op_stop_gradient()
op_unstack()
op_vectorized_map()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()

op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()

op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()

op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()

op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_zeros()
op_zeros_like()

---

op_zeros *Return a new tensor of given shape and type, filled with zeros.*

Description

Return a new tensor of given shape and type, filled with zeros.

Usage

```
op_zeros(shape, dtype = NULL)
```

Arguments

shape	Shape of the new tensor.
dtype	Desired data type of the tensor.

Value

Tensor of zeros with the given shape and dtype.

See Also

- https://keras.io/api/ops/numpy#zeros-function

Other numpy ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_bincount()
op_broadcast_to()
op_ceil()
op_clip()
op_concatenate()
op_conj()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_cumprod()
op_cumsum()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_einsum()
op_empty()

op_equal()
op_exp()
op_expand_dims()
op_expm1()
op_eye()
op_flip()
op_floor()
op_floor_divide()
op_full()
op_full_like()
op_get_item()
op_greater()
op_greater_equal()
op_hstack()
op_identity()
op_imag()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_matmul()
op_max()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moveaxis()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()

op_not_equal()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_repeat()
op_reshape()
op_roll()
op_round()
op_sign()
op_sin()
op_sinh()
op_size()
op_sort()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_subtract()
op_sum()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_var()
op_vdot()
op_vstack()
op_where()
op_zeros_like()

Other ops:
op_abs()

op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()

op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()
op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()

op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()
op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()

op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()
op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()

op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros_like()

---

op_zeros_like *Return a tensor of zeros with the same shape and type as* x.

Description

Return a tensor of zeros with the same shape and type as x.

Usage

```
op_zeros_like(x, dtype = NULL)
```

Arguments

x	Input tensor.
dtype	Overrides the data type of the result.

Value

A tensor of zeros with the same shape and type as x.

See Also

- https://keras.io/api/ops/numpy#zeroslike-function

Other numpy ops:
op_abs()
op_add()
op_all()

op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()
op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_bincount()
op_broadcast_to()
op_ceil()
op_clip()
op_concatenate()
op_conj()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_cumprod()
op_cumsum()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_einsum()
op_empty()
op_equal()
op_exp()
op_expand_dims()
op_expm1()
op_eye()
op_flip()
op_floor()
op_floor_divide()
op_full()
op_full_like()

op_get_item()
op_greater()
op_greater_equal()
op_hstack()
op_identity()
op_imag()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_matmul()
op_max()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moveaxis()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_not_equal()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_quantile()
op_ravel()
op_real()

op_reciprocal()
op_repeat()
op_reshape()
op_roll()
op_round()
op_sign()
op_sin()
op_sinh()
op_size()
op_sort()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_subtract()
op_sum()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()
op_var()
op_vdot()
op_vstack()
op_where()
op_zeros()

Other ops:
op_abs()
op_add()
op_all()
op_any()
op_append()
op_arange()
op_arccos()
op_arccosh()
op_arcsin()
op_arcsinh()
op_arctan()

op_arctan2()
op_arctanh()
op_argmax()
op_argmin()
op_argsort()
op_array()
op_average()
op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_bincount()
op_broadcast_to()
op_cast()
op_categorical_crossentropy()
op_ceil()
op_cholesky()
op_clip()
op_concatenate()
op_cond()
op_conj()
op_conv()
op_conv_transpose()
op_convert_to_numpy()
op_convert_to_tensor()
op_copy()
op_correlate()
op_cos()
op_cosh()
op_count_nonzero()
op_cross()
op_ctc_loss()
op_cumprod()
op_cumsum()
op_custom_gradient()
op_depthwise_conv()
op_det()
op_diag()
op_diagonal()
op_diff()
op_digitize()
op_divide()
op_divide_no_nan()
op_dot()
op_eig()
op_einsum()
op_elu()
op_empty()
op_equal()

op_erf()
op_erfinv()
op_exp()
op_expand_dims()
op_expm1()
op_extract_sequences()
op_eye()
op_fft()
op_fft2()
op_flip()
op_floor()
op_floor_divide()
op_fori_loop()
op_full()
op_full_like()
op_gelu()
op_get_item()
op_greater()
op_greater_equal()
op_hard_sigmoid()
op_hard_silu()
op_hstack()
op_identity()
op_imag()
op_image_affine_transform()
op_image_crop()
op_image_extract_patches()
op_image_map_coordinates()
op_image_pad()
op_image_resize()
op_in_top_k()
op_inv()
op_irfft()
op_is_tensor()
op_isclose()
op_isfinite()
op_isinf()
op_isnan()
op_istft()
op_leaky_relu()
op_less()
op_less_equal()
op_linspace()
op_log()
op_log10()
op_log1p()
op_log2()
op_log_sigmoid()

op_log_softmax()
op_logaddexp()
op_logical_and()
op_logical_not()
op_logical_or()
op_logical_xor()
op_logspace()
op_logsumexp()
op_lu_factor()
op_matmul()
op_max()
op_max_pool()
op_maximum()
op_mean()
op_median()
op_meshgrid()
op_min()
op_minimum()
op_mod()
op_moments()
op_moveaxis()
op_multi_hot()
op_multiply()
op_nan_to_num()
op_ndim()
op_negative()
op_nonzero()
op_norm()
op_normalize()
op_not_equal()
op_one_hot()
op_ones()
op_ones_like()
op_outer()
op_pad()
op_power()
op_prod()
op_qr()
op_quantile()
op_ravel()
op_real()
op_reciprocal()
op_relu()
op_relu6()
op_repeat()
op_reshape()
op_rfft()
op_roll()

op_round()
op_rsqrt()
op_scatter()
op_scatter_update()
op_segment_max()
op_segment_sum()
op_selu()
op_separable_conv()
op_shape()
op_sigmoid()
op_sign()
op_silu()
op_sin()
op_sinh()
op_size()
op_slice()
op_slice_update()
op_softmax()
op_softplus()
op_softsign()
op_solve()
op_solve_triangular()
op_sort()
op_sparse_categorical_crossentropy()
op_split()
op_sqrt()
op_square()
op_squeeze()
op_stack()
op_std()
op_stft()
op_stop_gradient()
op_subtract()
op_sum()
op_svd()
op_swapaxes()
op_take()
op_take_along_axis()
op_tan()
op_tanh()
op_tensordot()
op_tile()
op_top_k()
op_trace()
op_transpose()
op_tri()
op_tril()
op_triu()

op_unstack()
op_var()
op_vdot()
op_vectorized_map()
op_vstack()
op_where()
op_while_loop()
op_zeros()

---

pad_sequences *Pads sequences to the same length.*

Description

This function transforms a list (of length num_samples) of sequences (lists of integers) into a 2D NumPy array of shape (num_samples, num_timesteps). num_timesteps is either the maxlen argument if provided, or the length of the longest sequence in the list.

Sequences that are shorter than num_timesteps are padded with value until they are num_timesteps long.

Sequences longer than num_timesteps are truncated so that they fit the desired length.

The position where padding or truncation happens is determined by the arguments padding and truncating, respectively. Pre-padding or removing values from the beginning of the sequence is the default.

```
sequence <- list(c(1), c(2, 3), c(4, 5, 6))
pad_sequences(sequence)
```

```
##      [,1] [,2] [,3]
## [1,]    0    0    1
## [2,]    0    2    3
## [3,]    4    5    6
```

```
pad_sequences(sequence, value=-1)
```

```
##      [,1] [,2] [,3]
## [1,]   -1   -1    1
## [2,]   -1    2    3
## [3,]    4    5    6
```

```
pad_sequences(sequence, padding='post')
```

```
##      [,1] [,2] [,3]
## [1,]    1    0    0
## [2,]    2    3    0
## [3,]    4    5    6
```

```
pad_sequences(sequence, maxlen=2)
```

```
##      [,1] [,2]
## [1,]    0    1
## [2,]    2    3
## [3,]    5    6
```

Usage

```
pad_sequences(
  sequences,
  maxlen = NULL,
  dtype = "int32",
  padding = "pre",
  truncating = "pre",
  value = 0
)
```

Arguments

sequences	List of sequences (each sequence is a list of integers).
maxlen	Optional Int, maximum length of all sequences. If not provided, sequences will be padded to the length of the longest individual sequence.
dtype	(Optional, defaults to "int32"). Type of the output sequences. To pad sequences with variable length strings, you can use object.
padding	String, "pre" or "post" (optional, defaults to "pre"): pad either before or after each sequence.
truncating	String, "pre" or "post" (optional, defaults to "pre"): remove values from sequences larger than maxlen, either at the beginning or at the end of the sequences.
value	Float or String, padding value. (Optional, defaults to 0.)

Value

Array with shape (len(sequences), maxlen)

See Also

- https://keras.io/api/data_loading/timeseries#padsequences-function

Other utils:
audio_dataset_from_directory()

clear_session()
config_disable_interactive_logging()
config_disable_traceback_filtering()
config_enable_interactive_logging()
config_enable_traceback_filtering()
config_is_interactive_logging_enabled()
config_is_traceback_filtering_enabled()
get_file()
get_source_inputs()
image_array_save()
image_dataset_from_directory()
image_from_array()
image_load()
image_smart_resize()
image_to_array()
layer_feature_space()
normalize()
pack_x_y_sample_weight()
set_random_seed()
split_dataset()
text_dataset_from_directory()
timeseries_dataset_from_array()
to_categorical()
unpack_x_y_sample_weight()
zip_lists()

---

plot.keras.src.models.model.Model

Plot a Keras model

Description

Plot a Keras model

Usage

```
## S3 method for class 'keras.src.models.model.Model'
plot(
  x,
  show_shapes = FALSE,
  show_dtype = FALSE,
  show_layer_names = FALSE,
  ...,
  rankdir = "TB",
  expand_nested = FALSE,
  dpi = 200,
  layer_range = NULL,
  show_layer_activations = FALSE,
  show_trainable = NA,
  to_file = NULL
)
```

Arguments

x	A Keras model instance
show_shapes	whether to display shape information.
show_dtype	whether to display layer dtypes.
show_layer_names	whether to display layer names.
...	passed on to Python `keras.utils.model_to_dot()`. Used for forward and backward compatibility.
rankdir	a string specifying the format of the plot: `'TB'` creates a vertical plot; `'LR'` creates a horizontal plot. (argument passed to PyDot)
expand_nested	Whether to expand nested models into clusters.
dpi	Dots per inch. Increase this value if the image text appears excessively pixelated.
layer_range	`list` containing two character strings, which is the starting layer name and ending layer name (both inclusive) indicating the range of layers for which the plot will be generated. It also accepts regex patterns instead of exact name. In such case, start predicate will be the first element it matches to `layer_range[1]` and the end predicate will be the last element it matches to `layer_range[2]`. By default `NULL` which considers all layers of model. Note that you must pass range such that the resultant subgraph must be complete.
show_layer_activations	Display layer activations (only for layers that have an `activation` property).
show_trainable	whether to display if a layer is trainable.
to_file	File name of the plot image. If `NULL` (the default), the model is drawn on the default graphics device. Otherwise, a file is saved.

Value

Nothing, called for it's side effects.

Raises

ValueError: if `plot(model)` is called before the model is built, unless a `input_shape =` argument was supplied to `keras_model_sequential()`.

Requirements

This function requires pydot and graphviz. `pydot` is by default installed by `install_keras()`, but if you installed keras by other means, you can install `pydot` directly with :

```
reticulate::py_install("pydot", pip = TRUE)
```

You can install graphviz directly from here: https://graphviz.gitlab.io/download/

On most Linux platforms, can install graphviz via the package manager. For example, on Ubuntu/Debian you can install with

```
sudo apt install graphviz
```

In a conda environment, you can install graphviz with:

```
reticulate::conda_install(packages = "graphviz")
# Restart the R session after install.
```

---

plot.keras_training_history

Plot training history

Description

Plots metrics recorded during training.

Usage

```
## S3 method for class 'keras_training_history'
plot(
  x,
  y,
  metrics = NULL,
  method = c("auto", "ggplot2", "base"),
  smooth = getOption("keras.plot.history.smooth", TRUE),
  theme_bw = getOption("keras.plot.history.theme_bw", FALSE),
  ...
)
```

Arguments

`x`	Training history object returned from `fit.keras.src.models.model.Model()`.
`y`	Unused.
`metrics`	One or more metrics to plot (e.g. `c('loss', 'accuracy')`). Defaults to plotting all captured metrics.
`method`	Method to use for plotting. The default "auto" will use **ggplot2** if available, and otherwise will use base graphics.
`smooth`	Whether a loess smooth should be added to the plot, only available for the `ggplot2` method. If the number of epochs is smaller than ten, it is forced to false.
`theme_bw`	Use `ggplot2::theme_bw()` to plot the history in black and white.
`...`	Additional parameters to pass to the `plot()` method.

Value

if `method == "ggplot2"`, the ggplot object is returned. If `method == "base"`, then this function will draw to the graphics device and return `NULL`, invisibly.

`pop_layer` *Remove the last layer in a Sequential model*

Description

Remove the last layer in a Sequential model

Usage

```
pop_layer(object)
```

Arguments

`object`	Sequential keras model object

Value

The input `object`, invisibly.

See Also

Other model functions:
`get_config()`
`get_layer()`
`keras_model()`
`keras_model_sequential()`
`summary.keras.src.models.model.Model()`

`predict.keras.src.models.model.Model` *Generates output predictions for the input samples.*

Description

Generates output predictions for the input samples.

Usage

```
## S3 method for class 'keras.src.models.model.Model'
predict(
  object,
  x,
  ...,
  batch_size = NULL,
  verbose = getOption("keras.verbose", default = "auto"),
  steps = NULL,
  callbacks = NULL
)
```

Arguments

`object`	Keras model object
`x`	Input samples. It could be: • A array (or array-like), or a list of arrays (in case the model has multiple inputs). • A tensor, or a list of tensors (in case the model has multiple inputs). • A TF Dataset.
`...`	For forward/backward compatability.
`batch_size`	Integer or `NULL`. Number of samples per batch. If unspecified, `batch_size` will default to `32`. Do not specify the `batch_size` if your data is in the form of a TF Dataset or a generator (since they generate batches).
`verbose`	`"auto"`, `0`, `1`, or `2`. Verbosity mode. `0` = silent, `1` = progress bar, `2` = one line per epoch. `"auto"` becomes 1 for most cases, `2` if in a knitr render or running on a distributed training server. Note that the progress bar is not particularly useful when logged to a file, so `verbose=2` is recommended when not running interactively (e.g., in a production environment). Defaults to `"auto"`.
`steps`	Total number of steps (batches of samples) before declaring the prediction round finished. Ignored with the default value of `NULL`. If `x` is a TF Dataset and `steps` is `NULL`, `predict()` will run until the input dataset is exhausted.
`callbacks`	List of `Callback` instances. List of callbacks to apply during prediction.

Details

Computation is done in batches. This method is designed for batch processing of large numbers of inputs. It is not intended for use inside of loops that iterate over your data and process small numbers of inputs at a time.

For small numbers of inputs that fit in one batch, directly call the model `model$call` for faster execution, e.g., `model(x)`, or `model(x, training = FALSE)` if you have layers such as `BatchNormalization` that behave differently during inference.

Value

R array(s) of predictions.

Note

See this FAQ entry for more details about the difference between `Model` methods `predict()` and `call()`.

See Also

- `https://keras.io/api/models/model_training_apis#predict-method`

Other model training:
`compile.keras.src.models.model.Model()`
`evaluate.keras.src.models.model.Model()`
`predict_on_batch()`
`test_on_batch()`
`train_on_batch()`

`predict_on_batch` *Returns predictions for a single batch of samples.*

Description

Returns predictions for a single batch of samples.

Usage

```
predict_on_batch(object, x)
```

Arguments

`object`	Keras model object
`x`	Input data. It must be array-like.

Value

Array(s) of predictions.

See Also

- `https://keras.io/api/models/model_training_apis#predictonbatch-method`

Other model training:
`compile.keras.src.models.model.Model()`
`evaluate.keras.src.models.model.Model()`
`predict.keras.src.models.model.Model()`
`test_on_batch()`
`train_on_batch()`

process_utils *Preprocessing and postprocessing utilities*

Description

These functions are used to preprocess and postprocess inputs and outputs of Keras applications.

Usage

```
application_preprocess_inputs(model, x, ..., data_format = NULL)

application_decode_predictions(model, preds, top = 5L, ...)
```

Arguments

`model`	A Keras model initialized using any `application_` function.
`x`	A batch of inputs to the model.
`...`	Additional arguments passed to the preprocessing or decoding function.
`data_format`	Optional data format of the image tensor/array. `NULL` means the global setting `config_image_data_format()` is used (unless you changed it, it uses `"channels_last"`). Defaults to `NULL`.
`preds`	A batch of outputs from the model.
`top`	The number of top predictions to return.

Value

- A list of decoded predictions in case of `application_decode_predictions()`.
- A batch of preprocessed inputs in case of `application_preprocess_inputs()`.

Functions

- `application_preprocess_inputs()`: Pre-process inputs to be used in the model
- `application_decode_predictions()`: Decode predictions from the model

Examples

```
## Not run:
model <- application_convnext_tiny()

inputs <- random_normal(c(32, 224, 224, 3))
processed_inputs <- application_preprocess_inputs(model, inputs)

preds <- random_normal(c(32, 1000))
decoded_preds <- application_decode_predictions(model, preds)


## End(Not run)
```

`quantize_weights` *Quantize the weights of a model.*

Description

Note that the model must be built first before calling this method. `quantize_weights()` will recursively call `layer$quantize(mode)` in all layers and will be skipped if the layer doesn't implement the function.

Currently only `Dense` and `EinsumDense` layers support quantization.

Usage

```
quantize_weights(object, mode)
```

Arguments

`object`	A Keras Model or Layer.
`mode`	The mode of the quantization. Only 'int8' is supported at this time.

Value

`model`, invisibly. Note this is just a convenience for usage with `|>`, the model is modified in-place.

See Also

Other layer methods:
`count_params()`
`get_config()`
`get_weights()`
`reset_state()`

`random_beta` *Draw samples from a Beta distribution.*

Description

The values are drawm from a Beta distribution parametrized by alpha and beta.

Usage

```
random_beta(shape, alpha, beta, dtype = NULL, seed = NULL)
```

Arguments

`shape`	The shape of the random values to generate.
`alpha`	Float or an array of floats representing the first parameter alpha. Must be broadcastable with `beta` and `shape`.
`beta`	Float or an array of floats representing the second parameter beta. Must be broadcastable with `alpha` and `shape`.
`dtype`	Optional dtype of the tensor. Only floating point types are supported. If not specified, `config_floatx()` is used, which defaults to `"float32"` unless you configured it otherwise (via `config_set_floatx(float_dtype)`).
`seed`	An integer or instance of `random_seed_generator()`. Used to make the behavior of the initializer deterministic. Note that an initializer seeded with an integer or NULL (unseeded) will produce the same random values across multiple calls. To get different random values across multiple calls, use as seed an instance of `random_seed_generator()`.

Value

A tensor of random values.

See Also

Other random:
`random_binomial()`
`random_categorical()`
`random_dropout()`
`random_gamma()`
`random_integer()`
`random_normal()`
`random_seed_generator()`
`random_shuffle()`
`random_truncated_normal()`
`random_uniform()`

`random_binomial` *Draw samples from a Binomial distribution.*

Description

The values are drawn from a Binomial distribution with specified trial count and probability of success.

Usage

```
random_binomial(shape, counts, probabilities, dtype = NULL, seed = NULL)
```

Arguments

`shape`	The shape of the random values to generate.
`counts`	A number or array of numbers representing the number of trials. It must be broadcastable with `probabilities`.
`probabilities`	A float or array of floats representing the probability of success of an individual event. It must be broadcastable with `counts`.
`dtype`	Optional dtype of the tensor. Only floating point types are supported. If not specified, `config_floatx()` is used, which defaults to `"float32"` unless you configured it otherwise (via `config_set_floatx(float_dtype)`).
`seed`	A Python integer or instance of `random_seed_generator()`. Used to make the behavior of the initializer deterministic. Note that an initializer seeded with an integer or None (unseeded) will produce the same random values across multiple calls. To get different random values across multiple calls, use as seed an instance of `random_seed_generator()`.

Value

A tensor of random values.

See Also

- `https://www.tensorflow.org/api_docs/python/tf/keras/random/binomial`

Other random:
`random_beta()`
`random_categorical()`
`random_dropout()`
`random_gamma()`
`random_integer()`
`random_normal()`
`random_seed_generator()`
`random_shuffle()`
`random_truncated_normal()`
`random_uniform()`

`random_categorical` *Draws samples from a categorical distribution.*

Description

This function takes as input `logits`, a 2-D input tensor with shape (batch_size, num_classes). Each row of the input represents a categorical distribution, with each column index containing the log-probability for a given class.

The function will output a 2-D tensor with shape (batch_size, num_samples), where each row contains samples from the corresponding row in `logits`. Each column index contains an independent samples drawn from the input distribution.

Usage

```
random_categorical(logits, num_samples, dtype = "int32", seed = NULL)
```

Arguments

`logits`	2-D Tensor with shape (batch_size, num_classes). Each row should define a categorical distibution with the unnormalized log-probabilities for all classes.
`num_samples`	Int, the number of independent samples to draw for each row of the input. This will be the second dimension of the output tensor's shape.
`dtype`	Optional dtype of the output tensor.
`seed`	An R integer or instance of `random_seed_generator()`. Used to make the behavior of the initializer deterministic. Note that an initializer seeded with an integer or `NULL` (unseeded) will produce the same random values across multiple calls. To get different random values across multiple calls, use as seed an instance of `random_seed_generator()`.

Value

A 2-D tensor with (batch_size, num_samples).

See Also

Other random:
`random_beta()`
`random_binomial()`
`random_dropout()`
`random_gamma()`
`random_integer()`
`random_normal()`
`random_seed_generator()`
`random_shuffle()`
`random_truncated_normal()`
`random_uniform()`

`random_dropout` *Randomly set some values in a tensor to 0.*

Description

Randomly set some portion of values in the tensor to 0.

Usage

```
random_dropout(inputs, rate, noise_shape = NULL, seed = NULL)
```

Arguments

`inputs`	A tensor
`rate`	numeric
`noise_shape`	A `shape()` value
`seed`	Initial seed for the random number generator

Value

A tensor that is a copy of `inputs` with some values set to `0`.

See Also

Other random:
`random_beta()`
`random_binomial()`
`random_categorical()`
`random_gamma()`
`random_integer()`
`random_normal()`
`random_seed_generator()`
`random_shuffle()`
`random_truncated_normal()`
`random_uniform()`

`random_gamma` *Draw random samples from the Gamma distribution.*

Description

Draw random samples from the Gamma distribution.

Usage

```
random_gamma(shape, alpha, dtype = NULL, seed = NULL)
```

Arguments

`shape`	The shape of the random values to generate.
`alpha`	Float, the parameter of the distribution.
`dtype`	Optional dtype of the tensor. Only floating point types are supported. If not specified, `config_floatx()` is used, which defaults to `float32` unless you configured it otherwise (via `config_set_floatx(float_dtype)`).
`seed`	An R integer or instance of `random_seed_generator()`. Used to make the behavior of the initializer deterministic. Note that an initializer seeded with an integer or `NULL` (unseeded) will produce the same random values across multiple calls. To get different random values across multiple calls, use as seed an instance of `random_seed_generator()`.

Value

A tensor of random values.

See Also

Other random:
random_beta()
random_binomial()
random_categorical()
random_dropout()
random_integer()
random_normal()
random_seed_generator()
random_shuffle()
random_truncated_normal()
random_uniform()

---

random_integer — *Draw random integers from a uniform distribution.*

Description

The generated values follow a uniform distribution in the range `[minval, maxval)`. The lower bound `minval` is included in the range, while the upper bound `maxval` is excluded.

`dtype` must be an integer type.

Usage

```
random_integer(shape, minval, maxval, dtype = "int32", seed = NULL)
```

Arguments

`shape`	The shape of the random values to generate.
`minval`	integer, lower bound of the range of random values to generate (inclusive).
`maxval`	integer, upper bound of the range of random values to generate (exclusive).
`dtype`	Optional dtype of the tensor. Only integer types are supported. If not specified, `"int32"` is used.
`seed`	An R integer or instance of `random_seed_generator()`. Used to make the behavior of the initializer deterministic. Note that an initializer seeded with an integer or `NULL` (unseeded) will produce the same random values across multiple calls. To get different random values across multiple calls, use as seed an instance of `random_seed_generator()`.

Value

A tensor of random values.

See Also

Other random:
random_beta()
random_binomial()
random_categorical()
random_dropout()
random_gamma()
random_normal()
random_seed_generator()
random_shuffle()
random_truncated_normal()
random_uniform()

`random_normal` *Draw random samples from a normal (Gaussian) distribution.*

Description

Draw random samples from a normal (Gaussian) distribution.

Usage

```
random_normal(shape, mean = 0, stddev = 1, dtype = NULL, seed = NULL)
```

Arguments

shape	The shape of the random values to generate.
mean	Float, defaults to 0. Mean of the random values to generate.
stddev	Float, defaults to 1. Standard deviation of the random values to generate.
dtype	Optional dtype of the tensor. Only floating point types are supported. If not specified, `config_floatx()` is used, which defaults to `float32` unless you configured it otherwise (via `config_set_floatx(float_dtype)`).
seed	An R integer or instance of `random_seed_generator()`. Used to make the behavior of the initializer deterministic. Note that an initializer seeded with an integer or `NULL` (unseeded) will produce the same random values across multiple calls. To get different random values across multiple calls, use as seed an instance of `random_seed_generator()`.

Value

A tensor of random values.

See Also

Other random:
random_beta()
random_binomial()
random_categorical()
random_dropout()
random_gamma()
random_integer()
random_seed_generator()
random_shuffle()
random_truncated_normal()
random_uniform()

---

random_seed_generator *Generates variable seeds upon each call to a RNG-using function.*

Description

In Keras, all RNG-using methods (such as `random_normal()`) are stateless, meaning that if you pass an integer seed to them (such as `seed = 42`), they will return the same values at each call. In order to get different values at each call, you must use a `SeedGenerator` instead as the seed argument. The `SeedGenerator` object is stateful.

Usage

```
random_seed_generator(seed = NULL, ...)
```

Arguments

`seed`	Initial seed for the random number generator
`...`	For forward/backward compatability.

Value

A `SeedGenerator` instance, which can be passed as the `seed =` argument to other random tensor generators.

Examples

```
seed_gen <- random_seed_generator(seed = 42)
values <- random_normal(shape = c(2, 3), seed = seed_gen)
new_values <- random_normal(shape = c(2, 3), seed = seed_gen)
```

Usage in a layer:

```
layer_dropout2 <- new_layer_class(
  "dropout2",
  initialize = function(...) {
    super$initialize(...)
    self$seed_generator <- random_seed_generator(seed = 1337)
  },
  call = function(x, training = FALSE) {
    if (training) {
      return(random_dropout(x, rate = 0.5, seed = self$seed_generator))
    }
    return(x)
  }
)

out <- layer_dropout(rate = 0.8)
out(op_ones(10), training = TRUE)
```

```
## tf.Tensor([0. 5. 5. 0. 0. 0. 0. 0. 0. 0.], shape=(10), dtype=float32)
```

See Also

Other random:
`random_beta()`
`random_binomial()`
`random_categorical()`
`random_dropout()`
`random_gamma()`
`random_integer()`
`random_normal()`
`random_shuffle()`
`random_truncated_normal()`
`random_uniform()`

`random_shuffle` *Shuffle the elements of a tensor uniformly at random along an axis.*

Description

Shuffle the elements of a tensor uniformly at random along an axis.

Usage

```
random_shuffle(x, axis = 1L, seed = NULL)
```

Arguments

`x`	The tensor to be shuffled.
`axis`	An integer specifying the axis along which to shuffle. Defaults to `0`.
`seed`	An R integer or instance of `random_seed_generator()`. Used to make the behavior of the initializer deterministic. Note that an initializer seeded with an integer or `NULL` (unseeded) will produce the same random values across multiple calls. To get different random values across multiple calls, use as seed an instance of `random_seed_generator()`.

Value

A tensor, a copy of `x` with the `axis` axis shuffled.

See Also

Other random:
`random_beta()`
`random_binomial()`
`random_categorical()`
`random_dropout()`
`random_gamma()`
`random_integer()`
`random_normal()`
`random_seed_generator()`
`random_truncated_normal()`
`random_uniform()`

---

random_truncated_normal

Draw samples from a truncated normal distribution.

Description

The values are drawn from a normal distribution with specified mean and standard deviation, discarding and re-drawing any samples that are more than two standard deviations from the mean.

Usage

```
random_truncated_normal(shape, mean = 0, stddev = 1, dtype = NULL, seed = NULL)
```

Arguments

`shape`	The shape of the random values to generate.
`mean`	Float, defaults to 0. Mean of the random values to generate.
`stddev`	Float, defaults to 1. Standard deviation of the random values to generate.
`dtype`	Optional dtype of the tensor. Only floating point types are supported. If not specified, `config_floatx()` is used, which defaults to `float32` unless you configured it otherwise (via `config_set_floatx(float_dtype)`)
`seed`	An R integer or instance of `random_seed_generator()`. Used to make the behavior of the initializer deterministic. Note that an initializer seeded with an integer or `NULL` (unseeded) will produce the same random values across multiple calls. To get different random values across multiple calls, use as seed an instance of `random_seed_generator()`.

Value

A tensor of random values.

See Also

Other random:
`random_beta()`
`random_binomial()`
`random_categorical()`
`random_dropout()`
`random_gamma()`
`random_integer()`
`random_normal()`
`random_seed_generator()`
`random_shuffle()`
`random_uniform()`

`random_uniform` *Draw samples from a uniform distribution.*

Description

The generated values follow a uniform distribution in the range `[minval, maxval)`. The lower bound `minval` is included in the range, while the upper bound `maxval` is excluded.

`dtype` must be a floating point type, the default range is `[0, 1)`.

Usage

```
random_uniform(shape, minval = 0, maxval = 1, dtype = NULL, seed = NULL)
```

Arguments

shape	The shape of the random values to generate.
minval	Float, defaults to 0. Lower bound of the range of random values to generate (inclusive).
maxval	Float, defaults to 1. Upper bound of the range of random values to generate (exclusive).
dtype	Optional dtype of the tensor. Only floating point types are supported. If not specified, `config_floatx()` is used, which defaults to `float32` unless you configured it otherwise (via `config_set_floatx(float_dtype)`)
seed	An R integer or instance of `random_seed_generator()`. Used to make the behavior of the initializer deterministic. Note that an initializer seeded with an integer or `NULL` (unseeded) will produce the same random values across multiple calls. To get different random values across multiple calls, use as seed an instance of `random_seed_generator()`.

Value

A tensor of random values.

See Also

Other random:
`random_beta()`
`random_binomial()`
`random_categorical()`
`random_dropout()`
`random_gamma()`
`random_integer()`
`random_normal()`
`random_seed_generator()`
`random_shuffle()`
`random_truncated_normal()`

`register_keras_serializable`
Registers a custom object with the Keras serialization framework.

Description

This function registers a custom class or function with the Keras custom object registry, so that it can be serialized and deserialized without needing an entry in the user-provided `custom_objects` argument. It also injects a function that Keras will call to get the object's serializable string key.

Note that to be serialized and deserialized, classes must implement the `get_config()` method. Functions do not have this requirement.

The object will be registered under the key `'package>name'` where `name`, defaults to the object name if not passed.

Usage

```
register_keras_serializable(object, name = NULL, package = NULL)
```

Arguments

`object`	A keras object.
`name`	The name to serialize this class under in this package.
`package`	The package that this class belongs to. This is used for the key (which is `"package>name"`) to identify the class. Defaults to the current package name, or `"Custom"` outside of a package.

Value

`object` is returned invisibly, for convenient piping. This is primarily called for side effects.

Examples

```
# Note that `'my_package'` is used as the `package` argument here, and since
# the `name` argument is not provided, `'MyDense'` is used as the `name`.
layer_my_dense <- Layer("MyDense")
register_keras_serializable(layer_my_dense, package = "my_package")

MyDense <- environment(layer_my_dense)$`__class__` # the python class obj
stopifnot(exprs = {
  get_registered_object('my_package>MyDense') == MyDense
  get_registered_name(MyDense) == 'my_package>MyDense'
})
```

See Also

Other saving and loading functions:
`export_savedmodel.keras.src.models.model.Model()`
`layer_tfsm()`
`load_model()`
`load_model_weights()`
`save_model()`
`save_model_config()`
`save_model_weights()`
`with_custom_object_scope()`

Other serialization utilities:
`deserialize_keras_object()`
`get_custom_objects()`
`get_registered_name()`
`get_registered_object()`
`serialize_keras_object()`
`with_custom_object_scope()`

`regularizer_l1` *A regularizer that applies a L1 regularization penalty.*

Description

The L1 regularization penalty is computed as: `loss = l1 * reduce_sum(abs(x))`

L1 may be passed to a layer as a string identifier:

```
dense <- layer_dense(units = 3, kernel_regularizer = 'l1')
```

In this case, the default value used is `l1=0.01`.

Usage

```
regularizer_l1(l1 = 0.01)
```

Arguments

`l1`	float, L1 regularization factor.

Value

A `Regularizer` instance that can be passed to layer constructors or used as a standalone object.

See Also

- https://keras.io/api/layers/regularizers#l1-class

Other regularizers:
`regularizer_l1_l2()`
`regularizer_l2()`
`regularizer_orthogonal()`

`regularizer_l1_l2` *A regularizer that applies both L1 and L2 regularization penalties.*

Description

The L1 regularization penalty is computed as: `loss = l1 * reduce_sum(abs(x))`

The L2 regularization penalty is computed as `loss = l2 * reduce_sum(square(x))`

L1L2 may be passed to a layer as a string identifier:

```
dense <- layer_dense(units = 3, kernel_regularizer = 'L1L2')
```

In this case, the default values used are `l1=0.01` and `l2=0.01`.

Usage

```
regularizer_l1_l2(l1 = 0, l2 = 0)
```

Arguments

`l1`	float, L1 regularization factor.
`l2`	float, L2 regularization factor.

Value

A `Regularizer` instance that can be passed to layer constructors or used as a standalone object.

See Also

- `https://keras.io/api/layers/regularizers#l1l2-class`

Other regularizers:
`regularizer_l1()`
`regularizer_l2()`
`regularizer_orthogonal()`

`regularizer_l2` *A regularizer that applies a L2 regularization penalty.*

Description

The L2 regularization penalty is computed as: `loss = l2 * reduce_sum(square(x))`

L2 may be passed to a layer as a string identifier:

```
dense <- layer_dense(units = 3, kernel_regularizer='l2')
```

In this case, the default value used is `l2=0.01`.

Usage

```
regularizer_l2(l2 = 0.01)
```

Arguments

`l2`	float, L2 regularization factor.

Value

A `Regularizer` instance that can be passed to layer constructors or used as a standalone object.

See Also

- https://keras.io/api/layers/regularizers#l2-class

Other regularizers:
`regularizer_l1()`
`regularizer_l1_l2()`
`regularizer_orthogonal()`

`regularizer_orthogonal` *Regularizer that encourages input vectors to be orthogonal to each other.*

Description

It can be applied to either the rows of a matrix (`mode="rows"`) or its columns (`mode="columns"`). When applied to a `Dense` kernel of shape `(input_dim, units)`, rows mode will seek to make the feature vectors (i.e. the basis of the output space) orthogonal to each other.

Usage

```
regularizer_orthogonal(factor = 0.01, mode = "rows")
```

Arguments

`factor`	Float. The regularization factor. The regularization penalty will be proportional to `factor` times the mean of the dot products between the L2-normalized rows (if `mode="rows"`, or columns if `mode="columns"`) of the inputs, excluding the product of each row/column with itself. Defaults to `0.01`.
`mode`	String, one of `{"rows", "columns"}`. Defaults to `"rows"`. In rows mode, the regularization effect seeks to make the rows of the input orthogonal to each other. In columns mode, it seeks to make the columns of the input orthogonal to each other.

Value

A `Regularizer` instance that can be passed to layer constructors or used as a standalone object.

Examples

```
regularizer <- regularizer_orthogonal(factor=0.01)
layer <- layer_dense(units=4, kernel_regularizer=regularizer)
```

See Also

- https://keras.io/api/layers/regularizers#orthogonalregularizer-class

Other regularizers:
`regularizer_l1()`
`regularizer_l1_l2()`
`regularizer_l2()`

`reset_state` *Reset the state for a model, layer or metric.*

Description

Reset the state for a model, layer or metric.

Usage

```
reset_state(object)
```

Arguments

`object`	Model, Layer, or Metric instance Not all Layers have resettable state (E.g., `adapt()`-able preprocessing layers and rnn layers have resettable state, but a `layer_dense()` does not). Calling this on a Layer instance without any resettable-state will error.

Value

`object`, invisibly.

See Also

Other layer methods:
`count_params()`
`get_config()`
`get_weights()`
`quantize_weights()`

rnn_cells_stack *Wrapper allowing a stack of RNN cells to behave as a single cell.*

Description

Used to implement efficient stacked RNNs.

Usage

```
rnn_cells_stack(cells, ...)
```

Arguments

cells	List of RNN cell instances.
...	Unnamed arguments are treated as additional cells. Named arguments are passed on to the underlying layer.

Value

A Layer instance, which is intended to be used with layer_rnn().

Example

```
batch_size <- 3
sentence_length <- 5
num_features <- 2
new_shape <- c(batch_size, sentence_length, num_features)
x <- array(1:30, dim = new_shape)

rnn_cells <- lapply(1:2, function(x) rnn_cell_lstm(units = 128))
stacked_lstm <- rnn_cells_stack(rnn_cells)
lstm_layer <- layer_rnn(cell = stacked_lstm)

result <- lstm_layer(x)
str(result)

## <tf.Tensor: shape=(3, 128), dtype=float32, numpy=. . .>
```

See Also

Other rnn layers:
layer_bidirectional()
layer_conv_lstm_1d()
layer_conv_lstm_2d()
layer_conv_lstm_3d()
layer_gru()

layer_lstm()
layer_rnn()
layer_simple_rnn()
layer_time_distributed()
rnn_cell_gru()
rnn_cell_lstm()
rnn_cell_simple()

Other layers:
Layer()
layer_activation()
layer_activation_elu()
layer_activation_leaky_relu()
layer_activation_parametric_relu()
layer_activation_relu()
layer_activation_softmax()
layer_activity_regularization()
layer_add()
layer_additive_attention()
layer_alpha_dropout()
layer_attention()
layer_average()
layer_average_pooling_1d()
layer_average_pooling_2d()
layer_average_pooling_3d()
layer_batch_normalization()
layer_bidirectional()
layer_category_encoding()
layer_center_crop()
layer_concatenate()
layer_conv_1d()
layer_conv_1d_transpose()
layer_conv_2d()
layer_conv_2d_transpose()
layer_conv_3d()
layer_conv_3d_transpose()
layer_conv_lstm_1d()
layer_conv_lstm_2d()
layer_conv_lstm_3d()
layer_cropping_1d()
layer_cropping_2d()
layer_cropping_3d()
layer_dense()
layer_depthwise_conv_1d()
layer_depthwise_conv_2d()
layer_discretization()
layer_dot()
layer_dropout()

layer_einsum_dense()
layer_embedding()
layer_feature_space()
layer_flatten()
layer_flax_module_wrapper()
layer_gaussian_dropout()
layer_gaussian_noise()
layer_global_average_pooling_1d()
layer_global_average_pooling_2d()
layer_global_average_pooling_3d()
layer_global_max_pooling_1d()
layer_global_max_pooling_2d()
layer_global_max_pooling_3d()
layer_group_normalization()
layer_group_query_attention()
layer_gru()
layer_hashed_crossing()
layer_hashing()
layer_identity()
layer_integer_lookup()
layer_jax_model_wrapper()
layer_lambda()
layer_layer_normalization()
layer_lstm()
layer_masking()
layer_max_pooling_1d()
layer_max_pooling_2d()
layer_max_pooling_3d()
layer_maximum()
layer_mel_spectrogram()
layer_minimum()
layer_multi_head_attention()
layer_multiply()
layer_normalization()
layer_permute()
layer_random_brightness()
layer_random_contrast()
layer_random_crop()
layer_random_flip()
layer_random_rotation()
layer_random_translation()
layer_random_zoom()
layer_repeat_vector()
layer_rescaling()
layer_reshape()
layer_resizing()
layer_rnn()
layer_separable_conv_1d()

layer_separable_conv_2d()
layer_simple_rnn()
layer_spatial_dropout_1d()
layer_spatial_dropout_2d()
layer_spatial_dropout_3d()
layer_spectral_normalization()
layer_string_lookup()
layer_subtract()
layer_text_vectorization()
layer_tfsm()
layer_time_distributed()
layer_torch_module_wrapper()
layer_unit_normalization()
layer_upsampling_1d()
layer_upsampling_2d()
layer_upsampling_3d()
layer_zero_padding_1d()
layer_zero_padding_2d()
layer_zero_padding_3d()
rnn_cell_gru()
rnn_cell_lstm()
rnn_cell_simple()

---

rnn_cell_gru *Cell class for the GRU layer.*

Description

This class processes one step within the whole time sequence input, whereas layer_gru() processes the whole sequence.

Usage

```
rnn_cell_gru(
  units,
  activation = "tanh",
  recurrent_activation = "sigmoid",
  use_bias = TRUE,
  kernel_initializer = "glorot_uniform",
  recurrent_initializer = "orthogonal",
  bias_initializer = "zeros",
  kernel_regularizer = NULL,
  recurrent_regularizer = NULL,
  bias_regularizer = NULL,
  kernel_constraint = NULL,
  recurrent_constraint = NULL,
  bias_constraint = NULL,
  dropout = 0,
  recurrent_dropout = 0,
  reset_after = TRUE,
  seed = NULL,
  ...
)
```

Arguments

units	Positive integer, dimensionality of the output space.
activation	Activation function to use. Default: hyperbolic tangent (tanh). If you pass NULL, no activation is applied (ie. "linear" activation: a(x) = x).
recurrent_activation	Activation function to use for the recurrent step. Default: sigmoid (sigmoid). If you pass NULL, no activation is applied (ie. "linear" activation: a(x) = x).
use_bias	Boolean, (default TRUE), whether the layer should use a bias vector.
kernel_initializer	Initializer for the kernel weights matrix, used for the linear transformation of the inputs. Default: "glorot_uniform".
recurrent_initializer	Initializer for the recurrent_kernel weights matrix, used for the linear transformation of the recurrent state. Default: "orthogonal".
bias_initializer	Initializer for the bias vector. Default: "zeros".
kernel_regularizer	Regularizer function applied to the kernel weights matrix. Default: NULL.
recurrent_regularizer	Regularizer function applied to the recurrent_kernel weights matrix. Default: NULL.
bias_regularizer	Regularizer function applied to the bias vector. Default: NULL.
kernel_constraint	Constraint function applied to the kernel weights matrix. Default: NULL.
recurrent_constraint	Constraint function applied to the recurrent_kernel weights matrix. Default: NULL.
bias_constraint	Constraint function applied to the bias vector. Default: NULL.
dropout	Float between 0 and 1. Fraction of the units to drop for the linear transformation of the inputs. Default: 0.
recurrent_dropout	Float between 0 and 1. Fraction of the units to drop for the linear transformation of the recurrent state. Default: 0.
reset_after	GRU convention (whether to apply reset gate after or before matrix multiplication). FALSE = "before", TRUE = "after" (default and cuDNN compatible).
seed	Random seed for dropout.
...	For forward/backward compatability.

Value

A `Layer` instance, which is intended to be used with `layer_rnn()`.

Call Arguments

- `inputs`: A 2D tensor, with shape `(batch, features)`.
- `states`: A 2D tensor with shape `(batch, units)`, which is the state from the previous time step.
- `training`: Python boolean indicating whether the layer should behave in training mode or in inference mode. Only relevant when `dropout` or `recurrent_dropout` is used.

Examples

```
inputs <- random_uniform(c(32, 10, 8))
outputs <- inputs |> layer_rnn(rnn_cell_gru(4))
shape(outputs)

## shape(32, 4)


rnn <- layer_rnn(
   cell = rnn_cell_gru(4),
   return_sequences=TRUE,
   return_state=TRUE)
c(whole_sequence_output, final_state) %<-% rnn(inputs)
shape(whole_sequence_output)

## shape(32, 10, 4)


shape(final_state)

## shape(32, 4)
```

See Also

Other rnn cells:
`layer_rnn()`
`rnn_cell_lstm()`
`rnn_cell_simple()`

Other gru rnn layers:
`layer_gru()`

Other rnn layers:
`layer_bidirectional()`

layer_conv_lstm_1d()
layer_conv_lstm_2d()
layer_conv_lstm_3d()
layer_gru()
layer_lstm()
layer_rnn()
layer_simple_rnn()
layer_time_distributed()
rnn_cell_lstm()
rnn_cell_simple()
rnn_cells_stack()

Other layers:
Layer()
layer_activation()
layer_activation_elu()
layer_activation_leaky_relu()
layer_activation_parametric_relu()
layer_activation_relu()
layer_activation_softmax()
layer_activity_regularization()
layer_add()
layer_additive_attention()
layer_alpha_dropout()
layer_attention()
layer_average()
layer_average_pooling_1d()
layer_average_pooling_2d()
layer_average_pooling_3d()
layer_batch_normalization()
layer_bidirectional()
layer_category_encoding()
layer_center_crop()
layer_concatenate()
layer_conv_1d()
layer_conv_1d_transpose()
layer_conv_2d()
layer_conv_2d_transpose()
layer_conv_3d()
layer_conv_3d_transpose()
layer_conv_lstm_1d()
layer_conv_lstm_2d()
layer_conv_lstm_3d()
layer_cropping_1d()
layer_cropping_2d()
layer_cropping_3d()
layer_dense()
layer_depthwise_conv_1d()

layer_depthwise_conv_2d()
layer_discretization()
layer_dot()
layer_dropout()
layer_einsum_dense()
layer_embedding()
layer_feature_space()
layer_flatten()
layer_flax_module_wrapper()
layer_gaussian_dropout()
layer_gaussian_noise()
layer_global_average_pooling_1d()
layer_global_average_pooling_2d()
layer_global_average_pooling_3d()
layer_global_max_pooling_1d()
layer_global_max_pooling_2d()
layer_global_max_pooling_3d()
layer_group_normalization()
layer_group_query_attention()
layer_gru()
layer_hashed_crossing()
layer_hashing()
layer_identity()
layer_integer_lookup()
layer_jax_model_wrapper()
layer_lambda()
layer_layer_normalization()
layer_lstm()
layer_masking()
layer_max_pooling_1d()
layer_max_pooling_2d()
layer_max_pooling_3d()
layer_maximum()
layer_mel_spectrogram()
layer_minimum()
layer_multi_head_attention()
layer_multiply()
layer_normalization()
layer_permute()
layer_random_brightness()
layer_random_contrast()
layer_random_crop()
layer_random_flip()
layer_random_rotation()
layer_random_translation()
layer_random_zoom()
layer_repeat_vector()
layer_rescaling()

layer_reshape()
layer_resizing()
layer_rnn()
layer_separable_conv_1d()
layer_separable_conv_2d()
layer_simple_rnn()
layer_spatial_dropout_1d()
layer_spatial_dropout_2d()
layer_spatial_dropout_3d()
layer_spectral_normalization()
layer_string_lookup()
layer_subtract()
layer_text_vectorization()
layer_tfsm()
layer_time_distributed()
layer_torch_module_wrapper()
layer_unit_normalization()
layer_upsampling_1d()
layer_upsampling_2d()
layer_upsampling_3d()
layer_zero_padding_1d()
layer_zero_padding_2d()
layer_zero_padding_3d()
rnn_cell_lstm()
rnn_cell_simple()
rnn_cells_stack()

---

rnn_cell_lstm *Cell class for the LSTM layer.*

Description

This class processes one step within the whole time sequence input, whereas layer_lstm() processes the whole sequence.

Usage

```
rnn_cell_lstm(
  units,
  activation = "tanh",
  recurrent_activation = "sigmoid",
  use_bias = TRUE,
  kernel_initializer = "glorot_uniform",
  recurrent_initializer = "orthogonal",
  bias_initializer = "zeros",
  unit_forget_bias = TRUE,
  kernel_regularizer = NULL,
  recurrent_regularizer = NULL,
  bias_regularizer = NULL,
  kernel_constraint = NULL,
  recurrent_constraint = NULL,
  bias_constraint = NULL,
  dropout = 0,
  recurrent_dropout = 0,
  seed = NULL,
  ...
)
```

Arguments

`units`	Positive integer, dimensionality of the output space.
`activation`	Activation function to use. Default: hyperbolic tangent (`tanh`). If you pass `NULL`, no activation is applied (ie. "linear" activation: `a(x) = x`).
`recurrent_activation`	Activation function to use for the recurrent step. Default: sigmoid (`sigmoid`). If you pass `NULL`, no activation is applied (ie. "linear" activation: `a(x) = x`).
`use_bias`	Boolean, (default `TRUE`), whether the layer should use a bias vector.
`kernel_initializer`	Initializer for the `kernel` weights matrix, used for the linear transformation of the inputs. Default: `"glorot_uniform"`.
`recurrent_initializer`	Initializer for the `recurrent_kernel` weights matrix, used for the linear transformation of the recurrent state. Default: `"orthogonal"`.
`bias_initializer`	Initializer for the bias vector. Default: `"zeros"`.
`unit_forget_bias`	Boolean (default `TRUE`). If `TRUE`, add 1 to the bias of the forget gate at initialization. Setting it to `TRUE` will also force `bias_initializer="zeros"`. This is recommended in Jozefowicz et al.
`kernel_regularizer`	Regularizer function applied to the `kernel` weights matrix. Default: `NULL`.
`recurrent_regularizer`	Regularizer function applied to the `recurrent_kernel` weights matrix. Default: `NULL`.
`bias_regularizer`	Regularizer function applied to the bias vector. Default: `NULL`.
`kernel_constraint`	Constraint function applied to the `kernel` weights matrix. Default: `NULL`.
`recurrent_constraint`	Constraint function applied to the `recurrent_kernel` weights matrix. Default: `NULL`.
`bias_constraint`	Constraint function applied to the bias vector. Default: `NULL`.

`dropout`	Float between 0 and 1. Fraction of the units to drop for the linear transformation of the inputs. Default: 0.
`recurrent_dropout`	Float between 0 and 1. Fraction of the units to drop for the linear transformation of the recurrent state. Default: 0.
`seed`	Random seed for dropout.
`...`	For forward/backward compatability.

Value

A `Layer` instance, which is intended to be used with `layer_rnn()`.

Call Arguments

- `inputs`: A 2D tensor, with shape `(batch, features)`.
- `states`: A 2D tensor with shape `(batch, units)`, which is the state from the previous time step.
- `training`: Boolean indicating whether the layer should behave in training mode or in inference mode. Only relevant when `dropout` or `recurrent_dropout` is used.

Examples

```
inputs <- random_uniform(c(32, 10, 8))
output <- inputs |>
  layer_rnn(cell = rnn_cell_lstm(4))
shape(output)
```

```
## shape(32, 4)
```

```
rnn <- layer_rnn(cell = rnn_cell_lstm(4),
                 return_sequences = T,
                 return_state = T)
c(whole_sequence_output, ...final_state) %<-% rnn(inputs)
str(whole_sequence_output)
```

```
## <tf.Tensor: shape=(32, 10, 4), dtype=float32, numpy=. . .>
```

```
str(final_state)
```

```
## List of 2
##  $ :<tf.Tensor: shape=(32, 4), dtype=float32, numpy=. . .>
##  $ :<tf.Tensor: shape=(32, 4), dtype=float32, numpy=. . .>
```

See Also

Other rnn cells:
`layer_rnn()`
`rnn_cell_gru()`
`rnn_cell_simple()`

Other lstm rnn layers:
`layer_lstm()`

Other rnn layers:
`layer_bidirectional()`
`layer_conv_lstm_1d()`
`layer_conv_lstm_2d()`
`layer_conv_lstm_3d()`
`layer_gru()`
`layer_lstm()`
`layer_rnn()`
`layer_simple_rnn()`
`layer_time_distributed()`
`rnn_cell_gru()`
`rnn_cell_simple()`
`rnn_cells_stack()`

Other layers:
`Layer()`
`layer_activation()`
`layer_activation_elu()`
`layer_activation_leaky_relu()`
`layer_activation_parametric_relu()`
`layer_activation_relu()`
`layer_activation_softmax()`
`layer_activity_regularization()`
`layer_add()`
`layer_additive_attention()`
`layer_alpha_dropout()`
`layer_attention()`
`layer_average()`
`layer_average_pooling_1d()`
`layer_average_pooling_2d()`
`layer_average_pooling_3d()`
`layer_batch_normalization()`
`layer_bidirectional()`
`layer_category_encoding()`
`layer_center_crop()`
`layer_concatenate()`
`layer_conv_1d()`
`layer_conv_1d_transpose()`

layer_conv_2d()
layer_conv_2d_transpose()
layer_conv_3d()
layer_conv_3d_transpose()
layer_conv_lstm_1d()
layer_conv_lstm_2d()
layer_conv_lstm_3d()
layer_cropping_1d()
layer_cropping_2d()
layer_cropping_3d()
layer_dense()
layer_depthwise_conv_1d()
layer_depthwise_conv_2d()
layer_discretization()
layer_dot()
layer_dropout()
layer_einsum_dense()
layer_embedding()
layer_feature_space()
layer_flatten()
layer_flax_module_wrapper()
layer_gaussian_dropout()
layer_gaussian_noise()
layer_global_average_pooling_1d()
layer_global_average_pooling_2d()
layer_global_average_pooling_3d()
layer_global_max_pooling_1d()
layer_global_max_pooling_2d()
layer_global_max_pooling_3d()
layer_group_normalization()
layer_group_query_attention()
layer_gru()
layer_hashed_crossing()
layer_hashing()
layer_identity()
layer_integer_lookup()
layer_jax_model_wrapper()
layer_lambda()
layer_layer_normalization()
layer_lstm()
layer_masking()
layer_max_pooling_1d()
layer_max_pooling_2d()
layer_max_pooling_3d()
layer_maximum()
layer_mel_spectrogram()
layer_minimum()
layer_multi_head_attention()

layer_multiply()
layer_normalization()
layer_permute()
layer_random_brightness()
layer_random_contrast()
layer_random_crop()
layer_random_flip()
layer_random_rotation()
layer_random_translation()
layer_random_zoom()
layer_repeat_vector()
layer_rescaling()
layer_reshape()
layer_resizing()
layer_rnn()
layer_separable_conv_1d()
layer_separable_conv_2d()
layer_simple_rnn()
layer_spatial_dropout_1d()
layer_spatial_dropout_2d()
layer_spatial_dropout_3d()
layer_spectral_normalization()
layer_string_lookup()
layer_subtract()
layer_text_vectorization()
layer_tfsm()
layer_time_distributed()
layer_torch_module_wrapper()
layer_unit_normalization()
layer_upsampling_1d()
layer_upsampling_2d()
layer_upsampling_3d()
layer_zero_padding_1d()
layer_zero_padding_2d()
layer_zero_padding_3d()
rnn_cell_gru()
rnn_cell_simple()
rnn_cells_stack()

---

rnn_cell_simple *Cell class for SimpleRNN.*

Description

This class processes one step within the whole time sequence input, whereas `layer_simple_rnn()` processes the whole sequence.

Usage

```
rnn_cell_simple(
  units,
  activation = "tanh",
  use_bias = TRUE,
  kernel_initializer = "glorot_uniform",
  recurrent_initializer = "orthogonal",
  bias_initializer = "zeros",
  kernel_regularizer = NULL,
  recurrent_regularizer = NULL,
  bias_regularizer = NULL,
  kernel_constraint = NULL,
  recurrent_constraint = NULL,
  bias_constraint = NULL,
  dropout = 0,
  recurrent_dropout = 0,
  seed = NULL,
  ...
)
```

Arguments

units	Positive integer, dimensionality of the output space.
activation	Activation function to use. Default: hyperbolic tangent (tanh). If you pass NULL, no activation is applied (ie. "linear" activation: a(x) = x).
use_bias	Boolean, (default TRUE), whether the layer should use a bias vector.
kernel_initializer	Initializer for the kernel weights matrix, used for the linear transformation of the inputs. Default: "glorot_uniform".
recurrent_initializer	Initializer for the recurrent_kernel weights matrix, used for the linear transformation of the recurrent state. Default: "orthogonal".
bias_initializer	Initializer for the bias vector. Default: "zeros".
kernel_regularizer	Regularizer function applied to the kernel weights matrix. Default: NULL.
recurrent_regularizer	Regularizer function applied to the recurrent_kernel weights matrix. Default: NULL.
bias_regularizer	Regularizer function applied to the bias vector. Default: NULL.
kernel_constraint	Constraint function applied to the kernel weights matrix. Default: NULL.
recurrent_constraint	Constraint function applied to the recurrent_kernel weights matrix. Default: NULL.

`bias_constraint`	Constraint function applied to the bias vector. Default: `NULL`.
`dropout`	Float between 0 and 1. Fraction of the units to drop for the linear transformation of the inputs. Default: 0.
`recurrent_dropout`	Float between 0 and 1. Fraction of the units to drop for the linear transformation of the recurrent state. Default: 0.
`seed`	Random seed for dropout.
`...`	For forward/backward compatability.

Value

A `Layer` instance, which is intended to be used with `layer_rnn()`.

Call Arguments

- `sequence`: A 2D tensor, with shape `(batch, features)`.
- `states`: A 2D tensor with shape `(batch, units)`, which is the state from the previous time step.
- `training`: Python boolean indicating whether the layer should behave in training mode or in inference mode. Only relevant when `dropout` or `recurrent_dropout` is used.

Examples

```
inputs <- random_uniform(c(32, 10, 8))
rnn <- layer_rnn(cell = rnn_cell_simple(units = 4))
output <- rnn(inputs)  # The output has shape `(32, 4)`.
rnn <- layer_rnn(
    cell = rnn_cell_simple(units = 4),
    return_sequences=TRUE,
    return_state=TRUE
)
# whole_sequence_output has shape `(32, 10, 4)`.
# final_state has shape `(32, 4)`.
c(whole_sequence_output, final_state) %<-% rnn(inputs)
```

See Also

Other rnn cells:
`layer_rnn()`
`rnn_cell_gru()`
`rnn_cell_lstm()`

Other simple rnn layers:
`layer_simple_rnn()`

Other rnn layers:
`layer_bidirectional()`

layer_conv_lstm_1d()
layer_conv_lstm_2d()
layer_conv_lstm_3d()
layer_gru()
layer_lstm()
layer_rnn()
layer_simple_rnn()
layer_time_distributed()
rnn_cell_gru()
rnn_cell_lstm()
rnn_cells_stack()

Other layers:
Layer()
layer_activation()
layer_activation_elu()
layer_activation_leaky_relu()
layer_activation_parametric_relu()
layer_activation_relu()
layer_activation_softmax()
layer_activity_regularization()
layer_add()
layer_additive_attention()
layer_alpha_dropout()
layer_attention()
layer_average()
layer_average_pooling_1d()
layer_average_pooling_2d()
layer_average_pooling_3d()
layer_batch_normalization()
layer_bidirectional()
layer_category_encoding()
layer_center_crop()
layer_concatenate()
layer_conv_1d()
layer_conv_1d_transpose()
layer_conv_2d()
layer_conv_2d_transpose()
layer_conv_3d()
layer_conv_3d_transpose()
layer_conv_lstm_1d()
layer_conv_lstm_2d()
layer_conv_lstm_3d()
layer_cropping_1d()
layer_cropping_2d()
layer_cropping_3d()
layer_dense()
layer_depthwise_conv_1d()

layer_depthwise_conv_2d()
layer_discretization()
layer_dot()
layer_dropout()
layer_einsum_dense()
layer_embedding()
layer_feature_space()
layer_flatten()
layer_flax_module_wrapper()
layer_gaussian_dropout()
layer_gaussian_noise()
layer_global_average_pooling_1d()
layer_global_average_pooling_2d()
layer_global_average_pooling_3d()
layer_global_max_pooling_1d()
layer_global_max_pooling_2d()
layer_global_max_pooling_3d()
layer_group_normalization()
layer_group_query_attention()
layer_gru()
layer_hashed_crossing()
layer_hashing()
layer_identity()
layer_integer_lookup()
layer_jax_model_wrapper()
layer_lambda()
layer_layer_normalization()
layer_lstm()
layer_masking()
layer_max_pooling_1d()
layer_max_pooling_2d()
layer_max_pooling_3d()
layer_maximum()
layer_mel_spectrogram()
layer_minimum()
layer_multi_head_attention()
layer_multiply()
layer_normalization()
layer_permute()
layer_random_brightness()
layer_random_contrast()
layer_random_crop()
layer_random_flip()
layer_random_rotation()
layer_random_translation()
layer_random_zoom()
layer_repeat_vector()
layer_rescaling()

layer_reshape()
layer_resizing()
layer_rnn()
layer_separable_conv_1d()
layer_separable_conv_2d()
layer_simple_rnn()
layer_spatial_dropout_1d()
layer_spatial_dropout_2d()
layer_spatial_dropout_3d()
layer_spectral_normalization()
layer_string_lookup()
layer_subtract()
layer_text_vectorization()
layer_tfsm()
layer_time_distributed()
layer_torch_module_wrapper()
layer_unit_normalization()
layer_upsampling_1d()
layer_upsampling_2d()
layer_upsampling_3d()
layer_zero_padding_1d()
layer_zero_padding_2d()
layer_zero_padding_3d()
rnn_cell_gru()
rnn_cell_lstm()
rnn_cells_stack()

save_model — *Saves a model as a `.keras` file.*

Description

Saves a model as a `.keras` file.

Usage

```
save_model(model, filepath = NULL, overwrite = FALSE, ...)
```

Arguments

`model`	A keras model.
`filepath`	string, Path where to save the model. Must end in `.keras`.
`overwrite`	Whether we should overwrite any existing model at the target location, or instead ask the user via an interactive prompt.
`...`	For forward/backward compatability.

Value

If `filepath` is provided, then this function is called primarily for side effects, and `model` is returned invisibly. If `filepath` is not provided or `NULL`, then the serialized model is returned as an R raw vector.

Examples

```
model <- keras_model_sequential(input_shape = c(3)) |>
  layer_dense(5) |>
  layer_activation_softmax()

model |> save_model("model.keras")
loaded_model <- load_model("model.keras")
```

```
x <- random_uniform(c(10, 3))
stopifnot(all.equal(
  model |> predict(x),
  loaded_model |> predict(x)
))
```

The saved `.keras` file contains:

- The model's configuration (architecture)
- The model's weights
- The model's optimizer's state (if any)

Thus models can be reinstantiated in the exact same state.

```
zip::zip_list("model.keras")[, "filename"]
```

```
## [1] "metadata.json"    "config.json"      "model.weights.h5"
```

See Also

`load_model()`

Other saving and loading functions:
`export_savedmodel.keras.src.models.model.Model()`
`layer_tfsm()`
`load_model()`
`load_model_weights()`
`register_keras_serializable()`
`save_model_config()`
`save_model_weights()`
`with_custom_object_scope()`

save_model_config *Save and load model configuration as JSON*

Description

Save and re-load models configurations as JSON. Note that the representation does not include the weights, only the architecture.

Usage

```
save_model_config(model, filepath = NULL, overwrite = FALSE)

load_model_config(filepath, custom_objects = NULL)
```

Arguments

`model`	Model object to save
`filepath`	path to json file with the model config.
`overwrite`	Whether we should overwrite any existing model configuration json at `filepath`, or instead ask the user via an interactive prompt.
`custom_objects`	Optional named list mapping names to custom classes or functions to be considered during deserialization.

Details

Note: `save_model_config()` serializes the model to JSON using `serialize_keras_object()`, not `get_config()`. `serialize_keras_object()` returns a superset of `get_config()`, with additional information needed to create the class object needed to restore the model. See example for how to extract the `get_config()` value from a saved model.

Value

This is called primarily for side effects. `model` is returned, invisibly, to enable usage with the pipe.

Example

```
model <- keras_model_sequential(input_shape = 10) |> layer_dense(10)
file <- tempfile("model-config-", fileext = ".json")
save_model_config(model, file)

# load a new model instance with the same architecture but different weights
model2 <- load_model_config(file)

stopifnot(exprs = {
  all.equal(get_config(model), get_config(model2))

  # To extract the `get_config()` value from a saved model config:
```

```
  all.equal(
      get_config(model),
      structure(jsonlite::read_json(file)$config,
                "__class__" = keras_model_sequential()$`__class__`)
  )
})
```

See Also

Other saving and loading functions:
export_savedmodel.keras.src.models.model.Model()
layer_tfsm()
load_model()
load_model_weights()
register_keras_serializable()
save_model()
save_model_weights()
with_custom_object_scope()

`save_model_weights` *Saves all layer weights to a* `.weights.h5` *file.*

Description

Saves all layer weights to a `.weights.h5` file.

Usage

```
save_model_weights(model, filepath, overwrite = FALSE)
```

Arguments

`model`	A keras Model object
`filepath`	string. Path where to save the model. Must end in `.weights.h5`.
`overwrite`	Whether we should overwrite any existing model at the target location, or instead ask the user via an interactive prompt.

Value

This is called primarily for side effects. `model` is returned, invisibly, to enable usage with the pipe.

See Also

- https://keras.io/api/models/model_saving_apis/weights_saving_and_loading#saveweights-method

Other saving and loading functions:
export_savedmodel.keras.src.models.model.Model()
layer_tfsm()
load_model()
load_model_weights()
register_keras_serializable()
save_model()
save_model_config()
with_custom_object_scope()

---

serialize_keras_object *Retrieve the full config by serializing the Keras object.*

Description

`serialize_keras_object()` serializes a Keras object to a named list that represents the object, and is a reciprocal function of `deserialize_keras_object()`. See `deserialize_keras_object()` for more information about the full config format.

Usage

```
serialize_keras_object(obj)
```

Arguments

obj	the Keras object to serialize.

Value

A named list that represents the object config. The config is expected to contain simple types only, and can be saved as json. The object can be deserialized from the config via `deserialize_keras_object()`.

See Also

- https://keras.io/api/models/model_saving_apis/serialization_utils#serializekerasobject-functio

Other serialization utilities:
deserialize_keras_object()
get_custom_objects()
get_registered_name()
get_registered_object()
register_keras_serializable()
with_custom_object_scope()

set_random_seed — *Sets all random seeds (Python, NumPy, and backend framework, e.g. TF).*

Description

You can use this utility to make almost any Keras program fully deterministic. Some limitations apply in cases where network communications are involved (e.g. parameter server distribution), which creates additional sources of randomness, or when certain non-deterministic cuDNN ops are involved.

This sets:

- the R session seed: `set.seed()`
- the Python session seed: `import random; random.seed(seed)`
- the Python NumPy seed: `import numpy; numpy.random.seed(seed)`
- the TensorFlow seed: `tf$random$set_seed(seed)` (only if TF is installed)
- The Torch seed: `import("torch")$manual_seed(seed)` (only if the backend is torch)
- and disables Python hash randomization.

Note that the TensorFlow seed is set even if you're not using TensorFlow as your backend framework, since many workflows leverage `tf$data` pipelines (which feature random shuffling). Likewise many workflows might leverage NumPy APIs.

Usage

```
set_random_seed(seed)
```

Arguments

`seed`	Integer, the random seed to use.

Value

No return value, called for side effects.

See Also

- https://keras.io/api/utils/python_utils#setrandomseed-function

Other utils:
`audio_dataset_from_directory()`
`clear_session()`
`config_disable_interactive_logging()`
`config_disable_traceback_filtering()`
`config_enable_interactive_logging()`
`config_enable_traceback_filtering()`
`config_is_interactive_logging_enabled()`

config_is_traceback_filtering_enabled()
get_file()
get_source_inputs()
image_array_save()
image_dataset_from_directory()
image_from_array()
image_load()
image_smart_resize()
image_to_array()
layer_feature_space()
normalize()
pack_x_y_sample_weight()
pad_sequences()
split_dataset()
text_dataset_from_directory()
timeseries_dataset_from_array()
to_categorical()
unpack_x_y_sample_weight()
zip_lists()

---

shape *Tensor shape utility*

Description

This function can be used to create or get the shape of an object.

Usage

```
shape(...)

## S3 method for class 'keras_shape'
format(x, ..., prefix = TRUE)

## S3 method for class 'keras_shape'
print(x, ...)

## S3 method for class 'keras_shape'
x[...]

## S3 method for class 'keras_shape'
as.integer(x, ...)

## S3 method for class 'keras_shape'
as.list(x, ...)
```

Arguments

`...`	A shape specification. Numerics, `NULL` and tensors are valid. `NULL`, `NA`, and `-1L` can be used to specify an unspecified dim size. Tensors are dispatched to `op_shape()` to extract the tensor shape. Values wrapped in `I()` are used asis (see examples). All other objects are coerced via `as.integer()`.
`x`	A 'keras_shape' object
`prefix`	Whether to format the shape object with a prefix. Defaults to `"shape"`.

Value

A list with a `"keras_shape"` class attribute. Each element of the list will be either a) `NULL`, b) an integer or c) a scalar integer tensor (e.g., when supplied a TF tensor with a unspecified dimension in a function being traced).

Examples

```
shape(1, 2, 3)
```

```
## shape(1, 2, 3)
```

3 ways to specify an unknown dimension

```
shape(NA,   2, 3)
shape(NULL, 2, 3)
shape(-1,   2, 3)
```

```
## shape(NA, 2, 3)
## shape(NA, 2, 3)
## shape(NA, 2, 3)
```

Most functions that take a 'shape' argument also coerce with `shape()`

```
layer_input(c(1, 2, 3))
layer_input(shape(1, 2, 3))
```

```
## <KerasTensor shape=(None, 1, 2, 3), dtype=float32, sparse=None, name=keras_tensor>
## <KerasTensor shape=(None, 1, 2, 3), dtype=float32, sparse=None, name=keras_tensor_1>
```

You can also use `shape()` to get the shape of a tensor (excepting scalar integer tensors).

```
symbolic_tensor <- layer_input(shape(1, 2, 3))
shape(symbolic_tensor)
```

```
## shape(NA, 1, 2, 3)
```

```
eager_tensor <- op_ones(c(1,2,3))
shape(eager_tensor)
```

```
## shape(1, 2, 3)
```

```
op_shape(eager_tensor)
```

```
## shape(1, 2, 3)
```

Combine or expand shapes

```
shape(symbolic_tensor, 4)
```

```
## shape(NA, 1, 2, 3, 4)
```

```
shape(5, symbolic_tensor, 4)
```

```
## shape(5, NA, 1, 2, 3, 4)
```

Scalar integer tensors are treated as axis values. These are most commonly encountered when tracing a function in graph mode, where an axis size might be unknown.

```
tfn <- tensorflow::tf_function(function(x) {
  print(op_shape(x))
  x
},
input_signature = list(tensorflow::tf$TensorSpec(shape(1, NA, 3))))
invisible(tfn(op_ones(shape(1, 2, 3))))
```

```
## shape(1, Tensor("strided_slice:0", shape=(), dtype=int32), 3)
```

A useful pattern is to unpack the `shape()` with `%<-%`, like this:

```
c(batch_size, seq_len, channels) %<-% shape(x)
```

```
echo_print <- function(x) {
  message("> ", deparse(substitute(x)));
  if(!is.null(x)) print(x)
}
tfn <- tensorflow::tf_function(function(x) {
  c(axis1, axis2, axis3) %<-% shape(x)
  echo_print(str(list(axis1 = axis1, axis2 = axis2, axis3 = axis3)))

  echo_print(shape(axis1))               # use axis1 tensor as axis value
  echo_print(shape(axis1, axis2, axis3)) # use axis1 tensor as axis value

  # use shape() to compose a new shape, e.g., in multihead attention
  n_heads <- 4
  echo_print(shape(axis1, axis2, n_heads, axis3/n_heads))

  x
},
input_signature = list(tensorflow::tf$TensorSpec(shape(NA, 4, 16))))
invisible(tfn(op_ones(shape(2, 4, 16))))

## > str(list(axis1 = axis1, axis2 = axis2, axis3 = axis3))

## List of 3
##  $ axis1:<tf.Tensor 'strided_slice:0' shape=() dtype=int32>
##  $ axis2: int 4
##  $ axis3: int 16


## > shape(axis1)

## shape(Tensor("strided_slice:0", shape=(), dtype=int32))


## > shape(axis1, axis2, axis3)

## shape(Tensor("strided_slice:0", shape=(), dtype=int32), 4, 16)


## > shape(axis1, axis2, n_heads, axis3/n_heads)

## shape(Tensor("strided_slice:0", shape=(), dtype=int32), 4, 4, 4)
```

If you want to resolve the shape of a tensor that can potentially be a scalar integer, you can wrap the tensor in `I()`, or use `op_shape()`.

```
(x <- op_convert_to_tensor(2L))
```

```
## tf.Tensor(2, shape=(), dtype=int32)


# by default, shape() treats scalar integer tensors as axis values
shape(x)

## shape(tf.Tensor(2, shape=(), dtype=int32))


# to access the shape of a scalar integer,
# call `op_shape()`, or protect with `I()`
op_shape(x)

## shape()


shape(I(x))

## shape()
```

See Also

op_shape()

---

split_dataset *Splits a dataset into a left half and a right half (e.g. train / test).*

Description

Splits a dataset into a left half and a right half (e.g. train / test).

Usage

```
split_dataset(
  dataset,
  left_size = NULL,
  right_size = NULL,
  shuffle = FALSE,
  seed = NULL
)
```

Arguments

dataset	A tf$data$Dataset, a torch$utils$data$Dataset object, or a list of arrays with the same length.
left_size	If float (in the range [0, 1]), it signifies the fraction of the data to pack in the left dataset. If integer, it signifies the number of samples to pack in the left dataset. If NULL, defaults to the complement to right_size. Defaults to NULL.
right_size	If float (in the range [0, 1]), it signifies the fraction of the data to pack in the right dataset. If integer, it signifies the number of samples to pack in the right dataset. If NULL, defaults to the complement to left_size. Defaults to NULL.
shuffle	Boolean, whether to shuffle the data before splitting it.
seed	A random seed for shuffling.

Value

A list of two tf$data$Dataset objects: the left and right splits.

Examples

```
data <- random_uniform(c(1000, 4))
c(left_ds, right_ds) %<-% split_dataset(list(data$numpy()), left_size = 0.8)
left_ds$cardinality()

## tf.Tensor(800, shape=(), dtype=int64)


right_ds$cardinality()

## tf.Tensor(200, shape=(), dtype=int64)
```

See Also

- https://keras.io/api/utils/python_utils#splitdataset-function

Other dataset utils:
audio_dataset_from_directory()
image_dataset_from_directory()
text_dataset_from_directory()
timeseries_dataset_from_array()

Other utils:
audio_dataset_from_directory()
clear_session()
config_disable_interactive_logging()
config_disable_traceback_filtering()
config_enable_interactive_logging()
config_enable_traceback_filtering()

config_is_interactive_logging_enabled()
config_is_traceback_filtering_enabled()
get_file()
get_source_inputs()
image_array_save()
image_dataset_from_directory()
image_from_array()
image_load()
image_smart_resize()
image_to_array()
layer_feature_space()
normalize()
pack_x_y_sample_weight()
pad_sequences()
set_random_seed()
text_dataset_from_directory()
timeseries_dataset_from_array()
to_categorical()
unpack_x_y_sample_weight()
zip_lists()

`summary.keras.src.models.model.Model`

Print a summary of a Keras Model

Description

Print a summary of a Keras Model

Usage

```
## S3 method for class 'keras.src.models.model.Model'
summary(object, ...)

## S3 method for class 'keras.src.models.model.Model'
format(
  x,
  line_length = getOption("width"),
  positions = NULL,
  expand_nested = FALSE,
  show_trainable = NA,
  ...,
  layer_range = NULL,
  compact = TRUE
)
```

```
## S3 method for class 'keras.src.models.model.Model'
print(x, ...)
```

Arguments

`object, x`	Keras model instance
`...`	for `summary()` and `print()`, passed on to `format()`. For `format()`, passed on to `model$summary()`.
`line_length`	Total length of printed lines
`positions`	Relative or absolute positions of log elements in each line. If not provided, defaults to `c(0.33, 0.55, 0.67, 1.0)`.
`expand_nested`	Whether to expand the nested models. If not provided, defaults to `FALSE`.
`show_trainable`	Whether to show if a layer is trainable. If not provided, defaults to `FALSE`.
`layer_range`	a list, tuple, or vector of 2 strings, which is the starting layer name and ending layer name (both inclusive) indicating the range of layers to be printed in summary. It also accepts regex patterns instead of exact name. In such case, start predicate will be the first element it matches to `layer_range[[1]]` and the end predicate will be the last element it matches to `layer_range[[1]]`. By default `NULL` which considers all layers of model.
`compact`	Whether to remove white-space only lines from the model summary. (Default `TRUE`)

Value

`format()` returns a length 1 character vector. `print()` returns the model object invisibly. `summary()` returns the output of `format()` invisibly after printing it.

Enabling color output in Knitr (RMarkdown, Quarto)

In order to enable color output in a quarto or rmarkdown document with an html output format (include revealjs presentations), then you will need to do the following in a setup chunk:

~~~
```{r setup, include = FALSE}
options(cli.num_colors = 256)
fansi::set_knit_hooks(knitr::knit_hooks)
options(width = 75) # adjust as needed for format
```
~~~

See Also

Other model functions:
`get_config()`
`get_layer()`
`keras_model()`
`keras_model_sequential()`
`pop_layer()`

test_on_batch *Test the model on a single batch of samples.*

Description

Test the model on a single batch of samples.

Usage

```
test_on_batch(object, x, y = NULL, sample_weight = NULL, ...)
```

Arguments

object	Keras model object
x	Input data. Must be array-like.
y	Target data. Must be array-like.
sample_weight	Optional array of the same length as x, containing weights to apply to the model's loss for each sample. In the case of temporal data, you can pass a 2D array with shape (samples, sequence_length), to apply a different weight to every timestep of every sample.
...	for forward/backward compatability

Value

A scalar loss value (when no metrics), or a named list of loss and metric values (if there are metrics).

See Also

- https://keras.io/api/models/model_training_apis#testonbatch-method

Other model training:
compile.keras.src.models.model.Model()
evaluate.keras.src.models.model.Model()
predict.keras.src.models.model.Model()
predict_on_batch()
train_on_batch()

text_dataset_from_directory

Generates a `tf.data.Dataset` *from text files in a directory.*

Description

If your directory structure is:

```
main_directory/
...class_a/
......a_text_1.txt
......a_text_2.txt
...class_b/
......b_text_1.txt
......b_text_2.txt
```

Then calling `text_dataset_from_directory(main_directory, labels='inferred')` will return a `tf.data.Dataset` that yields batches of texts from the subdirectories `class_a` and `class_b`, together with labels 0 and 1 (0 corresponding to `class_a` and 1 corresponding to `class_b`).

Only `.txt` files are supported at this time.

Usage

```
text_dataset_from_directory(
  directory,
  labels = "inferred",
  label_mode = "int",
  class_names = NULL,
  batch_size = 32L,
  max_length = NULL,
  shuffle = TRUE,
  seed = NULL,
  validation_split = NULL,
  subset = NULL,
  follow_links = FALSE,
  verbose = TRUE
)
```

Arguments

`directory`	Directory where the data is located. If `labels` is `"inferred"`, it should contain subdirectories, each containing text files for a class. Otherwise, the directory structure is ignored.
`labels`	Either `"inferred"` (labels are generated from the directory structure), `NULL` (no labels), or a list/tuple of integer labels of the same size as the number of text files found in the directory. Labels should be sorted according to the alphanumeric order of the text file paths (obtained via `os.walk(directory)` in Python).

`label_mode`	String describing the encoding of `labels`. Options are: • `"int"`: means that the labels are encoded as integers (e.g. for `sparse_categorical_crossentropy` loss). • `"categorical"` means that the labels are encoded as a categorical vector (e.g. for `categorical_crossentropy` loss). • `"binary"` means that the labels (there can be only 2) are encoded as `float32` scalars with values 0 or 1 (e.g. for `binary_crossentropy`). • `NULL` (no labels).
`class_names`	Only valid if `"labels"` is `"inferred"`. This is the explicit list of class names (must match names of subdirectories). Used to control the order of the classes (otherwise alphanumerical order is used).
`batch_size`	Size of the batches of data. Defaults to 32. If `NULL`, the data will not be batched (the dataset will yield individual samples).
`max_length`	Maximum size of a text string. Texts longer than this will be truncated to `max_length`.
`shuffle`	Whether to shuffle the data. Defaults to `TRUE`. If set to `FALSE`, sorts the data in alphanumeric order.
`seed`	Optional random seed for shuffling and transformations.
`validation_split`	Optional float between 0 and 1, fraction of data to reserve for validation.
`subset`	Subset of the data to return. One of `"training"`, `"validation"` or `"both"`. Only used if `validation_split` is set. When `subset="both"`, the utility returns a tuple of two datasets (the training and validation datasets respectively).
`follow_links`	Whether to visits subdirectories pointed to by symlinks. Defaults to `FALSE`.
`verbose`	Whether to display number information on classes and number of files found. Defaults to `TRUE`.

Value

A `tf.data.Dataset` object.

- If `label_mode` is `NULL`, it yields `string` tensors of shape `(batch_size,)`, containing the contents of a batch of text files.
- Otherwise, it yields a tuple `(texts, labels)`, where `texts` has shape `(batch_size,)` and `labels` follows the format described below.

Rules regarding labels format:

- if `label_mode` is `int`, the labels are an `int32` tensor of shape `(batch_size,)`.
- if `label_mode` is `binary`, the labels are a `float32` tensor of 1s and 0s of shape `(batch_size, 1)`.
- if `label_mode` is `categorical`, the labels are a `float32` tensor of shape `(batch_size, num_classes)`, representing a one-hot encoding of the class index.

See Also

- https://keras.io/api/data_loading/text#textdatasetfromdirectory-function

Other dataset utils:
audio_dataset_from_directory()
image_dataset_from_directory()
split_dataset()
timeseries_dataset_from_array()

Other utils:
audio_dataset_from_directory()
clear_session()
config_disable_interactive_logging()
config_disable_traceback_filtering()
config_enable_interactive_logging()
config_enable_traceback_filtering()
config_is_interactive_logging_enabled()
config_is_traceback_filtering_enabled()
get_file()
get_source_inputs()
image_array_save()
image_dataset_from_directory()
image_from_array()
image_load()
image_smart_resize()
image_to_array()
layer_feature_space()
normalize()
pack_x_y_sample_weight()
pad_sequences()
set_random_seed()
split_dataset()
timeseries_dataset_from_array()
to_categorical()
unpack_x_y_sample_weight()
zip_lists()

Other preprocessing:
image_dataset_from_directory()
image_smart_resize()
timeseries_dataset_from_array()

---

timeseries_dataset_from_array

Creates a dataset of sliding windows over a timeseries provided as array.

Description

This function takes in a sequence of data-points gathered at equal intervals, along with time series parameters such as length of the sequences/windows, spacing between two sequence/windows, etc., to produce batches of timeseries inputs and targets.

Usage

```
timeseries_dataset_from_array(
  data,
  targets,
  sequence_length,
  sequence_stride = 1L,
  sampling_rate = 1L,
  batch_size = 128L,
  shuffle = FALSE,
  seed = NULL,
  start_index = NULL,
  end_index = NULL
)
```

Arguments

`data`	array or eager tensor containing consecutive data points (timesteps). The first dimension is expected to be the time dimension.
`targets`	Targets corresponding to timesteps in `data`. `targets[i]` should be the target corresponding to the window that starts at index `i` (see example 2 below). Pass `NULL` if you don't have target data (in this case the dataset will only yield the input data).
`sequence_length`	Length of the output sequences (in number of timesteps).
`sequence_stride`	Period between successive output sequences. For stride `s`, output samples would start at index `data[i]`, `data[i + s]`, `data[i + 2 * s]`, etc.
`sampling_rate`	Period between successive individual timesteps within sequences. For rate `r`, timesteps `data[i], data[i + r], ... data[i + sequence_length]` are used for creating a sample sequence.
`batch_size`	Number of timeseries samples in each batch (except maybe the last one). If `NULL`, the data will not be batched (the dataset will yield individual samples).
`shuffle`	Whether to shuffle output samples, or instead draw them in chronological order.
`seed`	Optional int; random seed for shuffling.
`start_index`	Optional int; data points earlier (exclusive) than `start_index` will not be used in the output sequences. This is useful to reserve part of the data for test or validation.
`end_index`	Optional int; data points later (exclusive) than `end_index` will not be used in the output sequences. This is useful to reserve part of the data for test or validation.

Value

A tf$data$Dataset instance. If targets was passed, the dataset yields list (batch_of_sequences, batch_of_targets). If not, the dataset yields only batch_of_sequences.

Example 1:

Consider indices [0, 1, ... 98]. With sequence_length=10, sampling_rate=2, sequence_stride=3, shuffle=FALSE, the dataset will yield batches of sequences composed of the following indices:

```
First sequence:  [0  2  4  6  8 10 12 14 16 18]
Second sequence: [3  5  7  9 11 13 15 17 19 21]
Third sequence:  [6  8 10 12 14 16 18 20 22 24]
...
Last sequence:   [78 80 82 84 86 88 90 92 94 96]
```

In this case the last 2 data points are discarded since no full sequence can be generated to include them (the next sequence would have started at index 81, and thus its last step would have gone over 98).

Example 2: Temporal regression.

Consider an array data of scalar values, of shape (steps,). To generate a dataset that uses the past 10 timesteps to predict the next timestep, you would use:

```
data <- op_array(1:20)
input_data <- data[1:10]
targets <- data[11:20]
dataset <- timeseries_dataset_from_array(
  input_data, targets, sequence_length=10)
iter <- reticulate::as_iterator(dataset)
reticulate::iter_next(iter)

## [[1]]
## tf.Tensor([[ 1  2  3  4  5  6  7  8  9 10]], shape=(1, 10), dtype=int32)
##
## [[2]]
## tf.Tensor([11], shape=(1), dtype=int32)
```

Example 3: Temporal regression for many-to-many architectures.

Consider two arrays of scalar values X and Y, both of shape (100,). The resulting dataset should consist samples with 20 timestamps each. The samples should not overlap. To generate a dataset that uses the current timestamp to predict the corresponding target timestep, you would use:

```
X <- op_array(1:100)
Y <- X*2

sample_length <- 20
input_dataset <- timeseries_dataset_from_array(
    X, NULL, sequence_length=sample_length, sequence_stride=sample_length)
```

```
target_dataset <- timeseries_dataset_from_array(
    Y, NULL, sequence_length=sample_length, sequence_stride=sample_length)


inputs <- reticulate::as_iterator(input_dataset) %>% reticulate::iter_next()
targets <- reticulate::as_iterator(target_dataset) %>% reticulate::iter_next()
```

See Also

- https://keras.io/api/data_loading/timeseries#timeseriesdatasetfromarray-function

Other dataset utils:
audio_dataset_from_directory()
image_dataset_from_directory()
split_dataset()
text_dataset_from_directory()

Other utils:
audio_dataset_from_directory()
clear_session()
config_disable_interactive_logging()
config_disable_traceback_filtering()
config_enable_interactive_logging()
config_enable_traceback_filtering()
config_is_interactive_logging_enabled()
config_is_traceback_filtering_enabled()
get_file()
get_source_inputs()
image_array_save()
image_dataset_from_directory()
image_from_array()
image_load()
image_smart_resize()
image_to_array()
layer_feature_space()
normalize()
pack_x_y_sample_weight()
pad_sequences()
set_random_seed()
split_dataset()
text_dataset_from_directory()
to_categorical()
unpack_x_y_sample_weight()
zip_lists()

Other preprocessing:
image_dataset_from_directory()
image_smart_resize()

text_dataset_from_directory()

to_categorical — *Converts a class vector (integers) to binary class matrix.*

Description

E.g. for use with loss_categorical_crossentropy().

Usage

```
to_categorical(x, num_classes = NULL)
```

Arguments

x	Array-like with class values to be converted into a matrix (integers from 0 to num_classes - 1). R factors are coerced to integer and offset to be 0-based, i.e., as.integer(x) - 1L.
num_classes	Total number of classes. If NULL, this would be inferred as max(x) + 1. Defaults to NULL.

Value

A binary matrix representation of the input as an R array. The class axis is placed last.

Examples

```
a <- to_categorical(c(0, 1, 2, 3), num_classes=4)
print(a)

##      [,1] [,2] [,3] [,4]
## [1,]    1    0    0    0
## [2,]    0    1    0    0
## [3,]    0    0    1    0
## [4,]    0    0    0    1


b <- array(c(.9, .04, .03, .03,
              .3, .45, .15, .13,
              .04, .01, .94, .05,
              .12, .21, .5, .17),
              dim = c(4, 4))
loss <- op_categorical_crossentropy(a, b)
loss

## tf.Tensor([0.41284522 0.45601739 0.54430155 0.80437282], shape=(4), dtype=float64)
```

```
loss <- op_categorical_crossentropy(a, a)
loss
```

```
## tf.Tensor([1.00000005e-07 1.00000005e-07 1.00000005e-07 1.00000005e-07], shape=(4), dtype=float64)
```

See Also

- op_one_hot(), which does the same operation as to_categorical(), but operating on tensors.
- loss_sparse_categorical_crossentropy(), which can accept labels (y_true) as an integer vector, instead of as a dense one-hot matrix.
- https://keras.io/api/utils/python_utils#tocategorical-function

Other numerical utils:
normalize()

Other utils:
audio_dataset_from_directory()
clear_session()
config_disable_interactive_logging()
config_disable_traceback_filtering()
config_enable_interactive_logging()
config_enable_traceback_filtering()
config_is_interactive_logging_enabled()
config_is_traceback_filtering_enabled()
get_file()
get_source_inputs()
image_array_save()
image_dataset_from_directory()
image_from_array()
image_load()
image_smart_resize()
image_to_array()
layer_feature_space()
normalize()
pack_x_y_sample_weight()
pad_sequences()
set_random_seed()
split_dataset()
text_dataset_from_directory()
timeseries_dataset_from_array()
unpack_x_y_sample_weight()
zip_lists()

train_on_batch — *Runs a single gradient update on a single batch of data.*

Description

Runs a single gradient update on a single batch of data.

Usage

```
train_on_batch(object, x, y = NULL, sample_weight = NULL, class_weight = NULL)
```

Arguments

`object`	Keras model object
`x`	Input data. Must be array-like.
`y`	Target data. Must be array-like.
`sample_weight`	Optional array of the same length as x, containing weights to apply to the model's loss for each sample. In the case of temporal data, you can pass a 2D array with shape `(samples, sequence_length)`, to apply a different weight to every timestep of every sample.
`class_weight`	Optional named list mapping class indices (integers, 0-based) to a weight (float) to apply to the model's loss for the samples from this class during training. This can be useful to tell the model to "pay more attention" to samples from an under-represented class. When `class_weight` is specified and targets have a rank of 2 or greater, either `y` must be one-hot encoded, or an explicit final dimension of 1 must be included for sparse class labels.

Value

A scalar loss value (when no metrics), or a named list of loss and metric values (if there are metrics). The property `model$metrics_names` will give you the display labels for the scalar outputs.

See Also

- `https://keras.io/api/models/model_training_apis#trainonbatch-method`

Other model training:
`compile.keras.src.models.model.Model()`
`evaluate.keras.src.models.model.Model()`
`predict.keras.src.models.model.Model()`
`predict_on_batch()`
`test_on_batch()`

use_backend *Configure a Keras backend*

Description

Configure a Keras backend

Usage

```
use_backend(backend)
```

Arguments

backend	string, can be "tensorflow", "jax", "numpy", or "torch".

Details

These functions allow configuring which backend keras will use. Note that only one backend can be configured at a time.

The function should be called after `library(keras3)` and before calling other functions within the package (see below for an example).

There is experimental support for changing the backend after keras has initialized. using `config_set_backend()`.

```
library(keras3)
use_backend("tensorflow")
```

Value

Called primarily for side effects. Returns the provided backend, invisibly.

with_custom_object_scope *Provide a scope with mappings of names to custom objects*

Description

Provide a scope with mappings of names to custom objects

Usage

```
with_custom_object_scope(objects, expr)
```

Arguments

objects	Named list of objects
expr	Expression to evaluate

Details

There are many elements of Keras models that can be customized with user objects (e.g. losses, metrics, regularizers, etc.). When loading saved models that use these functions you typically need to explicitly map names to user objects via the `custom_objects` parameter.

The `with_custom_object_scope()` function provides an alternative that lets you create a named alias for a user object that applies to an entire block of code, and is automatically recognized when loading saved models.

Value

The result from evaluating `expr` within the custom object scope.

Examples

```
# define custom metric
metric_top_3_categorical_accuracy <-
  custom_metric("top_3_categorical_accuracy", function(y_true, y_pred) {
    metric_top_k_categorical_accuracy(y_true, y_pred, k = 3)
  })

with_custom_object_scope(c(top_k_acc = sparse_top_k_cat_acc), {

  # ...define model...

  # compile model (refer to "top_k_acc" by name)
  model |> compile(
    loss = "binary_crossentropy",
    optimizer = optimizer_nadam(),
    metrics = c("top_k_acc")
  )

  # save the model
  model |> save_model("my_model.keras")

  # loading the model within the custom object scope doesn't
  # require explicitly providing the custom_object
  reloaded_model <- load_model("my_model.keras")
})
```

See Also

Other saving and loading functions:
`export_savedmodel.keras.src.models.model.Model()`
`layer_tfsm()`
`load_model()`
`load_model_weights()`
`register_keras_serializable()`
`save_model()`

save_model_config()
save_model_weights()

Other serialization utilities:
deserialize_keras_object()
get_custom_objects()
get_registered_name()
get_registered_object()
register_keras_serializable()
serialize_keras_object()

zip_lists *Zip lists*

Description

This is conceptually similar to `zip()` in Python, or R functions `purrr::transpose()` and `data.table::transpose()` (albeit, accepting elements in `...` instead of a single list), with one crucial difference: if the provided objects are named, then matching is done by names, not positions.

Usage

```
zip_lists(...)
```

Arguments

`...`	R lists or atomic vectors, optionally named.

Details

All arguments supplied must be of the same length. If positional matching is required, then all arguments provided must be unnamed. If matching by names, then all arguments must have the same set of names, but they can be in different orders.

Value

A inverted list

See Also

Other data utils:
pack_x_y_sample_weight()
unpack_x_y_sample_weight()

Other utils:
audio_dataset_from_directory()

clear_session()
config_disable_interactive_logging()
config_disable_traceback_filtering()
config_enable_interactive_logging()
config_enable_traceback_filtering()
config_is_interactive_logging_enabled()
config_is_traceback_filtering_enabled()
get_file()
get_source_inputs()
image_array_save()
image_dataset_from_directory()
image_from_array()
image_load()
image_smart_resize()
image_to_array()
layer_feature_space()
normalize()
pack_x_y_sample_weight()
pad_sequences()
set_random_seed()
split_dataset()
text_dataset_from_directory()
timeseries_dataset_from_array()
to_categorical()
unpack_x_y_sample_weight()

Examples

```
gradients <- list("grad_for_wt_1", "grad_for_wt_2", "grad_for_wt_3")
weights <- list("weight_1", "weight_2", "weight_3")
str(zip_lists(gradients, weights))
str(zip_lists(gradient = gradients, weight = weights))

names(gradients) <- names(weights) <- paste0("layer_", 1:3)
str(zip_lists(gradients, weights[c(3, 1, 2)]))

names(gradients) <- paste0("gradient_", 1:3)
try(zip_lists(gradients, weights)) # error, names don't match
# call unname directly for positional matching
str(zip_lists(unname(gradients), unname(weights)))
```

Index