Package ‘keras’

August 21, 2021

Type Package

Title R Interface to ‘Keras’

Version 2.6.0

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.rstudio.com

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

Depends R (>= 3.2)

Imports generics (>= 0.0.1), reticulate (>= 1.10), tensorflow (>= 2.6.0), tfruns (>= 1.0), magrittr, zeallot, glue, methods, R6, ellipsis, rlang

Suggests ggplot2, testthat (>= 2.1.0), knitr, rmarkdown, tfdatasets, jpeg

RoxygenNote 7.1.1

VignetteBuilder knitr

NeedsCompilation no

Author Tomasz Kalinowski [ctb, cph, cre],
       Daniel Falbel [ctb, cph],
       JJ Allaire [aut, cph],
       François Chollet [aut, cph],
       RStudio [ctb, cph, fnd],
       Google [ctb, 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.kalinowski@rstudio.com>
R topics documented:

keras-package ........................................ 10
activation_relu ........................................ 11
adapt .................................................. 13
application_dense ..................................... 14
application_inception_resnet_v2 .................... 15
application_inception_v3 ......................... 17
application_mobilenet ................................. 18
application_mobilenet_v2 ......................... 20
application_nas ........................................ 21
application_resnet50 ................................. 23
application_vgg ......................................... 25
application_xception .................................. 26
backend ................................................ 28
bidirectional .......................................... 28
callback_csv_logger .................................. 29
callback_early_stopping ........................... 30
callback_lambda ...................................... 31
callback_learning_rate_scheduler .................. 32
callback_model_checkpoint .......................... 33
callback_progbar_logger ............................. 34
callback_reduce_lr_on_plateau ...................... 35
callback_remote_monitor ............................. 36
callback_tensorboard .................................. 37
callback.terminate_on_naan .......................... 38
clone_model .......................................... 39
compile.keras.engine.training.Model .............. 39
custom_metric ........................................ 45
dataset_boston_housing ............................... 46
dataset_cifar10 ........................................ 47
dataset_cifar100 ....................................... 48
dataset_fashion_mnist ................................. 48
dataset_imdb ........................................... 49
dataset_mnist .......................................... 51
dataset_reuters ......................................... 51
evaluate.keras.engine.training.Model ............ 53
evaluate_generator ..................................... 54
export_savedmodel.keras.engine.training.Model .. 55
R topics documented:

fit.keras.engine.training.Model ............................................. 56
fit_generator ................................................................. 58
fit_image_data_generator .................................................... 60
fit_text_tokenizer ............................................................ 61
flow_images_from_data ....................................................... 61
flow_images_from_dataframe ................................................. 63
flow_images_from_directory ................................................ 65
freeze_weights ............................................................... 67
generator_next .................................................................. 68
get_config ......................................................................... 69
get_file .............................................................................. 70
get_input_at ........................................................................ 71
get_layer ........................................................................... 72
get_vocabulary ................................................................. 72
get_weights ........................................................................ 73
hdf5_matrix ....................................................................... 73
imagenet_decode_predictions ................................................ 74
imagenet_preprocess_input .................................................. 74
image_dataset_from_directory .............................................. 75
image_data_generator ........................................................ 76
image_load .......................................................................... 78
image_to_array .................................................................... 79
implementation .................................................................... 80
initializer_constant ............................................................. 81
initializer_glorot_normal ...................................................... 81
initializer_glorot_uniform .................................................... 82
initializer_he_normal .......................................................... 82
initializer_he_uniform ......................................................... 83
initializer_identity ............................................................. 84
initializer_lecun_normal ....................................................... 84
initializer_lecun_uniform ..................................................... 85
initializer_ones ................................................................. 85
initializer_orthogonal ......................................................... 86
initializer_random_normal .................................................. 87
initializer_random_uniform ................................................ 87
initializer_truncated_normal .............................................. 88
initializer_variance_scaling ................................................. 88
initializer_zeros ............................................................... 89
install_keras ..................................................................... 90
is_keras_available ............................................................. 91
keras ................................................................................. 92
KerasCallback ..................................................................... 93
KerasConstraint .................................................................. 94
KerasLayer ......................................................................... 95
KerasWrapper ..................................................................... 96
keras_array ........................................................................ 96
keras_model ....................................................................... 97
keras_model_custom ............................................................ 98
topics documented:

keras_model_sequential .................................................. 99
k_abs ................................................................. 100
k_all ................................................................. 101
k_any ................................................................. 102
k_arange ............................................................. 102
k_argmax ............................................................ 103
k_argmin ............................................................ 104
k_backend ............................................................ 104
k_batch_dot .......................................................... 105
k_batch_flatten ....................................................... 106
k_batch_get_value ..................................................... 106
k_batch_normalization .................................................. 107
k_batch_set_value ..................................................... 108
k_bias_add ........................................................... 108
k_binary_crossentropy ................................................. 109
k_cast ............................................................... 110
k_cast_to_floatx ........................................................ 110
k_categorical_crossentropy ........................................... 111
k_clear_session ........................................................ 112
k_clip ............................................................... 112
k_concatenate .......................................................... 113
k_constant .............................................................. 113
k_conv1d ............................................................. 114
k_conv2d ............................................................. 115
k_conv2d_transpose .................................................... 116
k_conv3d ............................................................. 117
k_conv3d_transpose .................................................... 118
k_cos ............................................................... 119
k_count_params ......................................................... 119
k_ctc_batch_cost ....................................................... 120
k_ctc_decode .......................................................... 121
k_ctc_label_dense_to_sparse ........................................... 122
k_cumprod ............................................................ 122
k_cumsum ............................................................. 123
k_depthwise_conv2d ..................................................... 124
k_dot ............................................................... 125
k_dropout ............................................................. 125
k_dtype ............................................................... 126
k_elu ............................................................... 127
k_epsilon .............................................................. 127
k_equal .............................................................. 128
k_eval ............................................................... 128
k_exp ............................................................... 129
k_expand_dims .......................................................... 130
k_eye ............................................................... 130
k_flatten .............................................................. 131
k_floatx .............................................................. 132
k_foldl .............................................................. 132
R topics documented:

k_foldr ................................................................. 133
k_function ............................................................. 134
k_gather ................................................................. 134
k_get_session ......................................................... 135
k_get_uid ............................................................... 136
k_get_value ............................................................ 136
k_get_variable_shape .............................................. 137
k_gradients ............................................................. 137
k_greater ............................................................... 138
k_greater_equal ....................................................... 139
k_hard_sigmoid ......................................................... 139
k_identity .............................................................. 140
k_image_data_format ................................................ 140
k_int_shape ............................................................ 141
k_in_test_phase ....................................................... 142
k_in_top_k ............................................................. 142
k_in_train_phase ..................................................... 143
k_is_keras_tensor .................................................... 144
k_is_placeholder ...................................................... 144
k_is_sparse ............................................................ 145
k_is_tensor ............................................................ 145
k_l2_normalize ......................................................... 146
k_learning_phase ..................................................... 147
k_less ................................................................. 147
k_less_equal ........................................................... 148
k_local_conv1d ....................................................... 148
k_local_conv2d ....................................................... 149
k_log ................................................................. 150
k_logsumexp ........................................................... 151
k_manual_variable_initialization ................................ 151
k_map_fn ............................................................... 152
k_max ................................................................. 153
k_maximum .............................................................. 153
k_mean ................................................................. 154
k_min ................................................................. 155
k_minimum ............................................................. 155
k_moving_average_update ......................................... 156
k_ndim ................................................................. 157
k_normalize_batch_in_training .................................... 157
k_not_equal ............................................................ 158
k_ones ................................................................. 159
k_ones_like ............................................................ 159
k_one_hot .............................................................. 160
k_permute_dimensions .............................................. 161
k_placeholder ........................................................ 161
k_pool2d ............................................................... 162
k_pool3d ............................................................... 163
k_pow ................................................................. 164
topics documented:

- k_print_tensor .......................................................... 164
- k_prod ................................................................. 165
- k_random_binomial .................................................... 166
- k_random_normal ....................................................... 166
- k_random_normal_variable .......................................... 167
- k_random_uniform ..................................................... 168
- k_random_uniform_variable ........................................ 169
- k_relu ................................................................. 170
- k_repeat ............................................................... 170
- k_repeat_elements ................................................... 171
- k_reset_uids .......................................................... 172
- k_reshape ............................................................. 172
- k_resize_images ...................................................... 173
- k_resize_volumes .................................................... 173
- k_reverse ............................................................. 174
- k_rnn ................................................................. 175
- k_round ............................................................... 176
- k_separable_conv2d .................................................. 176
- k_set_learning_phase ............................................... 177
- k_set_value .......................................................... 178
- k_shape .............................................................. 178
- k_sigmoid ............................................................. 179
- k_sign ............................................................... 180
- k_sin ................................................................. 180
- k_softmax ............................................................ 181
- k_softplus ........................................................... 182
- k_softsign ............................................................ 182
- k_sparse_categorical_crossentropy .............................. 183
- k_spatial_2d_padding ............................................... 184
- k_spatial_3d_padding ............................................... 184
- k_sqrt ............................................................... 185
- k_square ............................................................. 186
- k_squeeze ............................................................ 186
- k_stack ............................................................... 187
- k_std ................................................................. 188
- k_stop_gradient ...................................................... 188
- k_sum ............................................................... 189
- k_switch ............................................................. 190
- k_tanh ............................................................... 190
- k_temporal_padding .................................................. 191
- k_tile ............................................................... 192
- k_to_dense ........................................................... 192
- k_transpose .......................................................... 193
- k_truncated_normal .................................................. 193
- k_update ............................................................. 194
- k_update_add ........................................................ 195
- k_update_sub ........................................................ 195
- k_var ............................................................... 196
The topics documented:

- k_variable ............................................. 197
- k_zeros .................................................. 197
- k_zeros_like ............................................ 198
- Layer ..................................................... 199
- layer_activation ...................................... 200
- layer_activation_elu .................................. 201
- layer_activation_leaky_relu .......................... 203
- layer_activation_parametric_relu ..................... 204
- layer_activation_relu ................................ 205
- layer_activation_selu .................................. 206
- layer_activation_softmax .............................. 207
- layer_activity_regularization ........................ 209
- layer_add ................................................ 211
- layer_alpha_dropout ................................... 212
- layer_attention ........................................ 213
- layer_average .......................................... 214
- layer_average_pooling_1d ............................. 215
- layer_average_pooling_2d ............................. 216
- layer_average_pooling_3d ............................. 218
- layer_batch_normalization ............................. 219
- layer_concatenate ...................................... 222
- layer_conv_1d .......................................... 223
- layer_conv_1d_transpose ............................... 225
- layer_conv_2d .......................................... 228
- layer_conv_2d_transpose ............................... 230
- layer_conv_3d .......................................... 233
- layer_conv_3d_transpose ............................... 236
- layer_conv_lstm_2d ..................................... 238
- layer_cropping_1d ...................................... 241
- layer_cropping_2d ...................................... 242
- layer_cropping_3d ...................................... 244
- layer_cudnn_gru ........................................ 245
- layer_cudnn_lstm ....................................... 247
- layer_dense ............................................. 249
- layer_dense_features ................................... 251
- layer_depthwise_conv_2d ............................... 252
- layer_dot ................................................ 254
- layer_dropout .......................................... 255
- layer_embedding ........................................ 256
- layer_flatten .......................................... 258
- layer_gaussian_dropout ................................ 259
- layer_gaussian_noise .................................. 260
- layer_global_average_pooling_1d ..................... 261
- layer_global_average_pooling_2d ..................... 262
- layer_global_average_pooling_3d ..................... 263
- layer_global_max_pooling_1d .......................... 264
- layer_global_max_pooling_2d .......................... 265
R topics documented:

layer_global_max_pooling_3d .............................................. 267
layer_gru ................................................................. 268
layer_input ............................................................... 272
layer_lambda ............................................................... 273
layer_layer_normalization .................................................. 274
layer_locally_connected_1d .................................................. 275
layer_locally_connected_2d .................................................. 278
layer_lstm .................................................................. 280
layer_masking .............................................................. 284
layer_maximum .............................................................. 285
layer_max_pooling_1d ........................................................ 286
layer_max_pooling_2d ........................................................ 287
layer_max_pooling_3d ........................................................ 288
layer_minimum .............................................................. 290
layer_multiply .............................................................. 291
layer_multi_head_attention ................................................. 292
layer_permute ............................................................... 294
layer_repeat_vector ......................................................... 295
layer_reshape .............................................................. 296
layer_separable_conv_1d ..................................................... 297
layer_separable_conv_2d ..................................................... 300
layer_simple_rnn ........................................................... 303
layer_spatial_dropout_1d .................................................... 306
layer_spatial_dropout_2d .................................................... 307
layer_spatial_dropout_3d .................................................... 308
layer_subtract .............................................................. 310
layer_text_vectorization .................................................... 311
layer_upsampling_1d ......................................................... 312
layer_upsampling_2d ......................................................... 313
layer_upsampling_3d ......................................................... 315
layer_zero_padding_1d ....................................................... 316
layer_zero_padding_2d ....................................................... 317
layer_zero_padding_3d ....................................................... 319
loss-functions ............................................................... 320
loss_cosine_proximity ....................................................... 324
make_sampling_table ........................................................ 325
Metric ......................................................................... 326
metric_accuracy ............................................................ 327
metric_auc ................................................................. 328
metric_binary_accuracy ..................................................... 330
metric_binary_crossentropy ............................................... 332
metric_categorical_accuracy .............................................. 333
metric_categorical_crossentropy ......................................... 334
metric_categorical_hinge .................................................. 336
metric_cosine_proximity ................................................... 337
metric_cosine_similarity .................................................. 337
metric_false_negatives ..................................................... 338
metric_false_positives ..................................................... 339
<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>metric_hinge</td>
<td>340</td>
</tr>
<tr>
<td>metric_kullback_leibler_divergence</td>
<td>341</td>
</tr>
<tr>
<td>metric_logcosh_error</td>
<td>343</td>
</tr>
<tr>
<td>metric_mean</td>
<td>344</td>
</tr>
<tr>
<td>metric_mean_absolute_error</td>
<td>345</td>
</tr>
<tr>
<td>metric_mean_absolute_percentage_error</td>
<td>346</td>
</tr>
<tr>
<td>metric_mean_iou</td>
<td>347</td>
</tr>
<tr>
<td>metric_mean_relative_error</td>
<td>349</td>
</tr>
<tr>
<td>metric_mean_squared_error</td>
<td>350</td>
</tr>
<tr>
<td>metric_mean_squared_logarithmic_error</td>
<td>351</td>
</tr>
<tr>
<td>metric_mean_tensor</td>
<td>352</td>
</tr>
<tr>
<td>metric_mean_wrapper</td>
<td>353</td>
</tr>
<tr>
<td>metric_poisson</td>
<td>354</td>
</tr>
<tr>
<td>metric_precision</td>
<td>355</td>
</tr>
<tr>
<td>metric_precision_at_recall</td>
<td>357</td>
</tr>
<tr>
<td>metric_recall</td>
<td>358</td>
</tr>
<tr>
<td>metric_recall_at_precision</td>
<td>359</td>
</tr>
<tr>
<td>metric_root_mean_squared_error</td>
<td>361</td>
</tr>
<tr>
<td>metric_specificity_at_sensitivity</td>
<td>362</td>
</tr>
<tr>
<td>metric_sparse_categorical_accuracy</td>
<td>363</td>
</tr>
<tr>
<td>metric_sparse_categorical_crossentropy</td>
<td>364</td>
</tr>
<tr>
<td>metric_sparse_top_k_categorical_accuracy</td>
<td>365</td>
</tr>
<tr>
<td>metric_specificity_at_sensitivity</td>
<td>367</td>
</tr>
<tr>
<td>metric_squared_hinge</td>
<td>368</td>
</tr>
<tr>
<td>metric_sum</td>
<td>369</td>
</tr>
<tr>
<td>metric_top_k_categorical_accuracy</td>
<td>370</td>
</tr>
<tr>
<td>metric_true_negatives</td>
<td>371</td>
</tr>
<tr>
<td>metric_true_positives</td>
<td>372</td>
</tr>
<tr>
<td>model_from_saved_model</td>
<td>373</td>
</tr>
<tr>
<td>model_to_json</td>
<td>374</td>
</tr>
<tr>
<td>model_to_saved_model</td>
<td>375</td>
</tr>
<tr>
<td>model_to_yaml</td>
<td>376</td>
</tr>
<tr>
<td>multi_gpu_model</td>
<td>376</td>
</tr>
<tr>
<td>normalize</td>
<td>379</td>
</tr>
<tr>
<td>optimizer_adadelta</td>
<td>379</td>
</tr>
<tr>
<td>optimizer_adagrad</td>
<td>380</td>
</tr>
<tr>
<td>optimizer_adam</td>
<td>381</td>
</tr>
<tr>
<td>optimizer_adamax</td>
<td>382</td>
</tr>
<tr>
<td>optimizer_nadam</td>
<td>383</td>
</tr>
<tr>
<td>optimizer_rmsprop</td>
<td>384</td>
</tr>
<tr>
<td>optimizer_sgd</td>
<td>385</td>
</tr>
<tr>
<td>pad_sequences</td>
<td>386</td>
</tr>
<tr>
<td>plot.keras_training_history</td>
<td>387</td>
</tr>
<tr>
<td>pop_layer</td>
<td>388</td>
</tr>
<tr>
<td>predict.keras.engine.training.Model</td>
<td>388</td>
</tr>
<tr>
<td>predict_generator</td>
<td>389</td>
</tr>
<tr>
<td>predict_on_batch</td>
<td>390</td>
</tr>
<tr>
<td>predict_proba</td>
<td>391</td>
</tr>
</tbody>
</table>
Description

Keras is a high-level neural networks API, developed with a focus on enabling fast experimentation. Keras has the following key features:

Details

- Allows the same code to run on CPU or on GPU, seamlessly.
- User-friendly API which makes it easy to quickly prototype deep learning models.
- Built-in support for convolutional networks (for computer vision), recurrent networks (for sequence processing), and any combination of both.
- Supports arbitrary network architectures: multi-input or multi-output models, layer sharing, model sharing, etc. This means that Keras is appropriate for building essentially any deep learning model, from a memory network to a neural Turing machine.
• Is capable of running on top of multiple back-ends including TensorFlow, CNTK, or Theano. See the package website at https://keras.rstudio.com for complete documentation.

Author(s)

Maintainer: Tomasz Kalinowski <tomasz.kalinowski@rstudio.com> [contributor, copyright holder]

Authors:

• JJ Allaire [copyright holder]
• François Chollet [copyright holder]

Other contributors:

• Daniel Falbel <daniel@rstudio.com> [contributor, copyright holder]
• RStudio [contributor, copyright holder, funder]
• Google [contributor, copyright holder, funder]
• Yuan Tang <terrytangyuan@gmail.com> (ORCID) [contributor, copyright holder]
• Wouter Van Der Bijl [contributor, copyright holder]
• Martin Studer [contributor, copyright holder]
• Sigrid Keydana [contributor]

See Also

Useful links:

• https://keras.rstudio.com
• Report bugs at https://github.com/rstudio/keras/issues

---

**Description**

relu(...): Applies the rectified linear unit activation function.
elu(...): Exponential Linear Unit.
selu(...): Scaled Exponential Linear Unit (SELU).
hard_sigmoid(...): Hard sigmoid activation function.
linear(...): Linear activation function (pass-through).
sigmoid(...): Sigmoid activation function, sigmoid(x) = 1 / (1 + exp(-x)).
softmax(...): Softmax converts a vector of values to a probability distribution.
softplus(...): Softplus activation function, softplus(x) = log(exp(x) + 1).
softsign(...): Softsign activation function, softsign(x) = x / (abs(x) + 1).
tanh(...): Hyperbolic tangent activation function.

exponential(...): Exponential activation function.

gelu(...): Applies the Gaussian error linear unit (GELU) activation function.

swish(...): Swish activation function, swish(x) = x * sigmoid(x).

Usage

activation_relu(x, alpha = 0, max_value = NULL, threshold = 0)

activation_elu(x, alpha = 1)

activation_selu(x)

activation_hard_sigmoid(x)

activation_linear(x)

activation_sigmoid(x)

activation_softmax(x, axis = -1)

activation_softplus(x)

activation_softsign(x)

activation_tanh(x)

activation_exponential(x)

activation_gelu(x, approximate = FALSE)

activation_swish(x)

Arguments

x Tensor

alpha Alpha value

max_value Max value

threshold Threshold value for thresholded activation.

axis Integer, axis along which the softmax normalization is applied

approximate A bool, whether to enable approximation.

Details

Activations functions can either be used through layer_activation(), or through the activation argument supported by all forward layers.
adapt

- activation_selu() to be used together with the initialization "lecun_normal".
- activation_selu() to be used together with the dropout variant "AlphaDropout".

Value

Tensor with the same shape and dtype as x.

References

- activation_swish(): Searching for Activation Functions
- activation_gelu(): Gaussian Error Linear Units (GELUs)
- activation_selu(): Self-Normalizing Neural Networks
- activation_elu(): Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs)

See Also

https://www.tensorflow.org/api_docs/python/tf/keras/activations

---

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, reset_state = 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.

reset_state  
Optional argument specifying whether to clear the state of the layer at the start of the call to adapt, or whether to start from the existing state. Subclasses may choose to throw if reset_state is set to FALSE. NULL mean layer’s default.
Instantiates the DenseNet architecture.

**Usage**

```r
application_densenet(
    blocks,
    include_top = TRUE,
    weights = "imagenet",
    input_tensor = NULL,
    input_shape = NULL,
    pooling = NULL,
    classes = 1000
)
```

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

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

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

densenet_preprocess_input(x, data_format = NULL)
application_inception_resnet_v2

Arguments

blocks: numbers of building blocks for the four dense layers.
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 layer_input()) to use as image input for the model.
input_shape: optional shape list, 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.
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.
x: a 3D or 4D array consists of RGB values within [0, 255].
data_format: data format of the image tensor.

Details

Optionally loads weights pre-trained on ImageNet. Note that when using TensorFlow, for best performance you should set image_data_format='channels_last' in your Keras config at ~/.keras/keras.json.

The model and the weights are compatible with TensorFlow, Theano, and CNTK. The data format convention used by the model is the one specified in your Keras config file.

application_inception_resnet_v2

Inception-ResNet v2 model, with weights trained on ImageNet

Description

Inception-ResNet v2 model, with weights trained on ImageNet

Usage

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

inception_resnet_v2_preprocess_input(x)

**Arguments**

- **include_top**: whether to include the fully-connected layer at the top of the network.
- **weights**: NULL (random initialization), imagenet (ImageNet weights), or the path to the weights file to be loaded.
- **input_tensor**: optional Keras tensor to use as image input for the model.
- **input_shape**: optional shape list, 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 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 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.
- **x**: Input tensor for preprocessing

**Details**

Do note that the input image format for this model is different than for the VGG16 and ResNet models (299x299 instead of 224x224).

The `inception_resnet_v2_preprocess_input()` function should be used for image preprocessing.

**Value**

A Keras model instance.

**Reference**

application_inception_v3

Inception V3 model, with weights pre-trained on ImageNet.

Description

Inception V3 model, with weights pre-trained on ImageNet.

Usage

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

inception_v3_preprocess_input(x)

Arguments

include_top whether to include the fully-connected layer at the top of the network.
weights NULL (random initialization), imagenet (ImageNet weights), or the path to the weights file to be loaded.
input_tensor optional Keras tensor to use as image input for the model.
input_shape optional shape list, 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 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 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.
x Input tensor for preprocessing
Details

Do note that the input image format for this model is different than for the VGG16 and ResNet models (299x299 instead of 224x224).

The `inception_v3_preprocess_input()` function should be used for image preprocessing.

Value

A Keras model instance.

Reference

- Rethinking the Inception Architecture for Computer Vision

Description

MobileNet model architecture.

Usage

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

```r
mobilenet_preprocess_input(x)
```

```r
mobilenet_decode_predictions(preds, top = 5)
```

```r
mobilenet_load_model_hdf5(filepath)
```

Arguments

- `input_shape` optional shape list, 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.
alpha controls the width of the network.

- 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.

depth_multiplier depth multiplier for depthwise convolution (also called the resolution multiplier)
dropout dropout rate
include_top whether to include the fully-connected layer at the top of the network.
weights NULL (random initialization), imagenet (ImageNet weights), or the path to the weights file to be loaded.
input_tensor optional Keras tensor (i.e. output of layer_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.
x input tensor, 4D
preds Tensor encoding a batch of predictions.
top integer, how many top-guesses to return.
filepath File path

Details

The mobilenet_preprocess_input() function should be used for image preprocessing. To load a saved instance of a MobileNet model use the mobilenet_load_model_hdf5() function. To prepare image input for MobileNet use mobilenet_preprocess_input(). To decode predictions use mobilenet_decode_predictions().

Value

application_mobilenet() and mobilenet_load_model_hdf5() return a Keras model instance. mobilenet_preprocess_input() returns image input suitable for feeding into a mobilenet model. mobilenet_decode_predictions() returns a list of data frames with variables class_name, class_description, and score (one data frame per sample in batch input).

Reference

- MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications.
Description

MobileNetV2 model architecture

Usage

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

mobilenet_v2_preprocess_input(x)

mobilenet_v2_decode_predictions(preds, top = 5)

mobilenet_v2_load_model_hdf5(filepath)

Arguments

- **input_shape**: optional shape list, 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.

- **alpha**: controls the width of the network.
  - 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.

- **include_top**: whether to include the fully-connected layer at the top of the network.

- **weights**: NULL (random initialization), imagenet (ImageNet weights), or the path to the weights file to be loaded.

- **input_tensor**: optional Keras tensor (i.e. output of layer_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.
- **x**: input tensor, 4D
- **preds**: Tensor encoding a batch of predictions.
- **top**: integer, how many top-guesses to return.
- **filepath**: File path

**Value**

`application_mobilenet_v2()` and `mobilenet_v2_load_model_hdf5()` return a Keras model instance. `mobilenet_v2_preprocess_input()` returns image input suitable for feeding into a mobilenet v2 model. `mobilenet_v2_decode_predictions()` returns a list of data frames with variables `class_name`, `class_description`, and `score` (one data frame per sample in batch input).

**Reference**

- MobileNetV2: Inverted Residuals and Linear Bottlenecks

**See Also**

`application_mobilenet`

---

**application_nasnet**

Instantiates a NASNet model.

**Description**

Note that only TensorFlow is supported for now, therefore it only works with the data format `image_data_format='channels_last'` in your Keras config at `~/.keras/keras.json`.

**Usage**

```r
application_nasnet(
  input_shape = NULL,
  penultimate_filters = 4032L,
  num_blocks = 6L,
  stem_block_filters = 96L,
  skip_reduction = TRUE,
  filter_multiplier = 2L,
  include_top = TRUE,
  weights = NULL,
  input_tensor = NULL,
  pooling = NULL,
)```

classes = 1000,
default_size = NULL
)

application_nasnetlarge(
    input_shape = NULL,
    include_top = TRUE,
    weights = NULL,
    input_tensor = NULL,
    pooling = NULL,
    classes = 1000
)

application_nasnetmobile(
    input_shape = NULL,
    include_top = TRUE,
    weights = NULL,
    input_tensor = NULL,
    pooling = NULL,
    classes = 1000
)

nasnet_preprocess_input(x)

Arguments

input_shape  Optional shape list, the input shape is by default (331, 331, 3) for NASNetLarge and (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.

penultimate_filters  Number of filters in the penultimate layer. NASNet models use the notation NASNet (N @ P), where: N is the number of blocks, P is the number of penultimate filters.

num_blocks  Number of repeated blocks of the NASNet model. NASNet models use the notation NASNet (N @ P), where: N is the number of blocks, P is the number of penultimate filters.

stem_block_filters  Number of filters in the initial stem block.

skip_reduction  Whether to skip the reduction step at the tail end of the network. Set to FALSE for CIFAR models.

filter_multiplier  Controls the width of the network.
  • If filter_multiplier < 1.0, proportionally decreases the number of filters in each layer.
  • If filter_multiplier > 1.0, proportionally increases the number of filters in each layer. If filter_multiplier = 1, default number of filters from the paper are used at each layer.
include_top  Whether to include the fully-connected layer at the top of the network.
weights NULL (random initialization) or imagenet (ImageNet weights)
input_tensor Optional Keras tensor (i.e. output of layer_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.
default_size Specifies the default image size of the model
x a 4D array consists of RGB values within [0, 255].

Description
ResNet50 model for Keras.

Usage
application_resnet50(
  include_top = TRUE,
  weights = "imagenet",
  input_tensor = NULL,
  input_shape = NULL,
  pooling = NULL,
  classes = 1000
)

Arguments
include_top whether to include the fully-connected layer at the top of the network.
weights NULL (random initialization), imagenet (ImageNet weights), or the path to the weights file to be loaded.
input_tensor optional Keras tensor to use as image input for the model.
input_shape optional shape list, only to be specified if include_top is FALSE (otherwise the input shape has to be (224, 224, 3). 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 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.

Details
Optionally loads weights pre-trained on ImageNet.
The imagenet_preprocess_input() function should be used for image preprocessing.

Value
A Keras model instance.

Reference
- Deep Residual Learning for Image Recognition

Examples
```r
## Not run:
library(keras)

# instantiate the model
model <- application_resnet50(weights = 'imagenet')

# load the image
img_path <- "elephant.jpg"
img <- image_load(img_path, target_size = c(224,224))
x <- image_to_array(img)

# ensure we have a 4d tensor with single element in the batch dimension,
# the preprocess the input for prediction using resnet50
x <- array_reshape(x, c(1, dim(x)))
x <- imagenet_preprocess_input(x)

# make predictions then decode and print them
preds <- model %>% predict(x)
imagenet_decode_predictions(preds, top = 3)[[1]]
```
## End(Not run)
VGG16 and VGG19 models for Keras.

Usage

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

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

Arguments

- `include_top` : whether to include the 3 fully-connected layers at the top of the network.
- `weights` : NULL (random initialization), imagenet (ImageNet weights), or the path to the weights file to be loaded.
- `input_tensor` : optional Keras tensor to use as image input for the model.
- `input_shape` : optional shape list, only to be specified if `include_top` is FALSE (otherwise the input shape has to be (224, 224, 3) 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 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.
Details

Optionally loads weights pre-trained on ImageNet.
The imagenet_preprocess_input() function should be used for image preprocessing.

Value

Keras model instance.

Reference

- Very Deep Convolutional Networks for Large-Scale Image Recognition

Examples

```r
## Not run:
library(keras)

model <- application_vgg16(weights = 'imagenet', include_top = FALSE)

img_path <- "elephant.jpg"
img <- image_load(img_path, target_size = c(224,224))
x <- image_to_array(img)
x <- array_reshape(x, c(1, dim(x)))
x <- imagenet_preprocess_input(x)
features <- model %>% predict(x)

## End(Not run)
```

application_xception  Xception V1 model for Keras.

Description

Xception V1 model for Keras.

Usage

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

xception_preprocess_input(x)
```
application_xception

Arguments

- **include_top**: whether to include the fully-connected layer at the top of the network.
- **weights**: NULL (random initialization), imagenet (ImageNet weights), or the path to the weights file to be loaded.
- **input_tensor**: optional Keras tensor to use as image input for the model.
- **input_shape**: optional shape list, 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 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 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.
- **x**: Input tensor for preprocessing

Details

On ImageNet, this model gets to a top-1 validation accuracy of 0.790 and a top-5 validation accuracy of 0.945.

Do note that the input image format for this model is different than for the VGG16 and ResNet models (299x299 instead of 224x224).

The xception_preprocess_input() function should be used for image preprocessing.

This application is only available when using the TensorFlow back-end.

Value

A Keras model instance.

Reference

- Xception: Deep Learning with Depthwise Separable Convolutions
**backend**  
*Keras backend tensor engine*

**Description**
Obtain a reference to the keras.backend Python module used to implement tensor operations.

**Usage**
```
backend(convert = TRUE)
```

**Arguments**
- `convert`  
  Boolean; should Python objects be automatically converted to their R equivalent? If set to `FALSE`, you can still manually convert Python objects to R via the `py_to_r()` function.

**Value**
Reference to Keras backend python module.

**Note**
See the documentation here [https://keras.io/backend/](https://keras.io/backend/) for additional details on the available functions.

---

**bidirectional**  
*Bidirectional wrapper for RNNs.*

**Description**
Bidirectional wrapper for RNNs.

**Usage**
```
bidirectional(
    object,
    layer,
    merge_mode = "concat",
    input_shape = NULL,
    batch_input_shape = NULL,
    batch_size = NULL,
    dtype = NULL,
    name = NULL,
    trainable = NULL,
    weights = NULL
)
```
callback_csv_logger

Arguments

object  Model or layer object
layer   Recurrent instance.
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.
input_shape Dimensionality of the input (integer) not including the samples axis. This argument is required when using this layer as the first layer in a model.
batch_input_shape Shapes, including the batch size. For instance, batch_input_shape=c(10,32) indicates that the expected input will be batches of 10 32-dimensional vectors. batch_input_shape=list(NULL,32) indicates batches of an arbitrary number of 32-dimensional vectors.
batch_size Fixed batch size for layer
dtype The data type expected by the input, as a string (float32, float64, int32...)
name An 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.
trainable Whether the layer weights will be updated during training.
weights Initial weights for layer.

See Also

Other layer wrappers: time_distributed()

callback_csv_logger  Callback that streams epoch results to a csv file

Description

Supports all values that can be represented as a string

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  TRUE: append if file exists (useful for continuing training). FALSE: overwrite existing file,
callback_early_stopping

Stop training when a monitored quantity has stopped improving.

Description

Stop training when a monitored quantity has stopped improving.

Usage

callback_early_stopping(
    monitor = "val_loss",
    min_delta = 0,
    patience = 0,
    verbose = 0,
    mode = c("auto", "min", "max"),
    baseline = NULL,
    restore_best_weights = FALSE
)

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>monitor</td>
<td>quantity to be monitored.</td>
</tr>
<tr>
<td>min_delta</td>
<td>minimum change in the monitored quantity to qualify as an improvement, i.e.</td>
</tr>
<tr>
<td></td>
<td>an absolute change of less than min_delta, will count as no improvement.</td>
</tr>
<tr>
<td>patience</td>
<td>number of epochs with no improvement after which training will be stopped.</td>
</tr>
<tr>
<td>verbose</td>
<td>verbosity mode, 0 or 1.</td>
</tr>
<tr>
<td>mode</td>
<td>one of &quot;auto&quot;, &quot;min&quot;, &quot;max&quot;. In min mode, training will stop when the quantity</td>
</tr>
<tr>
<td></td>
<td>monitored has stopped decreasing; in max mode it will stop when the quantity</td>
</tr>
<tr>
<td></td>
<td>monitored has stopped increasing; in auto mode, the direction is automatically</td>
</tr>
<tr>
<td></td>
<td>inferred from the name of the monitored quantity.</td>
</tr>
<tr>
<td>baseline</td>
<td>Baseline value for the monitored quantity to reach. Training will stop if the</td>
</tr>
<tr>
<td></td>
<td>model doesn’t show improvement over the baseline.</td>
</tr>
<tr>
<td>restore_best_weights</td>
<td>Whether to restore model weights from the epoch with the best value of the</td>
</tr>
<tr>
<td></td>
<td>monitored quantity. If FALSE, the model weights obtained at the last step</td>
</tr>
<tr>
<td></td>
<td>of training are used.</td>
</tr>
</tbody>
</table>
See Also

Other callbacks: `callback_csv_logger()`, `callback_lambda()`, `callback_learning_rate_scheduler()`, `callback_model_checkpoint()`, `callback_progbar_logger()`, `callback_reduce_lr_on_plateau()`, `callback_remote_monitor()`, `callback_tensorboard()`, `callback_terminate_on_naan()`

---

**callback_lambda**  
Create a custom callback

**Description**

This callback is constructed with anonymous functions that will be called at the appropriate time. Note that the callbacks expects positional arguments, as:

**Usage**

```python
callback_lambda(  
on_epoch_begin = NULL,  
on_epoch_end = NULL,  
on_batch_begin = NULL,  
on_batch_end = NULL,  
on_train_batch_begin = NULL,  
on_train_batch_end = NULL,  
on_train_begin = NULL,  
on_train_end = NULL,  
on_predict_batch_begin = NULL,  
on_predict_batch_end = NULL,  
on_predict_begin = NULL,  
on_predict_end = NULL,  
on_test_batch_begin = NULL,  
on_test_batch_end = NULL,  
on_test_begin = NULL,  
on_test_end = NULL
)
```

**Arguments**

- `on_epoch_begin` called at the beginning of every epoch.
- `on_epoch_end` called at the end of every epoch.
- `on_batch_begin` called at the beginning of every training batch.
- `on_batch_end` called at the end of every training batch.
- `on_train_batch_begin` called at the beginning of every batch.
- `on_train_batch_end` called at the end of every batch.
- `on_train_begin` called at the beginning of model training.
on_train_end  called at the end of model training.
on_predict_batch_begin
called at the beginning of a batch in predict methods.
on_predict_batch_end
called at the end of a batch in predict methods.
on_predict_begin
called at the beginning of prediction.
on_predict_end  called at the end of prediction.
on_test_batch_begin
called at the beginning of a batch in evaluate methods. Also called at the begin-
ing of a validation batch in the fit methods, if validation data is provided.
on_test_batch_end
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.
on_test_begin  called at the beginning of evaluation or validation.
on_test_end  called at the end of evaluation or validation.

Details

• on_epoch_begin and on_epoch_end expect two positional arguments: epoch, logs
• on_batch_* on_train_batch_* on_predict_batch_* and on_test_batch_* expect two posi-
tional arguments: batch, logs
• on_train_* on_test_* and on_predict_* expect one positional argument: logs

See Also

Other callbacks: callback_csv_logger(), callback_early_stopping(), callback_learning_rate_scheduler(),
callback_model_checkpoint(), callback_progbar_logger(), callback_reduce_lr_on_plateau(),
callback_remote_monitor(), callback_tensorboard(), callback_terminate_on_naan()
callback_model_checkpoint

See Also

Other callbacks: callback_csv_logger(), callback_early_stopping(), callback_lambda(), callback_model_checkpoint(), callback_progbar_logger(), callback_reduce_lr_on_plateau(), callback_remote_monitor(), callback_tensorboard(), callback_terminate_on_naan()

---

callback_model_checkpoint

Save the model after every epoch.

Description

filepath can contain named formatting options, which will be filled the value of epoch and keys in logs (passed in on_epoch_end). For example: if filepath is weights.{epoch:02d}-{val_loss:.2f}.hdf5, then the model checkpoints will be saved with the epoch number and the validation loss in the filename.

Usage

callback_model_checkpoint(
    filepath,
    monitor = "val_loss",
    verbose = 0,
    save_best_only = FALSE,
    save_weights_only = FALSE,
    mode = c("auto", "min", "max"),
    period = NULL,
    save_freq = "epoch"
)

Arguments

filepath: string, path to save the model file.
monitor: quantity to monitor.
verbose: verbosity mode, 0 or 1.
save_best_only: if save_best_only=TRUE, the latest best model according to the quantity monitored will not be overwritten.
save_weights_only: if TRUE, then only the model's weights will be saved (save_model_weights_hdf5(filepath)), else the full model is saved (save_model_hdf5(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 direction is automatically inferred from the name of the monitored quantity.
period: Interval (number of epochs) between checkpoints.
Callback that prints metrics to stdout.

Description

Callback that prints metrics to stdout.

Usage

callback_progbar_logger(count_mode = "samples", stateful_metrics = NULL)

Arguments

count_mode
One of "steps" or "samples". Whether the progress bar should count samples
seens or steps (batches) seen.

stateful_metrics
List of metric names that should not be averaged over an epoch. Metrics in
this list will be logged as-is in on_epoch_end. All others will be averaged in
on_epoch_end.

See Also

Other callbacks: callback_csv_logger(), callback_early_stopping(), callback_lambda(),
callback_learning_rate_scheduler(), callback_progbar_logger(), callback_reduce_lr_on_plateau(),
callback_remote_monitor(), callback_tensorboard(), callback_terminate_on_naan()
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

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

Arguments

- **monitor**: quantity to be monitored.
- **factor**: factor by which the learning rate will be reduced. `new_lr = lr \times factor`
- **patience**: number of epochs with no improvement after which learning rate will be reduced.
- **verbose**: int. 0: quiet, 1: update messages.
- **mode**: one of "auto", "min", "max". In min mode, lr 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**: threshold for measuring the new optimum, to only focus on significant changes.
- **cooldown**: number of epochs to wait before resuming normal operation after lr has been reduced.
- **min_lr**: lower bound on the learning rate.

See Also

Other callbacks: `callback_csv_logger()`, `callback_early_stopping()`, `callback_lambda()`, `callback_learning_rate_scheduler()`, `callback_model_checkpoint()`, `callback_progbars_logger()`, `callback_remote_monitor()`, `callback_tensorboard()`, `callback_terminate_on_naan()`
callback_remote_monitor

Callback used to stream events to a server.

Description

Callback used to stream events to a server.

Usage

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

Arguments

- **root**: root url of the target server.
- **path**: path relative to root to which the events will be sent.
- **field**: JSON field under which the data will be stored.
- **headers**: Optional named list of custom HTTP headers. Defaults to: list(Accept = "application/json", Content-Type = "application/json")
- **send_as_json**: Whether the request should be sent as application/json.

Details

Events are sent to root + '/publish/epoch/end/' by default. Calls are HTTP POST, with a data argument which is a JSON-encoded dictionary of event data. If send_as_json is set to True, the content type of the request will be application/json. Otherwise the serialized JSON will be send within a form.

See Also

Other callbacks: callback_csv_logger(), callback_early_stopping(), callback_lambda(), callback_learning_rate_scheduler(), callback_model_checkpoint(), callback_progbar_logger(), callback_reduce_lr_on_plateau(), callback_tensorboard(), callback_terminate_on_naan()
**callback_tensorboard**  
*TensorBoard basic visualizations*

**Description**

This callback writes a log for TensorBoard, which allows you to visualize dynamic graphs of your training and test metrics, as well as activation histograms for the different layers in your model.

**Usage**

```r
callback_tensorboard(
  log_dir = NULL,
  histogram_freq = 0,
  batch_size = NULL,
  write_graph = TRUE,
  write_grads = FALSE,
  write_images = FALSE,
  embeddings_freq = 0,
  embeddings_layer_names = NULL,
  embeddings_metadata = NULL,
  embeddings_data = NULL,
  update_freq = "epoch",
  profile_batch = 0
)
```

**Arguments**

- **log_dir**  
The path of the directory where to save the log files to be parsed by Tensorboard. The default is `NULL`, which will use the active run directory (if available) and otherwise will use "logs".

- **histogram_freq**  
  frequency (in epochs) at which to compute activation histograms for the layers of the model. If set to 0, histograms won't be computed.

- **batch_size**  
  size of batch of inputs to feed to the network for histograms computation. No longer needed, ignored since TF 1.14.

- **write_graph**  
  whether to visualize the graph in Tensorboard. The log file can become quite large when write_graph is set to `TRUE`.

- **write_grads**  
  whether to visualize gradient histograms in TensorBoard. `histogram_freq` must be greater than 0.

- **write_images**  
  whether to write model weights to visualize as image in Tensorboard.

- **embeddings_freq**  
  frequency (in epochs) at which selected embedding layers will be saved.

- **embeddings_layer_names**  
  a list of names of layers to keep eye on. If `NULL` or empty list all the embedding layers will be watched.
callback_terminate_on_naan

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_naan()

See Also
Other callbacks: callback_csv_logger(), callback_early_stopping(), callback_lambda(),
callback_learning_rate_scheduler(), callback_model_checkpoint(), callback_progbar_logger(),
callback_reduce_lr_on_plateau(), callback_remote_monitor(), callback_tensorboard()
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**

```r
clone_model(model, input_tensors = 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.

---

compile.keras.engine.training.Model

Configure a Keras model for training

**Description**

Configure a Keras model for training

**Usage**

```r
## S3 method for class 'keras.engine.training.Model'
compile(
  object,
  optimizer = NULL,
  loss = NULL,
  metrics = NULL,
  loss_weights = NULL,
  weighted_metrics = NULL,
  run_eagerly = NULL,
  steps_per_execution = NULL,
  ...,
  target_tensors = NULL,
  sample_weight_mode = NULL
)
```
compile.keras.engine.training.Model

Arguments

Arguments:

object
Model object to compile.

optimizer
String (name of optimizer) or optimizer instance. For most models, this defaults to "rmsprop".

loss
String (name of objective function), objective function or a keras$losses$Loss subclass instance. An objective function is any callable with the signature `loss = fn(y_true, y_pred)`, where `y_true` = ground truth values with shape = [batch_size, d0, .. dN], except sparse loss functions such as sparse categorical crossentropy where shape = [batch_size, d0, .. dN-1]. `y_pred` = predicted values with shape = [batch_size, d0, .. dN]. It returns a weighted loss float tensor. If a custom Loss instance is used and reduction is set to NULL, return value has the shape [batch_size, d0, .. dN-1] i.e. per-sample or per-timestep loss values; otherwise, it is a scalar. If the model has multiple outputs, you can use a different loss on each output by passing a dictionary or a list of losses. The loss value that will be minimized by the model will then be the sum of all individual losses, unless loss_weights is specified.

metrics
List of metrics to be evaluated by the model during training and testing. Each of this can be a string (name of a built-in function), function or a keras$metrics$Metric class instance. See ?tf.keras$metrics. Typically you will use metrics=list('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 dictionary, such as `metrics=list(output_a = 'accuracy', output_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(list('accuracy'), list('accuracy', 'mse'))` or `metrics=list('accuracy', c('accuracy', 'mse'))`. When you pass the strings 'accuracy' or 'acc', this is converted to one of tf.keras.metrics.BinaryAccuracy, tf.keras.metrics.CategoricalAccuracy, tf.keras.metrics.SparseCategoricalAccuracy based on the loss function used and the model output shape. A similar conversion is done for the strings 'crossentropy' and 'ce'.

loss_weights
Optional list, dictionary, or named vector specifying scalar numeric coefficients 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 a list, it is expected to have a 1:1 mapping to the model’s outputs. If a dict, 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. Defaults to FALSE. If TRUE, this Model’s logic will not be wrapped in a tf.function. Recommended to leave this as NULL unless your Model cannot be run inside a tf.function. run_eagerly=True is not supported when using tf.distribute.experimental.ParameterServerStrategy. If the model’s logic uses tensors in R control flow expressions like if and for, the model is still traceable with tf.function, but you will have to enter a tfautograph::autograph({}) directly.

steps_per_execution
Int. Defaults to 1. The number of batches to run during each tf.function call. Running multiple batches inside a single tf.function call can greatly improve
performance on TPUs or small models with a large Python/R 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 tf.function execution).

Arguments supported for backwards compatibility only.

**target_tensors**
By default, Keras will create a placeholder for the model’s target, which will be fed with the target data during training. If instead you would like to use your own target tensor (in turn, Keras will not expect external data for these targets at training time), you can specify them via the `target_tensors` argument. It should be a single tensor (for a single-output sequential model).

**sample_weight_mode**
If you need to do timestep-wise sample weighting (2D weights), set this to “temporal”. NULL defaults to sample-wise weights (1D). If the model has multiple outputs, you can use a different `sample_weight_mode` on each output by passing a list of modes.

See Also

Other model functions: `evaluate.keras.engine.training.Model()`, `evaluate_generator()`, `fit.keras.engine.training.Model()`, `fit_generator()`, `get_config()`, `get_layer()`, `keras_model_sequential()`, `keras_model()`, `multi_gpu_model()`, `pop_layer()`, `predict.keras.engine.training.Model()`, `predict_generator()`, `predict_on_batch()`, `predict_proba()`, `summary.keras.engine.training.Model()`, `train_on_batch()`

---

<table>
<thead>
<tr>
<th>constraints</th>
<th>Weight constraints</th>
</tr>
</thead>
</table>

**Description**

Functions that impose constraints on weight values.

**Usage**

```python
constraint_maxnorm(max_value = 2, axis = 0)
constraint_nonneg()
constraint_unitnorm(axis = 0)
constraint_minmaxnorm(min_value = 0, max_value = 1, rate = 1, axis = 0)
```
Arguments

max_value  The maximum norm for the incoming weights.

axis  The 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 convolution 2D layer with dim_ordering="tf", the weight tensor has shape rows, cols, input_depth, output_depth, set axis to c(0,1,2) to constrain the weights of each filter tensor of size rows, cols, input_depth.

min_value  The minimum norm for the incoming weights.

rate  The rate for enforcing the constraint: weights will be rescaled to yield (1 - rate) * norm + rate * norm.clip(low, high). 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.

Details

- constraint_maxnorm() constrains the weights incident to each hidden unit to have a norm less than or equal to a desired value.
- constraint_nonneg() constrains the weights to be non-negative
- constraint_unitnorm() constrains the weights incident to each hidden unit to have unit norm.
- constraint_minmaxnorm() constrains the weights incident to each hidden unit to have the norm between a lower bound and an upper bound.

Custom constraints

You can implement your own constraint functions in R. A custom constraint is an R function that takes weights (w) as input and returns modified weights. Note that keras backend() tensor functions (e.g. k_greater_equal()) should be used in the implementation of custom constraints. For example:

```r	nonneg_constraint <- function(w) {
  w * k_cast(k_greater_equal(w, 0), k_floatx())
}

layer_dense(units = 32, input_shape = c(784),
            kernel_constraint = nonneg_constraint)
```

Note that models which use custom constraints cannot be serialized using save_model_hdf5(). Rather, the weights of the model should be saved and restored using save_model_weights_hdf5().

See Also


KerasConstraint
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_input_at(), get_weights(), reset_states()

create_layer
Create a Keras Layer

Description
Create a Keras Layer

Usage
create_layer(layer_class, object, args = list())

Arguments
layer_class Python layer class or R6 class of type KerasLayer
object Object to compose layer with. This is either a keras_model_sequential() to add the layer to, or another Layer which this layer will call.
args List of arguments to layer constructor function

Value
A Keras layer
create_layer_wrapper

Create a Keras Layer wrapper

Description

Create a Keras Layer wrapper

Usage

create_layer_wrapper(LayerClass, modifiers = NULL, convert = TRUE)

Arguments

LayerClass  A R6 or Python class generator that inherits from keras\$layers\$Layer
modifiers    A named list of functions to modify to user-supplied arguments before they are passed on to the class constructor. (e.g., list(units = as.integer))
convert      Boolean, whether the Python class and its methods should by default convert python objects to R objects.
             See guide 'making_new_layers_and_models_via_subclassing.Rmd' for example usage.

Value

An R function that behaves similarly to the builtin keras layer\_* functions. When called, it will create the class instance, and also optionally call it on a supplied argument object if it is present. This enables keras layers to compose nicely with the pipe (\texttt{%>%}).

The R function will arguments taken from the initialize (or \texttt{__init__}) method of the LayerClass.

If LayerClass is an R6 object, this will avoid initializing the python session, so it is safe to use in an R package.

Note

The object parameter can be missing, in which case the layer is created without a connection to an existing graph.
create_wrapper

(Deprecated) Create a Keras Wrapper

Description

R6 classes that inherit from keras\$layers\$Wrapper can now be instantiated directly by create_layer

Usage

create_wrapper(wrapper_class, object, args = list())

Arguments

- **wrapper_class**: R6 class of type KerasWrapper
- **object**: Object to compose layer with. This is either a keras_model_sequential() to add the layer to, or another Layer which this layer will call.
- **args**: List of arguments to layer constructor function

Value

A Keras wrapper

Note

The object parameter can be missing, in which case the layer is created without a connection to an existing graph.

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 backend 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 specify the metric in the call the `load_model_hdf5()`. For example: `load_model_hdf5("my_model.h5",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.

Documentation on the available backend tensor functions can be found at https://keras.rstudio.com/articles/backend.html#backend-functions.

Alternative ways of supplying custom metrics:

- `custom_metric()`: Arbitrary R function.
- `metric_mean_wrapper()`: Wrap an arbitrary R function in a `Metric` instance.
- subclass `keras$metrics$Metric`: see `?Metric` for example.

See Also

Other metrics: `metric_accuracy()`, `metric_auc()`, `metric_binary_accuracy()`, `metric_binary_crossentropy()`, `metric_categorical_accuracy()`, `metric_categorical_crossentropy()`, `metric_categorical_hinge()`, `metric_cosine_similarity()`, `metric_false_negatives()`, `metric_false_positives()`, `metric_hinge()`, `metric_kullback_leibler_divergence()`, `metric_logcosh_error()`, `metric_mean_absolute_error()`, `metric_mean_absolute_percentage_error()`, `metric_mean_iou()`, `metric_mean_relative_error()`, `metric_mean_squared_error()`, `metric_mean_squared_logarithmic_error()`, `metric_mean_tensor()`, `metric_mean_wrapper()`, `metric_mean()`, `metric_poisson()`, `metric_precision_at_recall()`, `metric_precision()`, `metric_recall_at_precision()`, `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()`

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_cifar100(), dataset_cifar10(), 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**

```r
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**

```r
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_cifar100(), dataset_cifar10(), 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 occurring 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.
**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**

```r
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_cifar100()`, `dataset_cifar10()`, `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

```r
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
)
```

```r
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 occurring 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_cifar100()`, `dataset_cifar10()`, `dataset_fashion_mnist()`, `dataset_imdb()`, `dataset_mnist()`
**evaluate.keras.engine.training.Model**

*Evaluate a Keras model*

**Description**

Evaluate a Keras model

**Usage**

```r
## S3 method for class 'keras.engine.training.Model'
evaluate(
  object,
  x = NULL,
  y = NULL,
  batch_size = NULL,
  verbose = 1,
  sample_weight = NULL,
  steps = NULL,
  callbacks = NULL,
  ...
)
```

**Arguments**

- **object**
  Model object to evaluate

- **x**
  Vector, matrix, or array of test data (or list if the model has multiple inputs). If all inputs in the model are named, you can also pass a list mapping input names to data. `x` can be `NULL` (default) if feeding from framework-native tensors (e.g. TensorFlow data tensors). You can also pass a `tfdataset` or a generator returning a list with (inputs, targets) or (inputs, targets, sample_weights).

- **y**
  Vector, matrix, or array of target (label) data (or list if the model has multiple outputs). If all outputs in the model are named, you can also pass a list mapping output names to data. `y` can be `NULL` (default) if feeding from framework-native tensors (e.g. TensorFlow data tensors).

- **batch_size**
  Integer or `NULL`. Number of samples per gradient update. If unspecified, `batch_size` will default to 32.

- **verbose**
  Verbosity mode (0 = silent, 1 = progress bar, 2 = one line per epoch).

- **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. In this case you should make sure to specify `sample_weight_mode="temporal"` in `compile()`.

- **steps**
  Total number of steps (batches of samples) before declaring the evaluation round finished. Ignored with the default value of `NULL`. 
callbacks: List of callbacks to apply during evaluation.

... Unused

Value

Named list of model test loss (or losses for models with multiple outputs) and model metrics.

See Also

Other model functions: `compile.keras.engine.training.Model()`, `evaluate_generator()`, `fit.keras.engine.training.Model()`, `fit_generator()`, `get_config()`, `get_layer()`, `keras_modelSequential()`, `keras_model()`, `multi_gpu_model()`, `pop_layer()`, `predict.keras.engine.training.Model()`, `predict_generator()`, `predict_on_batch()`, `predict_proba()`, `summary.keras.engine.training.Model()`, `train_on_batch()`

evaluate_generator | Evaluates the model on a data generator.

Description

The generator should return the same kind of data as accepted by `test_on_batch()`.

Usage

evaluate_generator(
  object,
  generator,
  steps,
  max_queue_size = 10,
  workers = 1,
  callbacks = NULL
)

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>object</td>
<td>Model object to evaluate</td>
</tr>
<tr>
<td>generator</td>
<td>Generator yielding lists (inputs, targets) or (inputs, targets, sample_weights)</td>
</tr>
<tr>
<td>steps</td>
<td>Total number of steps (batches of samples) to yield from generator before stopping.</td>
</tr>
<tr>
<td>max_queue_size</td>
<td>Maximum size for the generator queue. If unspecified, max_queue_size will default to 10.</td>
</tr>
<tr>
<td>workers</td>
<td>Maximum number of threads to use for parallel processing. Note that parallel processing will only be performed for native Keras generators (e.g. <code>flow_images_from_directory()</code>) as R based generators must run on the main thread.</td>
</tr>
<tr>
<td>callbacks</td>
<td>List of callbacks to apply during evaluation.</td>
</tr>
</tbody>
</table>
Value

Named list of model test loss (or losses for models with multiple outputs) and model metrics.

See Also

Other model functions: `compile.keras.engine.training.Model()`, `evaluate.keras.engine.training.Model()`, `fit.keras.engine.training.Model()`, `fit_generator()`, `get_config()`, `get_layer()`, `keras_model_sequential()`, `keras_model()`, `multi_gpu_model()`, `pop_layer()`, `predict.keras.engine.training.Model()`, `predict_generator()`, `predict_on_batch()`, `predict_proba()`, `summary.keras.engine.training.Model()`, `train_on_batch()`

Description

Serialize a model to disk.

Usage

```r
## S3 method for class 'keras.engine.training.Model'
export_savedmodel(
  object,
  export_dir_base,
  overwrite = TRUE,
  versioned = !overwrite,
  remove_learning_phase = TRUE,
  as_text = FALSE,
  ...
)
```

Arguments

- `object` An R object.
- `export_dir_base` A string containing a directory in which to export the SavedModel.
- `overwrite` Should the `export_dir_base` directory be overwritten?
- `versioned` Should the model be exported under a versioned subdirectory?
- `remove_learning_phase` Should the learning phase be removed by saving and reloading the model? Defaults to `TRUE`.
- `as_text` Whether to write the SavedModel in text format.
- `...` Other arguments passed to `tf.saved_model.save`. (Used only if TensorFlow version >= 2.0)
**Value**

The path to the exported directory, as a string.

---

**Description**

Trains the model for a fixed number of epochs (iterations on a dataset).

**Usage**

```r
## S3 method for class 'keras.engine.training.Model'
fit(
  object,
  x = NULL,
  y = NULL,
  batch_size = NULL,
  epochs = 10,
  verbose = getOption("keras.fit_verbose", default = 1),
  callbacks = NULL,
  view_metrics = getOption("keras.view_metrics", default = "auto"),
  validation_split = 0,
  validation_data = NULL,
  shuffle = TRUE,
  class_weight = NULL,
  sample_weight = NULL,
  initial_epoch = 0,
  steps_per_epoch = NULL,
  validation_steps = NULL,
  ...
)
```

**Arguments**

- `object` Model to train.
- `x` Vector, matrix, or array of training data (or list if the model has multiple inputs). If all inputs in the model are named, you can also pass a list mapping input names to data. `x` can be NULL (default) if feeding from framework-native tensors (e.g. TensorFlow data tensors). You can also pass a `tfdataset` or a generator returning a list with (inputs, targets) or (inputs, targets, sample_weights).
- `y` Vector, matrix, or array of target (label) data (or list if the model has multiple outputs). If all outputs in the model are named, you can also pass a list mapping output names to data. `y` can be NULL (default) if feeding from framework-native tensors (e.g. TensorFlow data tensors).
**batch_size**
Integer or NULL. Number of samples per gradient update. If unspecified, `batch_size` will default to 32.

**epochs**
Number of epochs to train the model. 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.

**verbose**
Verbosity mode (0 = silent, 1 = progress bar, 2 = one line per epoch).

**callbacks**
List of callbacks to be called during training.

**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`. Use the global `keras.view_metrics` option to establish a different default.

**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.

**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. This could be a list `(x_val, y_val)` or a list `(x_val, y_val, val_sample_weights)`. `validation_data` will override `validation_split`.

**shuffle**
shuffle: Logical (whether to shuffle the training data before each epoch) or string (for "batch"). "batch" is a special option for dealing with the limitations of HDF5 data; it shuffles in batch-sized chunks. Has no effect when `steps_per_epoch` is not NULL.

**class_weight**
Optional named list mapping indices (integers) 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.

**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. In this case you should make sure to specify `sample_weight_mode="temporal"` in `compile()`.

**initial_epoch**
Integer, Epoch at which to start training (useful for resuming a previous training run).

**steps_per_epoch**
Total number of steps (batches of samples) before declaring one epoch finished and starting the next epoch. 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.

**validation_steps**
Only relevant if `steps_per_epoch` is specified. Total number of steps (batches of samples) to validate before stopping.

... Unused
**Value**

A history object that contains all information collected during training.

**See Also**

Other model functions: `compile.keras.engine.training.Model()`, `evaluate.keras.engine.training.Model()`, `evaluate_generator()`, `fit_generator()`, `get_config()`, `get_layer()`, `keras_model_sequential()`, `keras_model()`, `multi_gpu_model()`, `pop_layer()`, `predict.keras.engine.training.Model()`, `predict_generator()`, `predict_on_batch()`, `predict_proba()`, `summary.keras.engine.training.Model()`, `train_on_batch()`

---

**fit_generator**

* Fits the model on data yielded batch-by-batch by a generator.

---

**Description**

The generator is run in parallel to the model, for efficiency. For instance, this allows you to do real-time data augmentation on images on CPU in parallel to training your model on GPU.

**Usage**

```r
fit_generator(
  object,          # Keras model object
  generator,       # A generator (e.g. like the one provided by flow_images_from_directory() or a custom R generator function).
  steps_per_epoch, # The output of the generator must be a list of one of these forms:
  epochs = 1,      # - (inputs, targets)
  verbose = 1,     # - (inputs, targets, sample_weights)
  callbacks = NULL,#
  view_metrics = NULL, #
  validation_data = NULL,#
  validation_steps = NULL,#
  class_weight = NULL,#
  max_queue_size = 10,#
  workers = 1,     #
  initial_epoch = 0
)
```

**Arguments**

- **object**
  - Keras model object

- **generator**
  - A generator (e.g. like the one provided by `flow_images_from_directory()` or a custom R generator function).

The output of the generator must be a list of one of these forms:

- (inputs, targets)
- (inputs, targets, sample_weights)
This list (a single output of the generator) makes a single batch. Therefore, all arrays in this list must have the same length (equal to the size of this batch). Different batches may have different sizes. For example, the last batch of the epoch is commonly smaller than the others, if the size of the dataset is not divisible by the batch size. The generator is expected to loop over its data indefinitely. An epoch finishes when `steps_per_epoch` batches have been seen by the model.

**steps_per_epoch**
Total number of steps (batches of samples) to yield from generator before declaring one epoch finished and starting the next epoch. It should typically be equal to the number of samples if your dataset divided by the batch size.

**epochs**
Integer. Number of epochs to train the model. An epoch is an iteration over the entire data provided, as defined by `steps_per_epoch`. 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.

**verbose**
Verbosity mode (0 = silent, 1 = progress bar, 2 = one line per epoch).

**callbacks**
List of callbacks to apply during training.

**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`. Use the global `keras.view_metrics` option to establish a different default.

**validation_data**
This can be either:
- a generator for the validation data
- a list (inputs, targets)
- a list (inputs, targets, sample_weights). 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.

**validation_steps**
Only relevant if `validation_data` is a generator. Total number of steps (batches of samples) to yield from generator before stopping at the end of every epoch. It should typically be equal to the number of samples of your validation dataset divided by the batch size.

**class_weight**
Optional named list mapping class indices (integer) 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.

**max_queue_size**
Maximum size for the generator queue. If unspecified, `max_queue_size` will default to 10.

**workers**
Maximum number of threads to use for parallel processing. Note that parallel processing will only be performed for native Keras generators (e.g. `flow_images_from_directory()`) as R based generators must run on the main thread.

**initial_epoch**
epoch at which to start training (useful for resuming a previous training run)

**Value**
Training history object (invisibly)
See Also

Other model functions: compile.keras.engine.training.Model(), evaluate.keras.engine.training.Model(), evaluate_generator(), fit.keras.engine.training.Model(), get_config(), get_layer(), keras_model_sequential(), keras_model(), multi_gpu_model(), pop_layer(), predict.keras.engine.training.Model(), predict_generator(), predict_on_batch(), predict_proba(), summary.keras.engine.training.Model(), train_on_batch()

---

fit_image_data_generator

*Fit image data generator internal statistics to some sample data.*

**Description**

Required for featurewise_center, featurewise_std_normalization and zca_whitening.

**Usage**

```r
fit_image_data_generator(object, x, augment = FALSE, rounds = 1, seed = NULL)
```

**Arguments**

- **object** `image_data_generator()`
- **x** array, the data to fit on (should have rank 4). In case of grayscale data, the channels axis should have value 1, and in case of RGB data, it should have value 3.
- **augment** Whether to fit on randomly augmented samples
- **rounds** If augment, how many augmentation passes to do over the data
- **seed** random seed.

**See Also**

Other image preprocessing: flow_images_from_dataframe(), flow_images_from_data(), flow_images_from_directory(), image_load(), image_to_array()
**fit_text_tokenizer**

Update tokenizer internal vocabulary based on a list of texts or list of sequences.

**Description**

Update tokenizer internal vocabulary based on a list of texts or list of sequences.

**Usage**

```
fit_text_tokenizer(object, x)
```

**Arguments**

- `object`: Tokenizer returned by `text_tokenizer()`.
- `x`: Vector/list of strings, or a generator of strings (for memory-efficiency); Alternatively a list of "sequence" (a sequence is a list of integer word indices).

**Note**

Required before using `texts_to_sequences()`, `texts_to_matrix()`, or `sequences_to_matrix()`.

**See Also**

Other text tokenization: `save_text_tokenizer()`, `sequences_to_matrix()`, `text_tokenizer()`, `texts_to_matrix()`, `texts_to_sequences_generator()`, `texts_to_sequences()`

---

**flow_images_from_data**

Generates batches of augmented/normalized data from image data and labels.

**Description**

Generates batches of augmented/normalized data from image data and labels.

**Usage**

```
flow_images_from_data(
  x,
  y = NULL,
  generator = image_data_generator(),
  batch_size = 32,
  shuffle = TRUE,
  sample_weight = NULL,
  seed = NULL,
  save_to_dir = NULL,
```
Arguments

x  data. Should have rank 4. In case of grayscale data, the channels axis should have value 1, and in case of RGB data, it should have value 3.
y  labels (can be NULL if no labels are required)
generator  Image data generator to use for augmenting/normalizing image data.
batch_size  int (default: 32).
shuffle  boolean (default: TRUE).
sample_weight  Sample weights.
seed  int (default: NULL).
save_to_dir  NULL or str (default: NULL). This allows you to optionally specify a directory to which to save the augmented pictures being generated (useful for visualizing what you are doing).
save_prefix  str (default: "). Prefix to use for filenames of saved pictures (only relevant if save_to_dir is set).
save_format  one of "png", "jpeg" (only relevant if save_to_dir is set). Default: "png".
subset  Subset of data ("training" or "validation") if validation_split is set in image_data_generator().

Details

Yields batches indefinitely, in an infinite loop.

Yields

(x, y) where x is an array of image data and y is a array of corresponding labels. The generator loops indefinitely.

See Also

Other image preprocessing: fit_image_data_generator(), flow_images_from_dataframe(), flow_images_from_directory(), image_load(), image_to_array()
flow_images_from_dataframe

Takes the dataframe and the path to a directory and generates batches of augmented/normalized data.

Description

Takes the dataframe and the path to a directory and generates batches of augmented/normalized data.

Usage

flow_images_from_dataframe(
  dataframe,
  directory = NULL,
  x_col = "filename",
  y_col = "class",
  generator = image_data_generator(),
  target_size = c(256, 256),
  color_mode = "rgb",
  classes = NULL,
  class_mode = "categorical",
  batch_size = 32,
  shuffle = TRUE,
  seed = NULL,
  save_to_dir = NULL,
  save_prefix = "",
  save_format = "png",
  subset = NULL,
  interpolation = "nearest",
  drop_duplicates = NULL
)

Arguments

dataframe data.frame containing the filepaths relative to directory (or absolute paths if directory is NULL) of the images in a character column. It should include other column/s depending on the class_mode:

- if class_mode is "categorical" (default value) it must include the y_col column with the class/es of each image. Values in column can be character/list if a single class or list if multiple classes.
- if class_mode is "binary" or "sparse" it must include the given y_col column with class values as strings.
- if class_mode is "other" it should contain the columns specified in y_col.
- if class_mode is "input" or NULL no extra column is needed.
**flow_images_from_dataframe**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>directory</td>
<td>character, path to the directory to read images from. If NULL, data in x_col column should be absolute paths.</td>
</tr>
<tr>
<td>x_col</td>
<td>character, column in dataframe that contains the filenames (or absolute paths if directory is NULL).</td>
</tr>
<tr>
<td>y_col</td>
<td>string or list, column/s in dataframe that has the target data.</td>
</tr>
<tr>
<td>generator</td>
<td>Image data generator to use for augmenting/normalizing image data.</td>
</tr>
<tr>
<td>target_size</td>
<td>Either NULL (default to original size) or integer vector (img_height, img_width).</td>
</tr>
<tr>
<td>color_mode</td>
<td>one of &quot;grayscale&quot;, &quot;rgb&quot;. Default: &quot;rgb&quot;. Whether the images will be converted to have 1 or 3 color channels.</td>
</tr>
<tr>
<td>classes</td>
<td>optional list of classes (e.g. c('dogs', 'cats')). Default: NULL If not provided, the list of classes will be automatically inferred from the y_col, which will map to the label indices, will be alphanumeric). The dictionary containing the mapping from class names to class indices can be obtained via the attribute class_indices.</td>
</tr>
<tr>
<td>class_mode</td>
<td>one of &quot;categorical&quot;, &quot;binary&quot;, &quot;sparse&quot;, &quot;input&quot;, &quot;other&quot; or None. Default: &quot;categorical&quot;. Mode for yielding the targets: • &quot;binary&quot;: 1D array of binary labels, • &quot;categorical&quot;: 2D array of one-hot encoded labels. Supports multi-label output. • &quot;sparse&quot;: 1D array of integer labels, • &quot;input&quot;: images identical to input images (mainly used to work with autoencoders), • &quot;other&quot;: array of y_col data, • &quot;multi_output&quot;: allow to train a multi-output model. Y is a list or a vector. NULL, no targets are returned (the generator will only yield batches of image data, which is useful to use in predict_generator()).</td>
</tr>
<tr>
<td>batch_size</td>
<td>int (default: 32).</td>
</tr>
<tr>
<td>shuffle</td>
<td>boolean (default: TRUE).</td>
</tr>
<tr>
<td>seed</td>
<td>int (default: NULL).</td>
</tr>
<tr>
<td>save_to_dir</td>
<td>NULL or str (default: NULL). This allows you to optionally specify a directory to which to save the augmented pictures being generated (useful for visualizing what you are doing).</td>
</tr>
<tr>
<td>save_prefix</td>
<td>str (default: &quot;&quot;). Prefix to use for filenames of saved pictures (only relevant if save_to_dir is set).</td>
</tr>
<tr>
<td>save_format</td>
<td>one of &quot;png&quot;, &quot;jpeg&quot; (only relevant if save_to_dir is set). Default: &quot;png&quot;.</td>
</tr>
<tr>
<td>subset</td>
<td>Subset of data (&quot;training&quot; or &quot;validation&quot;) if validation_split is set in image_data_generator().</td>
</tr>
<tr>
<td>interpolation</td>
<td>Interpolation method used to resample the image if the target size is different from that of the loaded image. Supported methods are &quot;nearest&quot;, &quot;bilinear&quot;, and &quot;bicubic&quot;. If PIL version 1.1.3 or newer is installed, &quot;lanczos&quot; is also supported. If PIL version 3.4.0 or newer is installed, &quot;box&quot; and &quot;hamming&quot; are also supported. By default, &quot;nearest&quot; is used.</td>
</tr>
<tr>
<td>drop_duplicates</td>
<td>(deprecated in TF &gt;= 2.3) Boolean, whether to drop duplicate rows based on filename. The default value is TRUE.</td>
</tr>
</tbody>
</table>
flow_images_from_directory

Details

Yields batches indefinitely, in an infinite loop.

Yields

(x, y) where x is an array of image data and y is a array of corresponding labels. The generator loops indefinitely.

Note

This function requires that pandas (python module) is installed in the same environment as tensorflow and keras.

If you are using r-tensorflow (the default environment) you can install pandas by running reticulate::virtualenv_install("pandas", envname = "r-tensorflow") or reticulate::conda_install("pandas", envname = "r-tensorflow") depending on the kind of environment you are using.

See Also

Other image preprocessing: fit_image_data_generator(), flow_images_from_data(), flow_images_from_directory(), image_load(), image_to_array()

flow_images_from_directory

Generates batches of data from images in a directory (with optional augmented/normalized data)

Description

Generates batches of data from images in a directory (with optional augmented/normalized data)

Usage

flow_images_from_directory(
  directory,
  generator = image_data_generator(),
  target_size = c(256, 256),
  color_mode = "rgb",
  classes = NULL,
  class_mode = "categorical",
  batch_size = 32,
  shuffle = TRUE,
  seed = NULL,
  save_to_dir = NULL,
  save_prefix = "",
  save_format = "png",
  follow_links = FALSE,
  subset = NULL,
interpolation = "nearest"
)

Arguments

directory
path to the target directory. It should contain one subdirectory per class. Any
PNG, JPG, BMP, PPM, or TIF images inside each of the subdirectories directory
tree will be included in the generator. See this script for more details.

generator
Image data generator (default generator does no data augmentation/normalization
transformations)

target_size
integer vector, default: c(256,256). The dimensions to which all images found
will be resized.

color_mode
one of "grayscale", "rgb". Default: "rgb". Whether the images will be converted
to have 1 or 3 color channels.

classes
optional list of class subdirectories (e.g. c('dogs', 'cats')). Default: NULL. If
not provided, the list of classes will be automatically inferred (and the order of
the classes, which will map to the label indices, will be alphanumeric).

class_mode
one of "categorical", "binary", "sparse" or NULL. Default: "categorical". Deter-
mines the type of label arrays that are returned: "categorical" will be 2D one-hot
encoded labels, "binary" will be 1D binary labels, "sparse" will be 1D integer la-
bels. If NULL, no labels are returned (the generator will only yield batches of im-
age data, which is useful to use predict_generator(), evaluate_generator(),
etc.).

batch_size
int (default: 32).

shuffle
boolean (default: TRUE).

seed
int (default: NULL).

save_to_dir
NULL or str (default: NULL). This allows you to optionally specify a directory
to which to save the augmented pictures being generated (useful for visualizing
what you are doing).

save_prefix
str (default: ""). Prefix to use for filenames of saved pictures (only relevant if
save_to_dir is set).

save_format
one of "png", "jpeg" (only relevant if save_to_dir is set). Default: "png".

follow_links
whether to follow symlinks inside class subdirectories (default: FALSE)

subset
Subset of data ("training" or "validation") if validation_split is set in
image_data_generator().

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 sup-
ported. By default, "nearest" is used.

Details

Yields batches indefinitely, in an infinite loop.
Yields
(x, y) where x is an array of image data and y is a array of corresponding labels. The generator loops indefinitely.

See Also
Other image preprocessing: fit_image_data_generator(), flow_images_from_dataframe(), flow_images_from_data(), image_load(), image_to_array()

__Description__

Freeze weights in a model or layer so that they are no longer trainable.

__Usage__

```r
freeze_weights(object, from = NULL, to = NULL)
unfreeze_weights(object, from = NULL, to = 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

__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.

__Examples__

```r
## Not run:
# 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)

# create a composite model that includes the base + more layers
model <- keras_model_sequential() %>%
  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(lr = 2e-5),
  metrics = c("accuracy")
)

# 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(lr = 2e-5),
  metrics = c("accuracy")
)

## End(Not run)

generator_next

Retrieves the next item from a generator

**Description**

Use to retrieve items from generators (e.g. `image_data_generator()`). Will return either the next item or NULL if there are no more items.

**Usage**

generator_next(generator, completed = NULL)

**Arguments**

- **generator**  
  Generator

- **completed**  
  Sentinel value to return from `generator_next()` if the iteration completes (defaults to NULL but can be any R value you specify).
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 re-instantiated 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

```python
get_config(object)
from_config(config)
```

Arguments

- **object**  
  Layer or model object
- **config**  
  Object with layer or model configuration

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. Therefore, if you want to save and restore a model across sessions, you can use the `model_to_json()` or `model_to_yaml()` functions (for model configuration only, not weights) or the `save_model_hdf5()` function to save the model configuration and weights to a file.

See Also


Other layer methods: `count_params()`, `get_input_at()`, `get_weights()`, `reset_states()`
**get_file**

Downloads a file from a URL if it not already in the cache.

**Description**

Passing the MD5 hash will verify the file after download as well as if it is already present in the cache.

**Usage**

```r
get_file(
  fname,
  origin,
  file_hash = NULL,
  cache_subdir = "datasets",
  hash_algorithm = "auto",
  extract = FALSE,
  archive_format = "auto",
  cache_dir = NULL,
  untar = FALSE
)
```

**Arguments**

- **fname**: Name of the file. If an absolute path /path/to/file.txt is specified the file will be saved at that location.
- **origin**: Original URL of the file.
- **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 /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 None. 'tar' includes tar, tar.gz, and tar.bz files. The default 'auto' is ('tar', 'zip'). None or an empty list will return no matches found.
- **cache_dir**: Location to store cached files, when NULL it defaults to the Keras configuration directory.
- **untar**: Deprecated in favor of 'extract'. boolean, whether the file should be decompressed

**Value**

Path to the downloaded file
**get_input_at**

*Retrieve tensors for layers with multiple nodes*

---

**Description**

Whenever you are calling a layer on some input, you are creating a new tensor (the output of the layer), and you are adding a "node" to the layer, linking the input tensor to the output tensor. When you are calling the same layer multiple times, that layer owns multiple nodes indexed as 1, 2, 3. These functions enable you to retrieve various tensor properties of layers with multiple nodes.

**Usage**

- `get_input_at(object, node_index)`
- `get_output_at(object, node_index)`
- `get_input_shape_at(object, node_index)`
- `get_output_shape_at(object, node_index)`
- `get_input_mask_at(object, node_index)`
- `get_output_mask_at(object, node_index)`

**Arguments**

- `object`: Layer or model object
- `node_index`: Integer, index of the node from which to retrieve the attribute. E.g. `node_index = 1` will correspond to the first time the layer was called.

**Value**

A tensor (or list of tensors if the layer has multiple inputs/outputs).

**See Also**

Other layer methods: `count_params()`, `get_config()`, `get_weights()`, `reset_states()`
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)

Value
A layer instance.

See Also
Other model functions: compile.keras.engine.training.Model(), evaluate.keras.engine.training.Model(), evaluate_generator(), fit.keras.engine.training.Model(), fit_generator(), get_config(), keras_model_sequential(), keras_model(), multi_gpu_model(), pop_layer(), predict.keras.engine.training.Model(), predict_generator(), predict_on_batch(), predict_proba(), summary.keras.engine.training.Model(), train_on_batch()

get_vocabulary

Get the vocabulary for text vectorization layers

Description
Get the vocabulary for text vectorization layers

Usage
get_vocabulary(object)

Arguments
- object: a text vectorization layer

See Also
set_vocabulary()
### get_weights

Layer/Model weights as R arrays

**Description**

Layer/Model weights as R arrays

**Usage**

get_weights(object)

set_weights(object, weights)

**Arguments**

- object: Layer or model object
- weights: Weights as R array

**See Also**

Other model persistence: `model_to_json()`, `model_to_yaml()`, `save_model_hdf5()`, `save_model_tf()`, `save_model_weights_hdf5()`, `serialize_model()`

Other layer methods: `count_params()`, `get_config()`, `get_input_at()`, `reset_states()`

---

### hdf5_matrix

Representation of HDF5 dataset to be used instead of an R array

**Description**

Representation of HDF5 dataset to be used instead of an R array

**Usage**

hdf5_matrix(datapath, dataset, start = 0, end = NULL, normalizer = NULL)

**Arguments**

- datapath: string, path to a HDF5 file
- dataset: string, name of the HDF5 dataset in the file specified in datapath
- start: int, start of desired slice of the specified dataset
- end: int, end of desired slice of the specified dataset
- normalizer: function to be called on data when retrieved
Details
Providing start and end allows use of a slice of the dataset.

Optionally, a normalizer function (or lambda) can be given. This will be called on every slice of data retrieved.

Value
An array-like HDF5 dataset.

imagenet_decode_predictions
Decodes the prediction of an ImageNet model.

Description
Decodes the prediction of an ImageNet model.

Usage
imagenet_decode_predictions(preds, top = 5)

Arguments
preds Tensor encoding a batch of predictions.
top integer, how many top-guesses to return.

Value
List of data frames with variables class_name, class_description, and score (one data frame per sample in batch input).

imagenet_preprocess_input
Preprocesses a tensor or array encoding a batch of images.

Description
Preprocesses a tensor or array encoding a batch of images.

Usage
imagenet_preprocess_input(x, data_format = NULL, mode = "caffe")
Arguments

- **x**: Input Numpy or symbolic tensor, 3D or 4D.
- **data_format**: Data format of the image tensor/array.
- **mode**: One of "caffe", "tf", or "torch"
  - caffe: will convert the images from RGB to BGR, then will zero-center each color channel with respect to the ImageNet dataset, without scaling.
  - tf: will scale pixels between -1 and 1, sample-wise.
  - torch: will scale pixels between 0 and 1 and then will normalize each channel with respect to the ImageNet dataset.

Value

Preprocessed tensor or array.

---

**image_dataset_from_directory**

*Create a dataset from a directory*

Description

Generates a tf.data.Dataset from image files in a directory. If your directory structure is:

Usage

```r
image_dataset_from_directory(
  directory,
  labels = "inferred",
  label_mode = "int",
  class_names = NULL,
  color_mode = "rgb",
  batch_size = 32,
  image_size = c(256, 256),
  shuffle = TRUE,
  seed = NULL,
  validation_split = NULL,
  subset = NULL,
  interpolation = "bilinear",
  follow_links = FALSE
)
```

Arguments

- **directory**: Directory where the data is located. If labels is "inferred", it should contain sub-directories, each containing images for a class. Otherwise, the directory structure is ignored.
labels Either "inferred" (labels are generated from the directory structure), 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 '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). - None (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". Default: "rgb". Whether the images will be converted to have 1, 3, or 4 channels.

batch_size Size of the batches of data. Default: 32.

image_size Size to resize images to after they are read from disk. 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. Default: TRUE. If set to FALSE, sorts the data in alphanumerical 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 One of "training" or "validation". Only used if validation_split is set.

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 visits subdirectories pointed to by symlinks. Defaults to FALSE.

---

**image_data_generator**

*Generate batches of image data with real-time data augmentation. The data will be looped over (in batches).*

**Description**

Generate batches of image data with real-time data augmentation. The data will be looped over (in batches).

**Usage**

```r
image_data_generator(
  featurewise_center = FALSE,
  samplewise_center = FALSE,
  featurewise_std_normalization = FALSE,
```
samplewise_std_normalization = FALSE,
zca_whitening = FALSE,
zca_epsilon = 1e-06,
rotation_range = 0,
width_shift_range = 0,
height_shift_range = 0,
brightness_range = NULL,
shear_range = 0,
zoom_range = 0,
channel_shift_range = 0,
fill_mode = "nearest",
cval = 0,
horizontal_flip = FALSE,
vertical_flip = FALSE,
rescale = NULL,
preprocessing_function = NULL,
data_format = NULL,
validation_split = 0
)

Arguments

featurewise_center
   Set input mean to 0 over the dataset, feature-wise.
samplewise_center
   Boolean. Set each sample mean to 0.
featurewise_std_normalization
   Divide inputs by std of the dataset, feature-wise.
samplewise_std_normalization
   Divide each input by its std.
zca_whitening
   apply ZCA whitening.
zca_epsilon
   Epsilon for ZCA whitening. Default is 1e-6.
rotation_range
   degrees (0 to 180).
width_shift_range
   fraction of total width.
height_shift_range
   fraction of total height.
brightness_range
   the range of brightness to apply
shear_range
   shear intensity (shear angle in radians).
zoom_range
   amount of zoom. if scalar z, zoom will be randomly picked in the range [1-
z, 1+z]. A sequence of two can be passed instead to select this range.
channel_shift_range
   shift range for each channels.
fill_mode
   One of "constant", "nearest", "reflect" or "wrap". Points outside the boundaries
   of the input are filled according to the given mode:
• "constant": kkkkkkkk|abcd|kkkkkkkk (cval=k)
• "nearest": aaaaaaaaa|abcd|dddddddd
• "reflect": abcdabca|abcd|dcbaabcd
• "wrap": abcdabcd|abcd|abcdabcd
cval value used for points outside the boundaries when fill_mode is 'constant’. Default is 0.
horizontal_flip whether to randomly flip images horizontally.
vertical_flip whether to randomly flip images vertically.
rescale rescaling factor. If NULL or 0, no rescaling is applied, otherwise we multiply the data by the value provided (before applying any other transformation).
preprocessing_function function that will be implied on each input. The function will run before any other modification on it. The function should take one argument: one image (tensor with rank 3), and should output a tensor with the same shape.
data_format 'channels_first' or 'channels_last'. In 'channels_first' mode, the channels dimension (the depth) is at index 1, in 'channels_last' mode it is 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".
validation_split fraction of images reserved for validation (strictly between 0 and 1).

image_load (path, grayscale = FALSE, target_size = NULL, interpolation = "nearest")

Arguments

path Path to image file
grayscale Boolean, whether to load the image as grayscale.
target_size Either NULL (default to original size) or integer vector (img_height, img_width).
**Value**

A PIL Image instance.

**See Also**

Other image preprocessing: `fit_image_data_generator()`, `flow_images_from_dataframe()`, `flow_images_from_data()`, `flow_images_from_directory()`, `image_to_array()`

**image_to_array**

3D array representation of images

**Description**

3D array that represents an image with dimensions (height,width,channels) or (channels,height,width) depending on the data_format.

**Usage**

```r
image_to_array(img, data_format = c("channels_last", "channels_first"))
```

```r
image_array_resize(
  img,
  height,
  width,
  data_format = c("channels_last", "channels_first")
)
```

```r
image_array_save(
  img,
  path,
  data_format = NULL,
  file_format = NULL,
  scale = TRUE
)
```

**Arguments**

- **img**: Image
- **data_format**: Image data format ("channels_last" or "channels_first")
- **height**: Height to resize to
**Description**

Obtain a reference to the Python module used for the implementation of Keras.

**Usage**

```
implementation()
```

**Details**

There are currently two Python modules which implement Keras:

- `keras` ("keras")
- `tensorflow.keras` ("tensorflow")

This function returns a reference to the implementation being currently used by the keras package. The default implementation is "keras". You can override this by setting the KERAS_IMPLEMENTATION environment variable to "tensorflow".

**Value**

Reference to the Python module used for the implementation of Keras.
initializer_constant

Initializes tensors to a constant value.

**Usage**

```python
initializer_constant(value = 0)
```

**Arguments**

**value**
float; the value of the generator tensors.

**See Also**

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

Glorot normal initializer, also called Xavier normal initializer.

**Description**

It draws samples from a truncated normal distribution centered on 0 with \( \text{stddev} = \sqrt{\frac{2}{\text{fan}_\text{in} + \text{fan}_\text{out}}} \) where \( \text{fan}_\text{in} \) is the number of input units in the weight tensor and \( \text{fan}_\text{out} \) is the number of output units in the weight tensor.

**Usage**

```python
initializer_glorot_normal(seed = NULL)
```

**Arguments**

**seed**
Integer used to seed the random generator.

**References**

initializer_glorot_uniform

Glorot uniform initializer, also called Xavier uniform initializer.

Description

It draws samples from a uniform distribution within \(-\text{limit}, \text{limit}\) where \(\text{limit} = \sqrt{6 / (\text{fan\_in} + \text{fan\_out})}\) where \(\text{fan\_in}\) is the number of input units in the weight tensor and \(\text{fan\_out}\) is the number of output units in the weight tensor.

Usage

initializer_glorot_uniform(seed = NULL)

Arguments

seed

Integer used to seed the random generator.

References


See Also

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 \(\text{stddev} = \sqrt{2 / \text{fan\_in}}\) where \(\text{fan\_in}\) is the number of input units in the weight tensor.

Usage

initializer_he_normal(seed = NULL)
initializer_he_uniform

Arguments

seed  Integer used to seed the random generator.

References

He et al., https://arxiv.org/abs/1502.01852

See Also

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_he_uniform

He uniform variance scaling initializer.

Description

It draws samples from a uniform distribution within -limit, limit where limit = sqrt(6 / fan_in) where fan_in is the number of input units in the weight tensor.

Usage

initializer_he_uniform(seed = NULL)

Arguments

seed  Integer used to seed the random generator.

References

He et al., https://arxiv.org/abs/1502.01852

See Also

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 use for square 2D matrices.

**Usage**

```r
initializer_identity(gain = 1)
```

**Arguments**

- **gain**  
  Multiplicative factor to apply to the identity matrix

**See Also**

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**

It draws samples from a truncated normal distribution centered on 0 with $\text{stddev} = \sqrt{1 / \text{fan_in}}$ where `fan_in` is the number of input units in the weight tensor.

**Usage**

```r
initializer_lecun_normal(seed = NULL)
```

**Arguments**

- **seed**  
  A Python integer. Used to seed the random generator.

**References**

- Self-Normalizing Neural Networks
- Efficient Backprop, *LeCun, Yann et al. 1998*
See Also

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**

It draws samples from a uniform distribution within \(-\text{limit}, \text{limit}\) where \(\text{limit} = \sqrt{3 / \text{fan\_in}}\)
where \text{fan\_in} is the number of input units in the weight tensor.

**Usage**

```
initializer_lecun_uniform(seed = NULL)
```

**Arguments**

- **seed** Integer used to seed the random generator.

**References**

LeCun 98, Efficient Backprop,

See Also

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**

Initializer that generates tensors initialized to 1.

**Usage**

```
initializer_ones()
```
**initializer_orthogonal**

Initializer that generates a random orthogonal matrix.

**Description**

Initializer that generates a random orthogonal matrix.

**Usage**

```
initializer_orthogonal(gain = 1, seed = NULL)
```

**Arguments**

- **gain**: Multiplicative factor to apply to the orthogonal matrix.
- **seed**: Integer used to seed the random generator.

**References**


**See Also**

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_random_normal

Initializer that generates tensors with a normal distribution.

Description

Initializer that generates tensors with a normal distribution.

Usage

initializer_random_normal(mean = 0, stddev = 0.05, seed = NULL)

Arguments

- **mean**: Mean of the random values to generate.
- **stddev**: Standard deviation of the random values to generate.
- **seed**: Integer used to seed the random generator.

See Also

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

Initializer that generates tensors with a uniform distribution.

Description

Initializer that generates tensors with a uniform distribution.

Usage

initializer_random_uniform(minval = -0.05, maxval = 0.05, seed = NULL)

Arguments

- **minval**: Lower bound of the range of random values to generate.
- **maxval**: Upper bound of the range of random values to generate. Defaults to 1 for float types.
- **seed**: seed
initializer_truncated_normal

*Initializer that generates a truncated normal distribution.*

**Description**

These values are similar to values from an `initializer_random_normal()` except that values more than two standard deviations from the mean are discarded and re-drawn. This is the recommended initializer for neural network weights and filters.

**Usage**

```r
initializer_truncated_normal(mean = 0, stddev = 0.05, seed = NULL)
```

**Arguments**

- `mean` Mean of the random values to generate.
- `stddev` Standard deviation of the random values to generate.
- `seed` Integer used to seed the random generator.

**See Also**

Other initializers: `initializer_constant()`, `initializer_glorot_normal()`, `initializer_glorot_uniform()`, `initializer_he_normal()`, `initializer_he_uniform()`, `initializer_identity()`, `initializer_identity()`, `initializer_ones()`, `initializer_orthogonal()`, `initializer_random_normal()`, `initializer_random_uniform()`, `initializer_truncated_normal()`, `initializer_variance_scaling()`, `initializer_zeros()`

initializer_variance_scaling

*Initializer capable of adapting its scale to the shape of weights.*

**Description**

With distribution="normal", samples are drawn from a truncated normal distribution centered on zero, with stddev = \(\sqrt{\text{scale} / n}\) 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"
Usage

initializer_variance_scaling(
  scale = 1,
  mode = c("fan_in", "fan_out", "fan_avg"),
  distribution = c("normal", "uniform", "truncated_normal", "untruncated_normal"),
  seed = NULL
)

Arguments

scale Scaling factor (positive float).
mode One of "fan_in", "fan_out", "fan_avg".
distribution One of "truncated_normal", "untruncated_normal" and "uniform". For back-
ward compatibility, "normal" will be accepted and converted to "untruncated_normal".
seed Integer used to seed the random generator.

details

With distribution="uniform", samples are drawn from a uniform distribution within -limit, limit,
with limit = sqrt(3 * scale / n).

See Also

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()

See Also

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 TensorFlow and Keras, including all python dependencies

Description

This function will install Tensorflow and all Keras dependencies. This is a thin wrapper around tensorflow::install_tensorflow(), with the only difference being that this includes by default additional extra packages that keras expects, and the default version of tensorflow installed by install_keras() may at times be different from the default installed install_tensorflow(). The default version of tensorflow installed by install_keras() is "2.6".

Usage

install_keras(
  method = c("auto", "virtualenv", "conda"),
  conda = "auto",
  version = "default",
  tensorflow = version,
  extra_packages = NULL,
  ...
)

Arguments

method Installation method. By default, "auto" automatically finds a method that will work in the local environment. Change the default to force a specific installation method. Note that the "virtualenv" method is not available on Windows.

conda The path to a conda executable. Use "auto" to allow reticulate to automatically find an appropriate conda binary. See Finding Conda for more details.

version TensorFlow version to install. Valid values include:
  • "default" installs 2.6
  • "release" installs the latest release version of tensorflow (which may be incompatible with the current version of the R package)
  • A version specification like "2.4" or "2.4.0". Note that if the patch version is not supplied, the latest patch release is installed (e.g., "2.4" today installs version "2.4.2")
  • nightly for the latest available nightly build.
  • To any specification, you can append "-cpu" to install the cpu version only of the package (e.g., "2.4-cpu")
  • The full URL or path to a installer binary or python *.whl file.

tensorflow Synonym for version. Maintained for backwards.

extra_packages Additional Python packages to install along with TensorFlow.

... other arguments passed to reticulate::conda_install() or reticulate::virtualenv_install(), depending on the method used.
is_keras_available

Details

The default additional packages are: tensorflow-hub, scipy, requests, pyyaml, Pillow, h5py, pandas, with their versions potentially constrained for compatibility with the requested tensorflow version.

Custom Installation

install_tensorflow() or keras::install_keras() isn’t required to use tensorflow with the package. If you manually configure a python environment with the required dependencies, you can tell R to use it by pointing reticulate at it, commonly by setting an environment variable:

Sys.setenv("RETICULATE_PYTHON" = "~/path/to/python-env/bin/python")

Apple Silicon

Tensorflow on Apple Silicon is not officially supported by the tensorflow maintainers. It is known that there can be issues running the official Tensorflow package under Rosetta as well. Fortunately, for the time being Apple has published a custom version of Tensorflow compatible with M1 macs. Installation instructions can be found at: https://developer.apple.com/metal/tensorflow-plugin/. Please note that this is an experimental build of both python and tensor-flow. After following the instructions provided by Apple, you can advise reticulate to use that python installation by placing the following in your .Renviron file:

RETICULATE_PYTHON = "~/miniforge3/bin/python"

Additional Packages

If you wish to add additional PyPI packages to your Keras / TensorFlow environment you can either specify the packages in the extra_packages argument of install_tensorflow() or install_keras(), or alternatively install them into an existing environment using the reticulate::py_install() function. Note that install_keras() includes a set of additional python packages by default, see ?keras::install_keras for details.

See Also

tensorflow::install_tensorflow()

is_keras_available Check if Keras is Available

Description

Probe to see whether the Keras python package is available in the current system environment.

Usage

is_keras_available(version = NULL)
**Arguments**

- **version**  Minimum required version of Keras (defaults to NULL, no required version).

**Value**

Logical indicating whether Keras (or the specified minimum version of Keras) is available.

**Examples**

```r
## Not run:
# testthat utility for skipping tests when Keras isn't available
skip_if_no_keras <- function(version = NULL) {
  if (!is_keras_available(version))
    skip("Required keras version not available for testing")
}

# use the function within a test
test_that("keras function works correctly", {
  skip_if_no_keras()
  # test code here
})

## End(Not run)
```

---

**Keras**  
**Main Keras module**

**Description**

The `keras` module object is the equivalent of `keras <- tensorflow::tf$keras` and provided mainly as a convenience.

**Usage**

`keras`

**Format**

An object of class `python.builtin.module` (inherits from `python.builtin.object`) of length 6.

**Value**

the keras Python module
**KerasCallback**

*(Deprecated) Base R6 class for Keras callbacks*

---

**Description**

New custom callbacks implemented as R6 classes are encouraged to inherit from `keras$callbacks$Callback` directly.

**Format**

An `R6Class` generator object

**Details**

The `logs` named list that callback methods take as argument will contain keys for quantities relevant to the current batch or epoch.

Currently, the `fit.keras.engine.training.Model()` method for sequential models will include the following quantities in the `logs` that it passes to its callbacks:

- **on.epoch.end**: logs include `acc` and `loss`, and optionally include `val_loss` (if validation is enabled in `fit`), and `val_acc` (if validation and accuracy monitoring are enabled).
- **on.batch.begin**: logs include `size`, the number of samples in the current batch.
- **on.batch.end**: logs include `loss`, and optionally `acc` (if accuracy monitoring is enabled).

**Value**

`KerasCallback`.

**Fields**

params Named list with training parameters (e.g., verbosity, batch size, number of epochs...).

model Reference to the Keras model being trained.

**Methods**

- `on.epoch.begin(epoch, logs)` Called at the beginning of each epoch.
- `on.epoch.end(epoch, logs)` Called at the end of each epoch.
- `on.batch.begin(batch, logs)` Called at the beginning of each batch.
- `on.batch.end(batch, logs)` Called at the end of each batch.
- `on.train.begin(logs)` Called at the beginning of training.
- `on.train.end(logs)` Called at the end of training.
KerasConstraint

Examples

```r
## Not run:
library(keras)

LossHistory <- R6::R6Class("LossHistory",
  inherit = KerasCallback,
  public = list(
    losses = NULL,
    on_batch_end = function(batch, logs = list()) {
      self$losses <- c(self$losses, logs[["loss"]])
    }
  )
)

## End(Not run)
```

KerasConstraint (Deprecated) Base R6 class for Keras constraints

Description

New custom constraints are encouraged to subclass `keras$constraints$Constraint` directly.

Format

An `R6Class` generator object

Details

You can implement a custom constraint either by creating an R function that accepts a weights \( w \) parameter, or by creating an R6 class that derives from `KerasConstraint` and implements a `call` method.

Methods

call(w) Constrain the specified weights.

Note

Models which use custom constraints cannot be serialized using `save_model_hdf5()`. Rather, the weights of the model should be saved and restored using `save_model_weights_hdf5()`.

See Also

customs
Examples

```r
## Not run:
CustomNonNegConstraint <- R6::R6Class(
  "CustomNonNegConstraint",
  inherit = KerasConstraint,
  public = list(
    call = function(x) {
      w * k_cast(k_greater_equal(w, 0), k_floatx())
    }
  )
)

layer_dense(units = 32, input_shape = c(784),
            kernel_constraint = CustomNonNegConstraint$new())
## End(Not run)
```

---

KerasLayer

*(Deprecated) Base R6 class for Keras layers*

---

Description

Custom R6 layers can now inherit directly from `keras$layers$Layer` or other layers.

Format

An `R6Class` generator object #'

Value

KerasLayer.

Methods

- `build(input_shape)` Creates the layer weights (must be implemented by all layers that have weights)
- `call(inputs,mask)` Call the layer on an input tensor.
- `compute_output_shape(input_shape)` Compute the output shape for the layer.
- `add_loss(losses, inputs)` Add losses to the layer.
- `add_weight(name,shape,dtype,initializer,regularizer,trainable,constraint)` Adds a weight variable to the layer.
KerasWrapper

*Deprecated* Base R6 class for Keras wrappers

### Description

Instead of inheriting from the proxy class KerasWrapper and using `create_wrapper` to create instances, new R6 custom classes are encouraged to inherit directly from `keras$layers$Wrapper` and use `create_layer` to create instances.

### Format

An `R6Class` generator object

### Value

KerasWrapper.

### Methods

- `build(input_shape)` Builds the wrapped layer. Subclasses can extend this to perform custom operations on that layer.
- `call(inputs, mask)` Calls the wrapped layer on an input tensor.
- `compute_output_shape(input_shape)` Computes the output shape for the wrapped layer.
- `add_loss(losses, inputs)` Subclasses can use this to add losses to the wrapped layer.
- `add_weight(name, shape, dtype, initializer, regularizer, trainable, constraint)` Subclasses can use this to add weights to the wrapped layer.

kera_array

*Keras array object*

### Description

Convert an R vector, matrix, or array object to an array that has the optimal in-memory layout and floating point data type for the current Keras backend.

### Usage

```
keras_array(x, dtype = NULL)
```

### Arguments

- `x` Object or list of objects to convert
- `dtype` NumPy data type (e.g. float32, float64). If this is unspecified then R doubles will be converted to the default floating point type for the current Keras backend.
Details

Keras does frequent row-oriented access to arrays (for shuffling and drawing batches) so the order of arrays created by this function is always row-oriented ("C" as opposed to "Fortran" ordering, which is the default for R arrays).

If the passed array is already a NumPy array with the desired dtype and "C" order then it is returned unmodified (no additional copies are made).

Value

NumPy array with the specified dtype (or list of NumPy arrays if a list was passed for x).

________________________

| keras_model | Keras Model |
________________________

Description

A model is a directed acyclic graph of layers.

Usage

keras_model(inputs, outputs = NULL, ...)

Arguments

inputs       Input layer
outputs      Output layer
...          Any additional arguments

See Also

Other model functions: compile.keras.engine.training.Model(), evaluate.keras.engine.training.Model(), evaluate_generator(), fit.keras.engine.training.Model(), fit_generator(), get_config(), get_layer(), keras_model_sequential(), multi_gpu_model(), pop_layer(), predict.keras.engine.training.Model(), predict_generator(), predict_on_batch(), predict_proba(), summary.keras.engine.training.Model(), train_on_batch()

Examples

```r
## Not run:
library(keras)

# input layer
inputs <- layer_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')
)

## End(Not run)

---

**keras_model_custom**  
*Create a Keras custom model*

### Description

Create a Keras custom model

### Usage

```r
keras_model_custom(model_fn, name = NULL)
```

### Arguments

- **model_fn**  
  Function that returns an R custom model

- **name**  
  Optional name for model

### Details

For documentation on using custom models, see [https://keras.rstudio.com/articles/custom_models.html](https://keras.rstudio.com/articles/custom_models.html).

### Value

A Keras model
**keras_model_sequential**

*Keras Model composed of a linear stack of layers*

---

**Description**

Keras Model composed of a linear stack of layers

**Usage**

```r
keras_model_sequential(layers = NULL, name = NULL, ...)
```

**Arguments**

- `layers` List of layers to add to the model
- `name` Name of model
- `...` Arguments passed on to `sequential_model_input_layer`
  - `input_shape` an integer vector of dimensions (not including the batch axis), or a `tf$TensorShape` instance (also not including the batch axis).
  - `batch_size` Optional input batch size (integer or NULL).
  - `dtype` Optional datatype of the input. When not provided, the Keras default float type will be used.
  - `input_tensor` Optional tensor to use as layer input. If set, the layer will use the `tf$TypeSpec` of this tensor rather than creating a new placeholder tensor.
  - `sparse` Boolean, whether the placeholder created is meant to be sparse. Default to FALSE.
  - `ragged` Boolean, whether the placeholder created is meant to be ragged. In this case, values of 'NULL' in the 'shape' argument represent ragged dimensions. For more information about RaggedTensors, see this guide. Default to FALSE.
  - `type_spec` A `tf$TypeSpec` object to create `Input` from. This `tf$TypeSpec` represents the entire batch. When provided, all other args except name must be NULL.
  - `input_layer_name` Optional name of the input layer (string).

**Note**

If any arguments are provided to `...`, then the sequential model is initialized with a `InputLayer` instance. If not, then the first layer passed to a Sequential model should have a defined input shape. What that means is that it should have received an `input_shape` or `batch_input_shape` argument, or for some type of layers (recurrent, Dense...) an `input_dim` argument.
See Also


Examples

```r
## Not run:

library(keras)

model <- keras_model_sequential()
model %>%
  layer_dense(units = 32, input_shape = c(784)) %>%
  layer_activation('relu') %>%
  layer_dense(units = 10) %>%
  layer_activation('softmax')

model %>% compile(
  optimizer = 'rmsprop',
  loss = 'categorical_crossentropy',
  metrics = c('accuracy')
)

# alternative way to provide input shape
model <- keras_model_sequential(input_shape = c(784)) %>%
  layer_dense(units = 32) %>%
  layer_activation('relu') %>%
  layer_dense(units = 10) %>%
  layer_activation('softmax')

## End(Not run)

---

**k_abs**

*Element-wise absolute value.*

**Description**

Element-wise absolute value.

**Usage**

`k_abs(x)`

**Arguments**

- `x` Tensor or variable.
**k_all**

**Value**

A tensor.

**Keras Backend**

This function is part of a set of Keras backend functions that enable lower level access to the core operations of the backend tensor engine (e.g. TensorFlow, CNTK, Theano, etc.).

You can see a list of all available backend functions here: https://keras.rstudio.com/articles/backend.html#backend-functions.

---

**k_all**  
*Bitwise reduction (logical AND).*

**Description**

Bitwise reduction (logical AND).

**Usage**

`k_all(x, axis = NULL, keepdims = FALSE)`

**Arguments**

- **x**: Tensor or variable.
- **axis**: Axis along which to perform the reduction (axis indexes are 1-based).
- **keepdims**: Whether the drop or broadcast the reduction axes.

**Value**

A uint8 tensor (0s and 1s).

**Keras Backend**

This function is part of a set of Keras backend functions that enable lower level access to the core operations of the backend tensor engine (e.g. TensorFlow, CNTK, Theano, etc.).

You can see a list of all available backend functions here: https://keras.rstudio.com/articles/backend.html#backend-functions.
k_any

Bitwise reduction (logical OR).

Description

Bitwise reduction (logical OR).

Usage

k_any(x, axis = NULL, keepdims = FALSE)

Arguments

- x: Tensor or variable.
- axis: Axis along which to perform the reduction (axis indexes are 1-based).
- keepdims: whether the drop or broadcast the reduction axes.

Value

A uint8 tensor (0s and 1s).

Keras Backend

This function is part of a set of Keras backend functions that enable lower level access to the core operations of the backend tensor engine (e.g. TensorFlow, CNTK, Theano, etc.).

You can see a list of all available backend functions here: https://keras.rstudio.com/articles/backend.html#backend-functions.

k_arange

Creates a 1D tensor containing a sequence of integers.

Description

The function arguments use the same convention as Theano's arange: if only one argument is provided, it is in fact the "stop" argument. The default type of the returned tensor is 'int32' to match TensorFlow's default.

Usage

k_arange(start, stop = NULL, step = 1, dtype = "int32")
**k_argmax**

**Arguments**
- **start**: Start value.
- **stop**: Stop value.
- **step**: Difference between two successive values.
- **dtype**: Integer dtype to use.

**Value**
An integer tensor.

**Keras Backend**
This function is part of a set of Keras backend functions that enable lower level access to the core operations of the backend tensor engine (e.g. TensorFlow, CNTK, Theano, etc.).
You can see a list of all available backend functions here: [https://keras.rstudio.com/articles/backend.html#backend-functions](https://keras.rstudio.com/articles/backend.html#backend-functions).

---

**k_argmax**

*Returns the index of the maximum value along an axis.*

**Description**
Returns the index of the maximum value along an axis.

**Usage**

```python
k_argmax(x, axis = -1)
```

**Arguments**
- **x**: Tensor or variable.
- **axis**: Axis along which to perform the reduction (axis indexes are 1-based). Pass -1 (the default) to select the last axis.

**Value**
A tensor.

**Keras Backend**
This function is part of a set of Keras backend functions that enable lower level access to the core operations of the backend tensor engine (e.g. TensorFlow, CNTK, Theano, etc.).
You can see a list of all available backend functions here: [https://keras.rstudio.com/articles/backend.html#backend-functions](https://keras.rstudio.com/articles/backend.html#backend-functions).
k_argmin  Returns the index of the minimum value along an axis.

Description

Returns the index of the minimum value along an axis.

Usage

k_argmin(x, axis = -1)

Arguments

x  Tensor or variable.
axis  Axis along which to perform the reduction (axis indexes are 1-based). Pass -1 (the default) to select the last axis.

Value

A tensor.

Keras Backend

This function is part of a set of Keras backend functions that enable lower level access to the core operations of the backend tensor engine (e.g. TensorFlow, CNTK, Theano, etc.).

You can see a list of all available backend functions here: https://keras.rstudio.com/articles/backend.html#backend-functions.

k_backend  Active Keras backend

Description

Active Keras backend

Usage

k_backend()

Value

The name of the backend Keras is currently using.
**k_batch_dot**

**Batchwise dot product.**

Description

`batch_dot` is used to compute dot product of `x` and `y` when `x` and `y` are data in batch, i.e. in a shape of `(batch_size)`. `batch_dot` results in a tensor or variable with less dimensions than the input. If the number of dimensions is reduced to 1, we use `expand_dims` to make sure that `ndim` is at least 2.

Usage

`k_batch_dot(x, y, axes)`

Arguments

- **x**: Keras tensor or variable with 2 more more axes.
- **y**: Keras tensor or variable with 2 or more axes
- **axes**: List of (or single) integer with target dimensions (axis indexes are 1-based). The lengths of `axes[[1]]` and `axes[[2]]` should be the same.

Value

A tensor with shape equal to the concatenation of `x`’s shape (less the dimension that was summed over) and `y`’s shape (less the batch dimension and the dimension that was summed over). If the final rank is 1, we reshape it to `(batch_size, 1)`.

Keras Backend

This function is part of a set of Keras backend functions that enable lower level access to the core operations of the backend tensor engine (e.g. TensorFlow, CNTK, Theano, etc.).

You can see a list of all available backend functions here: https://keras.rstudio.com/articles/backend.html#backend-functions.
**k_batch_flatten**

*Turn a nD tensor into a 2D tensor with same 1st dimension.*

**Description**

In other words, it flattens each data samples of a batch.

**Usage**

```python
k_batch_flatten(x)
```

**Arguments**

- `x` A tensor or variable.

**Value**

A tensor.

**Keras Backend**

This function is part of a set of Keras backend functions that enable lower level access to the core operations of the backend tensor engine (e.g. TensorFlow, CNTK, Theano, etc.).

You can see a list of all available backend functions here: [https://keras.rstudio.com/articles/backend.html#backend-functions](https://keras.rstudio.com/articles/backend.html#backend-functions).

---

**k_batch_get_value**

*Returns the value of more than one tensor variable.*

**Description**

Returns the value of more than one tensor variable.

**Usage**

```python
k_batch_get_value(ops)
```

**Arguments**

- `ops` List of ops to evaluate.

**Value**

A list of arrays.
k_batch_normalization

**Keras Backend**

This function is part of a set of Keras backend functions that enable lower level access to the core operations of the backend tensor engine (e.g. TensorFlow, CNTK, Theano, etc.).

You can see a list of all available backend functions here: [https://keras.rstudio.com/articles/backend.html#backend-functions](https://keras.rstudio.com/articles/backend.html#backend-functions).

**See Also**

  k_batch_set_value()

---

**k_batch_normalization**  Applies batch normalization on x given mean, var, beta and gamma.

**Description**

i.e. returns output <- (x -mean) / (sqrt(var) + epsilon) * gamma + beta

**Usage**

k_batch_normalization(x, mean, var, beta, gamma, axis = -1, epsilon = 0.001)

**Arguments**

- **x**  
  Input tensor or variable.
- **mean**  
  Mean of batch.
- **var**  
  Variance of batch.
- **beta**  
  Tensor with which to center the input.
- **gamma**  
  Tensor by which to scale the input.
- **axis**  
  Axis (axis indexes are 1-based). Pass -1 (the default) to select the last axis.
- **epsilon**  
  Fuzz factor.

**Value**

A tensor.

**Keras Backend**

This function is part of a set of Keras backend functions that enable lower level access to the core operations of the backend tensor engine (e.g. TensorFlow, CNTK, Theano, etc.).

You can see a list of all available backend functions here: [https://keras.rstudio.com/articles/backend.html#backend-functions](https://keras.rstudio.com/articles/backend.html#backend-functions).
**k_batch_set_value**

Sets the values of many tensor variables at once.

**Description**
Sets the values of many tensor variables at once.

**Usage**

```r
k_batch_set_value(lists)
```

**Arguments**

- `lists` a list of lists (tensor, value). value should be an R array.

**Keras Backend**
This function is part of a set of Keras backend functions that enable lower level access to the core operations of the backend tensor engine (e.g. TensorFlow, CNTK, Theano, etc.).
You can see a list of all available backend functions here: [https://keras.rstudio.com/articles/backend.html#backend-functions](https://keras.rstudio.com/articles/backend.html#backend-functions).

**See Also**

`k_batch_get_value()`

---

**k_bias_add**

Adds a bias vector to a tensor.

**Description**
Adds a bias vector to a tensor.

**Usage**

```r
k_bias_add(x, bias, data_format = NULL)
```

**Arguments**

- `x` Tensor or variable.
- `bias` Bias tensor to add.
- `data_format` string, "channels_last" or "channels_first".

**Value**
Output tensor.
**Keras Backend**

This function is part of a set of Keras backend functions that enable lower level access to the core operations of the backend tensor engine (e.g. TensorFlow, CNTK, Theano, etc.).

You can see a list of all available backend functions here: [https://keras.rstudio.com/articles/backend.html#backend-functions](https://keras.rstudio.com/articles/backend.html#backend-functions).

---

**k_binary_crossentropy**  
*Binary crossentropy between an output tensor and a target tensor.*

---

**Description**

Binary crossentropy between an output tensor and a target tensor.

**Usage**

```r
k_binary_crossentropy(target, output, from_logits = FALSE)
```

**Arguments**

- `target`  
  A tensor with the same shape as `output`.

- `output`  
  A tensor.

- `from_logits`  
  Whether `output` is expected to be a logits tensor. By default, we consider that `output` encodes a probability distribution.

**Value**

A tensor.

---

**Keras Backend**

This function is part of a set of Keras backend functions that enable lower level access to the core operations of the backend tensor engine (e.g. TensorFlow, CNTK, Theano, etc.).

You can see a list of all available backend functions here: [https://keras.rstudio.com/articles/backend.html#backend-functions](https://keras.rstudio.com/articles/backend.html#backend-functions).
**k_cast**

*Casts a tensor to a different dtype and returns it.*

**Description**

You can cast a Keras variable but it still returns a Keras tensor.

**Usage**

\[k\_cast(x, \text{dtype})\]

**Arguments**

- \(x\)  
  Keras tensor (or variable).
- \(\text{dtype}\)  
  String, either ('float16', 'float32', or 'float64').

**Value**

Keras tensor with dtype \(\text{dtype}\).

**Keras Backend**

This function is part of a set of Keras backend functions that enable lower level access to the core operations of the backend tensor engine (e.g. TensorFlow, CNTK, Theano, etc.).

You can see a list of all available backend functions here: [https://keras.rstudio.com/articles/backend.html#backend-functions](https://keras.rstudio.com/articles/backend.html#backend-functions).

---

**k_cast_to_floatx**

*Cast an array to the default Keras float type.*

**Description**

Cast an array to the default Keras float type.

**Usage**

\[k\_cast\_to\_floatx(x)\]

**Arguments**

- \(x\)  
  Array.

**Value**

The same array, cast to its new type.
**Keras Backend**

This function is part of a set of Keras backend functions that enable lower level access to the core operations of the backend tensor engine (e.g. TensorFlow, CNTK, Theano, etc.).

You can see a list of all available backend functions here: [https://keras.rstudio.com/articles/backend.html#backend-functions](https://keras.rstudio.com/articles/backend.html#backend-functions).

---

**k_categorical_crossentropy**

*Categorical crossentropy between an output tensor and a target tensor.*

---

**Description**

Categorical crossentropy between an output tensor and a target tensor.

**Usage**

```r
k_categorical_crossentropy(target, output, from_logits = FALSE, axis = -1)
```

**Arguments**

- `target`: A tensor of the same shape as `output`.
- `output`: A tensor resulting from a softmax (unless `from_logits` is `TRUE`, in which case `output` is expected to be the logits).
- `from_logits`: Logical, whether `output` is the result of a softmax, or is a tensor of logits.
- `axis`: Axis (axis indexes are 1-based). Pass `-1` (the default) to select the last axis.

**Value**

Output tensor.

---

**Keras Backend**

This function is part of a set of Keras backend functions that enable lower level access to the core operations of the backend tensor engine (e.g. TensorFlow, CNTK, Theano, etc.).

You can see a list of all available backend functions here: [https://keras.rstudio.com/articles/backend.html#backend-functions](https://keras.rstudio.com/articles/backend.html#backend-functions).
k_clear_session

*Destroys the current TF graph and creates a new one.*

**Description**
Useful to avoid clutter from old models / layers.

**Usage**
k_clear_session()

**Keras Backend**
This function is part of a set of Keras backend functions that enable lower level access to the core operations of the backend tensor engine (e.g. TensorFlow, CNTK, Theano, etc.).
You can see a list of all available backend functions here: [https://keras.rstudio.com/articles/backend.html#backend-functions](https://keras.rstudio.com/articles/backend.html#backend-functions).

---

k_clip

*Element-wise value clipping.*

**Description**
Element-wise value clipping.

**Usage**
k_clip(x, min_value, max_value)

**Arguments**
- **x**: Tensor or variable.
- **min_value**: Float or integer.
- **max_value**: Float or integer.

**Value**
A tensor.

**Keras Backend**
This function is part of a set of Keras backend functions that enable lower level access to the core operations of the backend tensor engine (e.g. TensorFlow, CNTK, Theano, etc.).
You can see a list of all available backend functions here: [https://keras.rstudio.com/articles/backend.html#backend-functions](https://keras.rstudio.com/articles/backend.html#backend-functions).
**k_concatenate**

*Concatenates a list of tensors alongside the specified axis.*

**Description**

Concatenates a list of tensors alongside the specified axis.

**Usage**

```r
k_concatenate(tensors, axis = -1)
```

**Arguments**

- `tensors` : list of tensors to concatenate.
- `axis` : concatenation axis (axis indexes are 1-based). Pass -1 (the default) to select the last axis.

**Value**

A tensor.

**Keras Backend**

This function is part of a set of Keras backend functions that enable lower level access to the core operations of the backend tensor engine (e.g. TensorFlow, CNTK, Theano, etc.).

You can see a list of all available backend functions here: [https://keras.rstudio.com/articles/backend.html#backend-functions](https://keras.rstudio.com/articles/backend.html#backend-functions).

---

**k_constant**

*Creates a constant tensor.*

**Description**

Creates a constant tensor.

**Usage**

```r
k_constant(value, dtype = NULL, shape = NULL, name = NULL)
```

**Arguments**

- `value` : A constant value.
- `dtype` : The type of the elements of the resulting tensor.
- `shape` : Optional dimensions of resulting tensor.
- `name` : Optional name for the tensor.
**k_conv1d**

**Value**
A Constant Tensor.

**Keras Backend**
This function is part of a set of Keras backend functions that enable lower level access to the core operations of the backend tensor engine (e.g. TensorFlow, CNTK, Theano, etc.).
You can see a list of all available backend functions here: https://keras.rstudio.com/articles/backend.html#backend-functions.

---

**Description**
1D convolution.

**Usage**

```r
k_conv1d(
  x, 
  kernel, 
  strides = 1, 
  padding = "valid", 
  data_format = NULL, 
  dilation_rate = 1 
)
```

**Arguments**

- `x` Tensor or variable.
- `kernel` kernel tensor.
- `strides` stride integer.
- `padding` string, "same", "causal" or "valid".
- `data_format` string, "channels_last" or "channels_first".
- `dilation_rate` integer dilate rate.

**Value**
A tensor, result of 1D convolution.

**Keras Backend**
This function is part of a set of Keras backend functions that enable lower level access to the core operations of the backend tensor engine (e.g. TensorFlow, CNTK, Theano, etc.).
You can see a list of all available backend functions here: https://keras.rstudio.com/articles/backend.html#backend-functions.
**k_conv2d**

2D convolution.

**Description**

2D convolution.

**Usage**

```r
k_conv2d(
  x,
  kernel,
  strides = c(1, 1),
  padding = "valid",
  data_format = NULL,
  dilation_rate = c(1, 1)
)
```

**Arguments**

- **x**  
  Tensor or variable.
- **kernel**  
  kernel tensor.
- **strides**  
  strides
- **padding**  
  string, "same" or "valid".
- **data_format**  
  string, "channels_last" or "channels_first". Whether to use Theano or TensorFlow/CNTK data format for inputs/kernels/outputs.
- **dilation_rate**  
  vector of 2 integers.

**Value**

A tensor, result of 2D convolution.

**Keras Backend**

This function is part of a set of Keras backend functions that enable lower level access to the core operations of the backend tensor engine (e.g. TensorFlow, CNTK, Theano, etc.).

You can see a list of all available backend functions here: [https://keras.rstudio.com/articles/backend.html#backend-functions](https://keras.rstudio.com/articles/backend.html#backend-functions).
**k_conv2d_transpose**  
2D deconvolution (i.e. transposed convolution).

**Description**

2D deconvolution (i.e. transposed convolution).

**Usage**

```r
k_conv2d_transpose(
  x,
  kernel,
  output_shape,
  strides = c(1, 1),
  padding = "valid",
  data_format = "channels_last"
)
```

**Arguments**

- `x` Tensor or variable.
- `kernel` kernel tensor.
- `output_shape` 1D int tensor for the output shape.
- `strides` strides list.
- `padding` string, "same" or "valid".
- `data_format` string, "channels_last" or "channels_first". Whether to use Theano or TensorFlow/CNTK data format for inputs/kernels/outputs.

**Value**

A tensor, result of transposed 2D convolution.

**Keras Backend**

This function is part of a set of Keras backend functions that enable lower level access to the core operations of the backend tensor engine (e.g. TensorFlow, CNTK, Theano, etc.).

You can see a list of all available backend functions here: [https://keras.rstudio.com/articles/backend.html#backend-functions](https://keras.rstudio.com/articles/backend.html#backend-functions).
k_conv3d 3D convolution.

Description

3D convolution.

Usage

```r
k_conv3d(
  x,
  kernel,
  strides = c(1, 1, 1),
  padding = "valid",
  data_format = NULL,
  dilation_rate = c(1, 1, 1)
)
```

Arguments

- `x` Tensor or variable.
- `kernel` kernel tensor.
- `strides` strides
- `padding` string, "same" or "valid".
- `data_format` string, "channels_last" or "channels_first". Whether to use Theano or TensorFlow/CNTK data format for inputs/kernels/outputs.
- `dilation_rate` list of 3 integers.

Value

A tensor, result of 3D convolution.

Keras Backend

This function is part of a set of Keras backend functions that enable lower level access to the core operations of the backend tensor engine (e.g. TensorFlow, CNTK, Theano, etc.).

You can see a list of all available backend functions here: https://keras.rstudio.com/articles/backend.html#backend-functions.
**k_conv3d_transpose**  
*3D deconvolution (i.e. transposed convolution).*

### Description

3D deconvolution (i.e. transposed convolution).

### Usage

```r
k_conv3d_transpose(  
x,  
kernel,  
output_shape,  
strides = c(1, 1, 1),  
padding = "valid",  
data_format = NULL  
)
```

### Arguments

- `x` input tensor.
- `kernel` kernel tensor.
- `output_shape` 1D int tensor for the output shape.
- `strides` strides
- `padding` string, "same" or "valid".
- `data_format` string, "channels_last" or "channels_first". Whether to use Theano or TensorFlow/CNTK data format for inputs/kernels/outputs.

### Value

A tensor, result of transposed 3D convolution.

### Keras Backend

This function is part of a set of Keras backend functions that enable lower level access to the core operations of the backend tensor engine (e.g. TensorFlow, CNTK, Theano, etc.).

You can see a list of all available backend functions here: [https://keras.rstudio.com/articles/backend.html#backend-functions](https://keras.rstudio.com/articles/backend.html#backend-functions).
**k\_cos**

*Computes cos of x element-wise.*

---

**Description**

Computes cos of x element-wise.

**Usage**

\[k\_\text{cos}(x)\]

**Arguments**

- \(x\) Tensor or variable.

**Value**

A tensor.

**Keras Backend**

This function is part of a set of Keras backend functions that enable lower level access to the core operations of the backend tensor engine (e.g. TensorFlow, CNTK, Theano, etc.).

You can see a list of all available backend functions here: [https://keras.rstudio.com/articles/backend.html#backend-functions](https://keras.rstudio.com/articles/backend.html#backend-functions).

---

**k\_count\_params**

*Returns the static number of elements in a Keras variable or tensor.*

---

**Description**

Returns the static number of elements in a Keras variable or tensor.

**Usage**

\[k\_\text{count\_params}(x)\]

**Arguments**

- \(x\) Keras variable or tensor.

**Value**

Integer, the number of elements in \(x\), i.e., the product of the array’s static dimensions.
**Keras Backend**

This function is part of a set of Keras backend functions that enable lower level access to the core operations of the backend tensor engine (e.g. TensorFlow, CNTK, Theano, etc.).

You can see a list of all available backend functions here: https://keras.rstudio.com/articles/backend.html#backend-functions.

---

**k_ctc_batch_cost**

*Runs CTC loss algorithm on each batch element.*

**Description**

Runs CTC loss algorithm on each batch element.

**Usage**

\[
k_{ctc\_batch\_cost}(y_{true}, y_{pred}, input\_length, label\_length)
\]

**Arguments**

- **y_true**: tensor (samples, max_string_length) containing the truth labels.
- **y_pred**: tensor (samples, time_steps, num_categories) containing the prediction, or output of the softmax.
- **input_length**: tensor (samples, 1) containing the sequence length for each batch item in \(y_{pred}\).
- **label_length**: tensor (samples, 1) containing the sequence length for each batch item in \(y_{true}\).

**Value**

Tensor with shape (samples,1) containing the CTC loss of each element.

---

**Keras Backend**

This function is part of a set of Keras backend functions that enable lower level access to the core operations of the backend tensor engine (e.g. TensorFlow, CNTK, Theano, etc.).

You can see a list of all available backend functions here: https://keras.rstudio.com/articles/backend.html#backend-functions.
**k_ctc_decode**

Decodes the output of a softmax.

**Description**

Can use either greedy search (also known as best path) or a constrained dictionary search.

**Usage**

```r
k_ctc_decode(
  y_pred,
  input_length,
  greedy = TRUE,
  beam_width = 100L,
  top_paths = 1
)
```

**Arguments**

- **y_pred**: tensor (samples, time_steps, num_categories) containing the prediction, or output of the softmax.
- **input_length**: tensor (samples, ) containing the sequence length for each batch item in y_pred.
- **greedy**: perform much faster best-path search if TRUE. This does not use a dictionary.
- **beam_width**: if greedy is FALSE: a beam search decoder will be used with a beam of this width.
- **top_paths**: if greedy is FALSE, how many of the most probable paths will be returned.

**Value**

If greedy is TRUE, returns a list of one element that contains the decoded sequence. If FALSE, returns the top_paths most probable decoded sequences. Important: blank labels are returned as -1. Tensor (top_paths) that contains the log probability of each decoded sequence.

**Keras Backend**

This function is part of a set of Keras backend functions that enable lower level access to the core operations of the backend tensor engine (e.g. TensorFlow, CNTK, Theano, etc.).

You can see a list of all available backend functions here: [https://keras.rstudio.com/articles/backend.html#backend-functions](https://keras.rstudio.com/articles/backend.html#backend-functions).
**k_ctc_label_dense_to_sparse**

*Converts CTC labels from dense to sparse.*

**Description**

Converts CTC labels from dense to sparse.

**Usage**

```
k_ctc_label_dense_to_sparse(labels, label_lengths)
```

**Arguments**

- **labels**: dense CTC labels.
- **label_lengths**: length of the labels.

**Value**

A sparse tensor representation of the labels.

**Keras Backend**

This function is part of a set of Keras backend functions that enable lower level access to the core operations of the backend tensor engine (e.g. TensorFlow, CNTK, Theano, etc.).

You can see a list of all available backend functions here: [https://keras.rstudio.com/articles/backend.html#backend-functions](https://keras.rstudio.com/articles/backend.html#backend-functions).

---

**k_cumprod**

*Cumulative product of the values in a tensor, alongside the specified axis.*

**Description**

Cumulative product of the values in a tensor, alongside the specified axis.

**Usage**

```
k_cumprod(x, axis = 1)
```

**Arguments**

- **x**: A tensor or variable.
- **axis**: An integer, the axis to compute the product (axis indexes are 1-based).
**Value**

A tensor of the cumulative product of values of `x` along `axis`.

**Keras Backend**

This function is part of a set of Keras backend functions that enable lower level access to the core operations of the backend tensor engine (e.g. TensorFlow, CNTK, Theano, etc.).

You can see a list of all available backend functions here: [https://keras.rstudio.com/articles/backend.html#backend-functions](https://keras.rstudio.com/articles/backend.html#backend-functions).

---

| `k_cumsum` | Cumulative sum of the values in a tensor, alongside the specified axis. |

**Description**

Cumulative sum of the values in a tensor, alongside the specified axis.

**Usage**

`k_cumsum(x, axis = 1)`

**Arguments**

- `x` A tensor or variable.
- `axis` An integer, the axis to compute the sum (axis indexes are 1-based).

**Value**

A tensor of the cumulative sum of values of `x` along `axis`.

**Keras Backend**

This function is part of a set of Keras backend functions that enable lower level access to the core operations of the backend tensor engine (e.g. TensorFlow, CNTK, Theano, etc.).

You can see a list of all available backend functions here: [https://keras.rstudio.com/articles/backend.html#backend-functions](https://keras.rstudio.com/articles/backend.html#backend-functions).
**k_depthwise_conv2d**  
*Depthwise 2D convolution with separable filters.*

**Description**

Depthwise 2D convolution with separable filters.

**Usage**

```r
k_depthwise_conv2d(
  x,
  depthwise_kernel,
  strides = c(1, 1),
  padding = "valid",
  data_format = NULL,
  dilation_rate = c(1, 1)
)
```

**Arguments**

- **x**  
  input tensor
- **depthwise_kernel**  
  convolution kernel for the depthwise convolution.
- **strides**  
  strides (length 2).
- **padding**  
  string, "same" or "valid".
- **data_format**  
  string, "channels_last" or "channels_first".
- **dilation_rate**  
  vector of integers, dilation rates for the separable convolution.

**Value**

Output tensor.

**Keras Backend**

This function is part of a set of Keras backend functions that enable lower level access to the core operations of the backend tensor engine (e.g. TensorFlow, CNTK, Theano, etc.).

You can see a list of all available backend functions here: [https://keras.rstudio.com/articles/backend.html#backend-functions](https://keras.rstudio.com/articles/backend.html#backend-functions).
**k_dot**

*Multiplies 2 tensors (and/or variables) and returns a tensor.*

**Description**

When attempting to multiply a nD tensor with a nD tensor, it reproduces the Theano behavior. *(e.g. (2, 3) * (4, 3, 5) -> (2, 4, 5))*

**Usage**

```python
k_dot(x, y)
```

**Arguments**

- **x**: Tensor or variable.
- **y**: Tensor or variable.

**Value**

A tensor, dot product of `x` and `y`.

**Keras Backend**

This function is part of a set of Keras backend functions that enable lower level access to the core operations of the backend tensor engine (e.g. TensorFlow, CNTK, Theano, etc.).

You can see a list of all available backend functions here: [https://keras.rstudio.com/articles/backend.html#backend-functions](https://keras.rstudio.com/articles/backend.html#backend-functions).

**k_dropout**

*Sets entries in x to zero at random, while scaling the entire tensor.*

**Description**

Sets entries in `x` to zero at random, while scaling the entire tensor.

**Usage**

```python
k_dropout(x, level, noise_shape = NULL, seed = NULL)
```

**Arguments**

- **x**: Tensor or variable.
- **level**: Fraction of the entries in the tensor that will be set to 0.
- **noise_shape**: Shape for randomly generated keep/drop flags, must be broadcastable to the shape of `x`.
- **seed**: Random seed to ensure determinism.
**k_dtype**

Value

A tensor.

Keras Backend

This function is part of a set of Keras backend functions that enable lower level access to the core operations of the backend tensor engine (e.g. TensorFlow, CNTK, Theano, etc.).

You can see a list of all available backend functions here: [https://keras.rstudio.com/articles/backend.html#backend-functions](https://keras.rstudio.com/articles/backend.html#backend-functions).

---

**k_dtype**

*Returns the dtype of a Keras tensor or variable, as a string.*

---

Description

Returns the dtype of a Keras tensor or variable, as a string.

Usage

\[
k_{\text{dtype}}(x)
\]

Arguments

- **x**: Tensor or variable.

Value

String, dtype of \(x\).

Keras Backend

This function is part of a set of Keras backend functions that enable lower level access to the core operations of the backend tensor engine (e.g. TensorFlow, CNTK, Theano, etc.).

You can see a list of all available backend functions here: [https://keras.rstudio.com/articles/backend.html#backend-functions](https://keras.rstudio.com/articles/backend.html#backend-functions).
**k_elu**  
*Exponential linear unit.*

**Description**  
Exponential linear unit.

**Usage**  
k_elu(x, alpha = 1)

**Arguments**
- **x**  
  A tensor or variable to compute the activation function for.
- **alpha**  
  A scalar, slope of negative section.

**Value**  
A tensor.

**Keras Backend**  
This function is part of a set of Keras backend functions that enable lower level access to the core operations of the backend tensor engine (e.g. TensorFlow, CNTK, Theano, etc.).

You can see a list of all available backend functions here: [https://keras.rstudio.com/articles/backend.html#backend-functions](https://keras.rstudio.com/articles/backend.html#backend-functions).

---

**k_epsilon**  
*Fuzz factor used in numeric expressions.*

**Description**  
Fuzz factor used in numeric expressions.

**Usage**  
k_epsilon()  
k_set_epsilon(e)

**Arguments**
- **e**  
  float. New value of epsilon.
Keras Backend

This function is part of a set of Keras backend functions that enable lower level access to the core operations of the backend tensor engine (e.g. TensorFlow, CNTK, Theano, etc.).

You can see a list of all available backend functions here: https://keras.rstudio.com/articles/backend.html#backend-functions.

k_equal

Element-wise equality between two tensors.

Description

Element-wise equality between two tensors.

Usage

k_equal(x, y)

Arguments

x Tensor or variable.

y Tensor or variable.

Value

A bool tensor.

Keras Backend

This function is part of a set of Keras backend functions that enable lower level access to the core operations of the backend tensor engine (e.g. TensorFlow, CNTK, Theano, etc.).

You can see a list of all available backend functions here: https://keras.rstudio.com/articles/backend.html#backend-functions.

k_eval

Evaluates the value of a variable.

Description

Evaluates the value of a variable.

Usage

k_eval(x)
**k_exp**

**Arguments**

- **x**: A variable.

**Value**

An R array.

**Keras Backend**

This function is part of a set of Keras backend functions that enable lower level access to the core operations of the backend tensor engine (e.g. TensorFlow, CNTK, Theano, etc.).

You can see a list of all available backend functions here: [https://keras.rstudio.com/articles/backend.html#backend-functions](https://keras.rstudio.com/articles/backend.html#backend-functions).

---

**k_exp**

*Element-wise exponential.*

**Description**

Element-wise exponential.

**Usage**

```python
k_exp(x)
```

**Arguments**

- **x**: Tensor or variable.

**Value**

A tensor.

**Keras Backend**

This function is part of a set of Keras backend functions that enable lower level access to the core operations of the backend tensor engine (e.g. TensorFlow, CNTK, Theano, etc.).

You can see a list of all available backend functions here: [https://keras.rstudio.com/articles/backend.html#backend-functions](https://keras.rstudio.com/articles/backend.html#backend-functions).
**k_expand_dims**  
*Adds a 1-sized dimension at index axis.*

**Description**  
Adds a 1-sized dimension at index axis.

**Usage**  
```
k_expand_dims(x, axis = -1)
```

**Arguments**

- `x`  
  A tensor or variable.
- `axis`  
  Position where to add a new axis (axis indexes are 1-based). Pass -1 (the default) to select the last axis.

**Value**  
A tensor with expanded dimensions.

**Keras Backend**

This function is part of a set of Keras backend functions that enable lower level access to the core operations of the backend tensor engine (e.g. TensorFlow, CNTK, Theano, etc.).

You can see a list of all available backend functions here: [https://keras.rstudio.com/articles/backend.html#backend-functions](https://keras.rstudio.com/articles/backend.html#backend-functions).

---

**k_eye**  
*Instantiate an identity matrix and returns it.*

**Description**  
Instantiate an identity matrix and returns it.

**Usage**  
```
k_eye(size, dtype = NULL, name = NULL)
```

**Arguments**

- `size`  
  Integer, number of rows/columns.
- `dtype`  
  String, data type of returned Keras variable.
- `name`  
  String, name of returned Keras variable.
**k_flatten**

**Value**

A Keras variable, an identity matrix.

**Keras Backend**

This function is part of a set of Keras backend functions that enable lower level access to the core operations of the backend tensor engine (e.g. TensorFlow, CNTK, Theano, etc.).

You can see a list of all available backend functions here: https://keras.rstudio.com/articles/backend.html#backend-functions.

---

**k_flatten**  

*Flatten a tensor.*

**Description**

Flatten a tensor.

**Usage**

\[ k\_flatten(x) \]

**Arguments**

\[ x \]

A tensor or variable.

**Value**

A tensor, reshaped into 1-D

**Keras Backend**

This function is part of a set of Keras backend functions that enable lower level access to the core operations of the backend tensor engine (e.g. TensorFlow, CNTK, Theano, etc.).

You can see a list of all available backend functions here: https://keras.rstudio.com/articles/backend.html#backend-functions.
**k_floatx**

_DEFAULT FLOAT TYPE_

**Description**

Default float type

**Usage**

k_floatx()

k_set_floatx(floatx)

**Arguments**

floatx String, 'float16', 'float32', or 'float64'.

**Keras Backend**

This function is part of a set of Keras backend functions that enable lower level access to the core operations of the backend tensor engine (e.g. TensorFlow, CNTK, Theano, etc.).

You can see a list of all available backend functions here: https://keras.rstudio.com/articles/backend.html#backend-functions.

---

**k_foldl**

Reduce elems using fn to combine them from left to right.

**Description**

Reduce elems using fn to combine them from left to right.

**Usage**

k_foldl(fn, elems, initializer = NULL, name = NULL)

**Arguments**

fn Function that will be called upon each element in elems and an accumulator

elems tensor

initializer The first value used (first element of elems in case of ‘NULL’)

name A string name for the foldl node in the graph

**Value**

Tensor with same type and shape as initializer.
**Keras Backend**

This function is part of a set of Keras backend functions that enable lower level access to the core operations of the backend tensor engine (e.g. TensorFlow, CNTK, Theano, etc.).

You can see a list of all available backend functions here: [https://keras.rstudio.com/articles/backend.html#backend-functions](https://keras.rstudio.com/articles/backend.html#backend-functions).

---

```r
k_foldr
```

*Reduce elems using fn to combine them from right to left.*

**Description**

Reduce elems using fn to combine them from right to left.

**Usage**

```r
k_foldr(fn, elems, initializer = NULL, name = NULL)
```

**Arguments**

- `fn` Function that will be called upon each element in elems and an accumulator
- `elems` tensor
- `initializer` The first value used (last element of `elems` in case of NULL)
- `name` A string name for the foldr node in the graph

**Value**

Tensor with same type and shape as `initializer`.

**Keras Backend**

This function is part of a set of Keras backend functions that enable lower level access to the core operations of the backend tensor engine (e.g. TensorFlow, CNTK, Theano, etc.).

You can see a list of all available backend functions here: [https://keras.rstudio.com/articles/backend.html#backend-functions](https://keras.rstudio.com/articles/backend.html#backend-functions).
**k_function**

*Instantiates a Keras function*

**Description**

Instantiates a Keras function

**Usage**

```
k_function(inputs, outputs, updates = NULL, ...)
```

**Arguments**

- `inputs`: List of placeholder tensors.
- `outputs`: List of output tensors.
- `updates`: List of update ops.
- `...`: Named arguments passed to `tf$Session$run`.

**Value**

Output values as R arrays.

**Keras Backend**

This function is part of a set of Keras backend functions that enable lower level access to the core operations of the backend tensor engine (e.g. TensorFlow, CNTK, Theano, etc.).

You can see a list of all available backend functions here: [https://keras.rstudio.com/articles/backend.html#backend-functions](https://keras.rstudio.com/articles/backend.html#backend-functions).

---

**k_gather**

*Retrieves the elements of indices indices in the tensor reference.*

**Description**

Retrieves the elements of indices indices in the tensor reference.

**Usage**

```
k_gather(reference, indices)
```

**Arguments**

- `indices`: Indices. Dimension indices are 1-based. Note however that if you pass a tensor for `indices` they will be passed as-is, in which case indices will be 0 based because no normalizing of R 1-based axes to Python 0-based axes is performed.
Value

A tensor of same type as reference.

Keras Backend

This function is part of a set of Keras backend functions that enable lower level access to the core operations of the backend tensor engine (e.g. TensorFlow, CNTK, Theano, etc.).

You can see a list of all available backend functions here: https://keras.rstudio.com/articles/backend.html#backend-functions.

---

**k_get_session**

*TF session to be used by the backend.*

Description

If a default TensorFlow session is available, we will return it. Else, we will return the global Keras session. If no global Keras session exists at this point: we will create a new global session. Note that you can manually set the global session via `k_set_session()`.

Usage

```
k_get_session()
k_set_session(session)
```

Arguments

- `session` A TensorFlow Session.

Value

A TensorFlow session

Keras Backend

This function is part of a set of Keras backend functions that enable lower level access to the core operations of the backend tensor engine (e.g. TensorFlow, CNTK, Theano, etc.).

You can see a list of all available backend functions here: https://keras.rstudio.com/articles/backend.html#backend-functions.
**k_get_uid**

*Get the uid for the default graph.*

**Description**
Get the uid for the default graph.

**Usage**
```r
k_get_uid(prefix = "")
```

**Arguments**
- `prefix`: An optional prefix of the graph.

**Value**
A unique identifier for the graph.

**Keras Backend**
This function is part of a set of Keras backend functions that enable lower level access to the core operations of the backend tensor engine (e.g. TensorFlow, CNTK, Theano, etc.).

You can see a list of all available backend functions here: [https://keras.rstudio.com/articles/backend.html#backend-functions](https://keras.rstudio.com/articles/backend.html#backend-functions).

---

**k_get_value**

*Returns the value of a variable.*

**Description**
Returns the value of a variable.

**Usage**
```r
k_get_value(x)
```

**Arguments**
- `x`: input variable.

**Value**
An R array.
**k_get_variable_shape**

Returns the shape of a variable.

**Description**

Returns the shape of a variable.

**Usage**

```
 k_get_variable_shape(x)
```

**Arguments**

- `x`: A variable.

**Value**

A vector of integers.

---

**k_gradients**

Returns the gradients of variables w.r.t. loss.

**Description**

Returns the gradients of variables w.r.t. loss.

**Usage**

```
 k_gradients(loss, variables)
```
Arguments

- **loss**: Scalar tensor to minimize.
- **variables**: List of variables.

Value

A gradients tensor.

Keras Backend

This function is part of a set of Keras backend functions that enable lower level access to the core operations of the backend tensor engine (e.g. TensorFlow, CNTK, Theano, etc.).

You can see a list of all available backend functions here: https://keras.rstudio.com/articles/backend.html#backend-functions.

---

**k_greater**

<table>
<thead>
<tr>
<th>k_greater</th>
<th>Element-wise truth value of ((x &gt; y)).</th>
</tr>
</thead>
</table>

Description

Element-wise truth value of \((x > y)\).

Usage

```python
k_greater(x, y)
```

Arguments

- **x**: Tensor or variable.
- **y**: Tensor or variable.

Value

A bool tensor.

Keras Backend

This function is part of a set of Keras backend functions that enable lower level access to the core operations of the backend tensor engine (e.g. TensorFlow, CNTK, Theano, etc.).

You can see a list of all available backend functions here: https://keras.rstudio.com/articles/backend.html#backend-functions.
**k_greater_equal**  
*Element-wise truth value of (x >= y).*

---

**Description**
Element-wise truth value of (x >= y).

**Usage**
$k_{\text{greater\_equal}}(x, y)$

**Arguments**
- **x**  
  Tensor or variable.
- **y**  
  Tensor or variable.

**Value**  
A bool tensor.

**Keras Backend**
This function is part of a set of Keras backend functions that enable lower level access to the core operations of the backend tensor engine (e.g. TensorFlow, CNTK, Theano, etc.).
You can see a list of all available backend functions here: [https://keras.rstudio.com/articles/backend.html#backend-functions](https://keras.rstudio.com/articles/backend.html#backend-functions).

---

**k_hard_sigmoid**  
*Segment-wise linear approximation of sigmoid.*

---

**Description**
Faster than sigmoid. Returns 0 if x < -2.5, 1 if x > 2.5. In -2.5 <= x <= 2.5, returns 0.2 * x + 0.5.

**Usage**
$k_{\text{hard\_sigmoid}}(x)$

**Arguments**
- **x**  
  A tensor or variable.

**Value**  
A tensor.
Keras Backend

This function is part of a set of Keras backend functions that enable lower level access to the core operations of the backend tensor engine (e.g. TensorFlow, CNTK, Theano, etc.).

You can see a list of all available backend functions here: https://keras.rstudio.com/articles/backend.html#backend-functions.

---

**k_identity**

Returns a tensor with the same content as the input tensor.

**Description**

Returns a tensor with the same content as the input tensor.

**Usage**

```r
k_identity(x, name = NULL)
```

**Arguments**

- **x**: The input tensor.
- **name**: String, name for the variable to create.

**Value**

A tensor of the same shape, type and content.

---

Keras Backend

This function is part of a set of Keras backend functions that enable lower level access to the core operations of the backend tensor engine (e.g. TensorFlow, CNTK, Theano, etc.).

You can see a list of all available backend functions here: https://keras.rstudio.com/articles/backend.html#backend-functions.

---

**k_image_data_format**

Default image data format convention ('channels_first' or 'channels_last').

**Description**

Default image data format convention ('channels_first' or 'channels_last').

**Usage**

```r
k_image_data_format()
```

```r
k_set_image_data_format(data_format)
```
Arguments

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

Keras Backend

This function is part of a set of Keras backend functions that enable lower level access to the core operations of the backend tensor engine (e.g. TensorFlow, CNTK, Theano, etc.).

You can see a list of all available backend functions here: https://keras.rstudio.com/articles/backend.html#backend-functions.

---

**k_int_shape**

*Returns the shape of tensor or variable as a list of int or NULL entries.*

---

Description

Returns the shape of tensor or variable as a list of int or NULL entries.

Usage

```python
k_int_shape(x)
```

Arguments

- `x` Tensor or variable.

Value

A list of integers (or NULL entries).

Keras Backend

This function is part of a set of Keras backend functions that enable lower level access to the core operations of the backend tensor engine (e.g. TensorFlow, CNTK, Theano, etc.).

You can see a list of all available backend functions here: https://keras.rstudio.com/articles/backend.html#backend-functions.
### k_in_test_phase

Selects \( x \) in test phase, and \( alt \) otherwise.

#### Description

Note that \( alt \) should have the *same shape* as \( x \).

#### Usage

```r
k_in_test_phase(x, alt, training = NULL)
```

#### Arguments

- **x**: What to return in test phase (tensor or function that returns a tensor).
- **alt**: What to return otherwise (tensor or function that returns a tensor).
- **training**: Optional scalar tensor (or R logical or integer) specifying the learning phase.

#### Value

Either \( x \) or \( alt \) based on \( k_{\text{learning\_phase}}() \).

#### Keras Backend

This function is part of a set of Keras backend functions that enable lower level access to the core operations of the backend tensor engine (e.g. TensorFlow, CNTK, Theano, etc.).

You can see a list of all available backend functions here: [https://keras.rstudio.com/articles/backend.html#backend-functions](https://keras.rstudio.com/articles/backend.html#backend-functions).

### k_in_top_k

Returns whether the targets are in the top \( k \) predictions.

#### Description

Returns whether the targets are in the top \( k \) predictions.

#### Usage

```r
k_in_top_k(predictions, targets, k)
```

#### Arguments

- **predictions**: A tensor of shape (batch_size, classes) and type float32.
- **targets**: A 1D tensor of length batch_size and type int32 or int64.
- **k**: An int, number of top elements to consider.
Value

A 1D tensor of length batch_size and type bool. output[[i]] is TRUE if predictions[i, targets[[i]]
 is within top-k values of predictions[[i]].

Keras Backend

This function is part of a set of Keras backend functions that enable lower level access to the core
operations of the backend tensor engine (e.g. TensorFlow, CNTK, Theano, etc.).

You can see a list of all available backend functions here: https://keras.rstudio.com/articles/
backend.html#backend-functions.

k_in_train_phase

Selects x in train phase, and alt otherwise.

Description

Note that alt should have the same shape as x.

Usage

k_in_train_phase(x, alt, training = NULL)

Arguments

x What to return in train phase (tensor or function that returns a tensor).
alt What to return otherwise (tensor or function that returns a tensor).
training Optional scalar tensor (or R logical or integer) specifying the learning phase.

Value

Either x or alt based on the training flag. the training flag defaults to k_learning_phase().

Keras Backend

This function is part of a set of Keras backend functions that enable lower level access to the core
operations of the backend tensor engine (e.g. TensorFlow, CNTK, Theano, etc.).

You can see a list of all available backend functions here: https://keras.rstudio.com/articles/
backend.html#backend-functions.
**k_is_keras_tensor**

Returns whether `x` is a Keras tensor.

**Description**

A "Keras tensor" is a tensor that was returned by a Keras layer.

**Usage**

```
k_is_keras_tensor(x)
```

**Arguments**

`x`  
A candidate tensor.

**Value**

A logical: Whether the argument is a Keras tensor.

**Keras Backend**

This function is part of a set of Keras backend functions that enable lower level access to the core operations of the backend tensor engine (e.g. TensorFlow, CNTK, Theano, etc.).

You can see a list of all available backend functions here: https://keras.rstudio.com/articles/backend.html#backend-functions.

**k_is_placeholder**

Returns whether `x` is a placeholder.

**Description**

Returns whether `x` is a placeholder.

**Usage**

```
k_is_placeholder(x)
```

**Arguments**

`x`  
A candidate placeholder.

**Value**

A logical
Keras Backend

This function is part of a set of Keras backend functions that enable lower level access to the core operations of the backend tensor engine (e.g. TensorFlow, CNTK, Theano, etc.).

You can see a list of all available backend functions here: https://keras.rstudio.com/articles/backend.html#backend-functions.

---

**k_is_sparse**

<table>
<thead>
<tr>
<th>Description</th>
<th>Returns whether a tensor is a sparse tensor.</th>
</tr>
</thead>
</table>

**Usage**

```python
k_is_sparse(tensor)
```

**Arguments**

- `tensor`: A tensor instance.

**Value**

A logical

---

Keras Backend

This function is part of a set of Keras backend functions that enable lower level access to the core operations of the backend tensor engine (e.g. TensorFlow, CNTK, Theano, etc.).

You can see a list of all available backend functions here: https://keras.rstudio.com/articles/backend.html#backend-functions.

---

**k_is_tensor**

<table>
<thead>
<tr>
<th>Description</th>
<th>Returns whether x is a symbolic tensor.</th>
</tr>
</thead>
</table>

**Usage**

```python
k_is_tensor(x)
```

**Arguments**

- `x`: A candidate tensor.
**Value**

A logical: Whether the argument is a symbolic tensor.

**Keras Backend**

This function is part of a set of Keras backend functions that enable lower level access to the core operations of the backend tensor engine (e.g. TensorFlow, CNTK, Theano, etc.).

You can see a list of all available backend functions here: https://keras.rstudio.com/articles/backend.html#backend-functions.

---

**k_l2_normalize**

Normalizes a tensor wrt the L2 norm alongside the specified axis.

**Description**

Normalizes a tensor wrt the L2 norm alongside the specified axis.

**Usage**

```
k_l2_normalize(x, axis = NULL)
```

**Arguments**

- **x**: Tensor or variable.
- **axis**: Axis along which to perform normalization (axis indexes are 1-based)

**Value**

A tensor.

**Keras Backend**

This function is part of a set of Keras backend functions that enable lower level access to the core operations of the backend tensor engine (e.g. TensorFlow, CNTK, Theano, etc.).

You can see a list of all available backend functions here: https://keras.rstudio.com/articles/backend.html#backend-functions.
k_learning_phase

Returns the learning phase flag.

Description

The learning phase flag is a bool tensor (0 = test, 1 = train) to be passed as input to any Keras function that uses a different behavior at train time and test time.

Usage

k_learning_phase()

Value

Learning phase (scalar integer tensor or R integer).

Keras Backend

This function is part of a set of Keras backend functions that enable lower level access to the core operations of the backend tensor engine (e.g. TensorFlow, CNTK, Theano, etc.).

You can see a list of all available backend functions here: https://keras.rstudio.com/articles/backend.html#backend-functions.

k_less

Element-wise truth value of (x < y).

Description

Element-wise truth value of (x < y).

Usage

k_less(x, y)

Arguments

x Tensor or variable.

y Tensor or variable.

Value

A bool tensor.
Keras Backend

This function is part of a set of Keras backend functions that enable lower level access to the core operations of the backend tensor engine (e.g. TensorFlow, CNTK, Theano, etc.).

You can see a list of all available backend functions here: https://keras.rstudio.com/articles/backend.html#backend-functions.

---

**k_less_equal**

Element-wise truth value of \((x \leq y)\).

Description

Element-wise truth value of \((x \leq y)\).

Usage

```
k_less_equal(x, y)
```

Arguments

- `x`: Tensor or variable.
- `y`: Tensor or variable.

Value

A bool tensor.

---

Keras Backend

This function is part of a set of Keras backend functions that enable lower level access to the core operations of the backend tensor engine (e.g. TensorFlow, CNTK, Theano, etc.).

You can see a list of all available backend functions here: https://keras.rstudio.com/articles/backend.html#backend-functions.

---

**k_local_conv1d**

Apply 1D conv with un-shared weights.

Description

Apply 1D conv with un-shared weights.

Usage

```
k_local_conv1d(inputs, kernel, kernel_size, strides, data_format = NULL)
```
```
k_local_conv2d
149

Arguments
inputs 3D tensor with shape: (batch_size, steps, input_dim)
kernel the unshared weight for convolution, with shape (output_length, feature_dim, filters)
kernel_size a list of a single integer, specifying the length of the 1D convolution window
strides a list of a single integer, specifying the stride length of the convolution
data_format the data format, channels_first or channels_last

Value
the tensor after 1d conv with un-shared weights, with shape (batch_size, output_length, filters)

Keras Backend
This function is part of a set of Keras backend functions that enable lower level access to the core operations of the backend tensor engine (e.g. TensorFlow, CNTK, Theano, etc.).
You can see a list of all available backend functions here: https://keras.rstudio.com/articles/backend.html#backend-functions.

k_local_conv2d
Apply 2D conv with un-shared weights.

Description
Apply 2D conv with un-shared weights.

Usage
k_local_conv2d(
  inputs,
  kernel,
  kernel_size,
  strides,
  output_shape,
  data_format = NULL
)

Arguments
inputs 4D tensor with shape: (batch_size, filters, new_rows, new_cols) if data_format='channels_first' or 4D tensor with shape: (batch_size, new_rows, new_cols, filters) if data_format='channels_last'.
kernel the unshared weight for convolution, with shape (output_items, feature_dim, filters)
kernel_size a list of 2 integers, specifying the width and height of the 2D convolution window.
```
**strides**
a list of 2 integers, specifying the strides of the convolution along the width and height.

**output_shape**
a list with (output_row, output_col)

**data_format**
the data format, channels_first or channels_last

**Value**
A 4d tensor with shape: (batch_size, filters, new_rows, new_cols) if data_format=’channels_first’ or 4D tensor with shape: (batch_size, new_rows, new_cols, filters) if data_format=’channels_last’.

**Keras Backend**
This function is part of a set of Keras backend functions that enable lower level access to the core operations of the backend tensor engine (e.g. TensorFlow, CNTK, Theano, etc.).

You can see a list of all available backend functions here: https://keras.rstudio.com/articles/backend.html#backend-functions.

---

**k_log**

*Element-wise log.*

**Description**
Element-wise log.

**Usage**

```
k_log(x)
```

**Arguments**

- **x**
  Tensor or variable.

**Value**
A tensor.

**Keras Backend**
This function is part of a set of Keras backend functions that enable lower level access to the core operations of the backend tensor engine (e.g. TensorFlow, CNTK, Theano, etc.).

You can see a list of all available backend functions here: https://keras.rstudio.com/articles/backend.html#backend-functions.
**k_logsumexp**

*Computes log(sum(exp(elements across dimensions of a tensor))).*

---

**Description**

This function is more numerically stable than log(sum(exp(x))). It avoids overflows caused by taking the exp of large inputs and underflows caused by taking the log of small inputs.

**Usage**

```r
k_logsumexp(x, axis = NULL, keepdims = FALSE)
```

**Arguments**

- **x**: A tensor or variable.
- **axis**: An integer, the axis to reduce over (axis indexes are 1-based).
- **keepdims**: A boolean, whether to keep the dimensions or not. If `keepdims` is FALSE, the rank of the tensor is reduced by 1. If `keepdims` is TRUE, the reduced dimension is retained with length 1.

**Value**

The reduced tensor.

---

**Keras Backend**

This function is part of a set of Keras backend functions that enable lower level access to the core operations of the backend tensor engine (e.g. TensorFlow, CNTK, Theano, etc.).

You can see a list of all available backend functions here: [https://keras.rstudio.com/articles/backend.html#backend-functions](https://keras.rstudio.com/articles/backend.html#backend-functions).

---

**k_manual_variable-initialization**

*Sets the manual variable initialization flag.*

---

**Description**

This boolean flag determines whether variables should be initialized as they are instantiated (default), or if the user should handle the initialization (e.g. via `tf$initialize_all_variables()`).

**Usage**

```r
k_manual_variable-initialization(value)
```
**k_map_fn**

**Arguments**

- value: Logical

**Keras Backend**

This function is part of a set of Keras backend functions that enable lower level access to the core operations of the backend tensor engine (e.g. TensorFlow, CNTK, Theano, etc.).

You can see a list of all available backend functions here: [https://keras.rstudio.com/articles/backend.html#backend-functions](https://keras.rstudio.com/articles/backend.html#backend-functions).

---

**k_map_fn**  
*Map the function fn over the elements elems and return the outputs.*

---

**Description**

Map the function fn over the elements elems and return the outputs.

**Usage**

```r
k_map_fn(fn, elems, name = NULL, dtype = NULL)
```

**Arguments**

- fn: Function that will be called upon each element in elems
- elems: tensor
- name: A string name for the map node in the graph
- dtype: Output data type.

**Value**

Tensor with dtype dtype.

**Keras Backend**

This function is part of a set of Keras backend functions that enable lower level access to the core operations of the backend tensor engine (e.g. TensorFlow, CNTK, Theano, etc.).

You can see a list of all available backend functions here: [https://keras.rstudio.com/articles/backend.html#backend-functions](https://keras.rstudio.com/articles/backend.html#backend-functions).
**k_max**

Maximum value in a tensor.

**Description**

Maximum value in a tensor.

**Usage**

k_max(x, axis = NULL, keepdims = FALSE)

**Arguments**

- **x**: A tensor or variable.
- **axis**: An integer, the axis to find maximum values (axis indexes are 1-based).
- **keepdims**: A boolean, whether to keep the dimensions or not. If keepdims is FALSE, the rank of the tensor is reduced by 1. If keepdims is TRUE, the reduced dimension is retained with length 1.

**Value**

A tensor with maximum values of x.

**Keras Backend**

This function is part of a set of Keras backend functions that enable lower level access to the core operations of the backend tensor engine (e.g. TensorFlow, CNTK, Theano, etc.).

You can see a list of all available backend functions here: [https://keras.rstudio.com/articles/backend.html#backend-functions](https://keras.rstudio.com/articles/backend.html#backend-functions).

---

**k_maximum**

Element-wise maximum of two tensors.

**Description**

Element-wise maximum of two tensors.

**Usage**

k_maximum(x, y)

**Arguments**

- **x**: Tensor or variable.
- **y**: Tensor or variable.
k_mean

Value
A tensor.

Keras Backend
This function is part of a set of Keras backend functions that enable lower level access to the core operations of the backend tensor engine (e.g. TensorFlow, CNTK, Theano, etc.).

You can see a list of all available backend functions here: https://keras.rstudio.com/articles/backend.html#backend-functions.

---

**k_mean**

| Mean of a tensor, alongside the specified axis. |

---

Description
Mean of a tensor, alongside the specified axis.

Usage
```
k_mean(x, axis = NULL, keepdims = FALSE)
```

Arguments
- `x` A tensor or variable.
- `axis` A list of axes to compute the mean over (axis indexes are 1-based).
- `keepdims` A boolean, whether to keep the dimensions or not. If `keepdims` is FALSE, the rank of the tensor is reduced by 1 for each entry in `axis`. If `keep_dims` is TRUE, the reduced dimensions are retained with length 1.

Value
A tensor with the mean of elements of `x`.

Keras Backend
This function is part of a set of Keras backend functions that enable lower level access to the core operations of the backend tensor engine (e.g. TensorFlow, CNTK, Theano, etc.).

You can see a list of all available backend functions here: https://keras.rstudio.com/articles/backend.html#backend-functions.
**k_min**

*Minimum value in a tensor.*

---

**Description**

Minimum value in a tensor.

**Usage**

`k_min(x, axis = NULL, keepdims = FALSE)`

**Arguments**

- **x**: A tensor or variable.
- **axis**: An integer, axis to find minimum values (axis indexes are 1-based).
- **keepdims**: A boolean, whether to keep the dimensions or not. If `keepdims` is FALSE, the rank of the tensor is reduced by 1. If `keepdims` is TRUE, the reduced dimension is retained with length 1.

**Value**

A tensor with minimum values of `x`.

**Keras Backend**

This function is part of a set of Keras backend functions that enable lower level access to the core operations of the backend tensor engine (e.g. TensorFlow, CNTK, Theano, etc.).

You can see a list of all available backend functions here: [https://keras.rstudio.com/articles/backend.html#backend-functions](https://keras.rstudio.com/articles/backend.html#backend-functions).

---

**k_minimum**

*Element-wise minimum of two tensors.*

---

**Description**

Element-wise minimum of two tensors.

**Usage**

`k_minimum(x, y)`

**Arguments**

- **x**: Tensor or variable.
- **y**: Tensor or variable.
**k_moving_average_update**

**Value**

A tensor.

**Keras Backend**

This function is part of a set of Keras backend functions that enable lower level access to the core operations of the backend tensor engine (e.g. TensorFlow, CNTK, Theano, etc.).

You can see a list of all available backend functions here: https://keras.rstudio.com/articles/backend.html#backend-functions.

---

**k_moving_average_update**

*Compute the moving average of a variable.*

**Description**

Compute the moving average of a variable.

**Usage**

`k_moving_average_update(x, value, momentum)`

**Arguments**

- `x` A Variable.
- `value` A tensor with the same shape as `x`.
- `momentum` The moving average momentum.

**Value**

An operation to update the variable.

**Keras Backend**

This function is part of a set of Keras backend functions that enable lower level access to the core operations of the backend tensor engine (e.g. TensorFlow, CNTK, Theano, etc.).

You can see a list of all available backend functions here: https://keras.rstudio.com/articles/backend.html#backend-functions.
\textit{k\_ndim} \hfill 157

\textit{k\_ndim} \quad \textit{Returns the number of axes in a tensor, as an integer.}

\textbf{Description}

Returns the number of axes in a tensor, as an integer.

\textbf{Usage}

\texttt{k\_ndim(x)}

\textbf{Arguments}

- \texttt{x} \quad \text{Tensor or variable.}

\textbf{Value}

Integer (scalar), number of axes.

\textbf{Keras Backend}

This function is part of a set of Keras backend functions that enable lower level access to the core operations of the backend tensor engine (e.g. TensorFlow, CNTK, Theano, etc.).

You can see a list of all available backend functions here: \url{https://keras.rstudio.com/articles/backend.html#backend-functions}.

\textit{k\_normalize\_batch\_in\_training} \\
\textit{Computes mean and std for batch then apply batch\_normalization on batch.}

\textbf{Description}

Computes mean and std for batch then apply batch\_normalization on batch.

\textbf{Usage}

\texttt{k\_normalize\_batch\_in\_training(x, gamma, beta, reduction\_axes, epsilon = 0.001)}

\textbf{Arguments}

- \texttt{x} \quad \text{Input tensor or variable.}
- \texttt{gamma} \quad \text{Tensor by which to scale the input.}
- \texttt{beta} \quad \text{Tensor with which to center the input.}
- \texttt{reduction\_axes} \quad \text{iterable of integers, axes over which to normalize.}
- \texttt{epsilon} \quad \text{Fuzz factor.}
**Value**

A list length of 3, (normalized_tensor, mean, variance).

**Keras Backend**

This function is part of a set of Keras backend functions that enable lower level access to the core operations of the backend tensor engine (e.g. TensorFlow, CNTK, Theano, etc.).

You can see a list of all available backend functions here: [https://keras.rstudio.com/articles/backend.html#backend-functions](https://keras.rstudio.com/articles/backend.html#backend-functions).

---

**k_not_equal**

*Element-wise inequality between two tensors.*

**Description**

Element-wise inequality between two tensors.

**Usage**

```
k_not_equal(x, y)
```

**Arguments**

- **x**
  - Tensor or variable.

- **y**
  - Tensor or variable.

**Value**

A bool tensor.

**Keras Backend**

This function is part of a set of Keras backend functions that enable lower level access to the core operations of the backend tensor engine (e.g. TensorFlow, CNTK, Theano, etc.).

You can see a list of all available backend functions here: [https://keras.rstudio.com/articles/backend.html#backend-functions](https://keras.rstudio.com/articles/backend.html#backend-functions).
**k.ones**

*Instantiates an all-ones tensor variable and returns it.*

**Description**

Instantiates an all-ones tensor variable and returns it.

**Usage**

```r
k.ones(shape, dtype = NULL, name = NULL)
```

**Arguments**

- `shape` Tuple of integers, shape of returned Keras variable.
- `dtype` String, data type of returned Keras variable.
- `name` String, name of returned Keras variable.

**Value**

A Keras variable, filled with 1.0.

**Keras Backend**

This function is part of a set of Keras backend functions that enable lower level access to the core operations of the backend tensor engine (e.g. TensorFlow, CNTK, Theano, etc.).

You can see a list of all available backend functions here: https://keras.rstudio.com/articles/backend.html#backend-functions.

---

**k.ones_like**

*Instantiates an all-ones variable of the same shape as another tensor.*

**Description**

Instantiates an all-ones variable of the same shape as another tensor.

**Usage**

```r
k.ones_like(x, dtype = NULL, name = NULL)
```

**Arguments**

- `x` Keras variable or tensor.
- `dtype` String, dtype of returned Keras variable. NULL uses the dtype of x.
- `name` String, name for the variable to create.
**Value**

A Keras variable with the shape of x filled with ones.

**Keras Backend**

This function is part of a set of Keras backend functions that enable lower level access to the core operations of the backend tensor engine (e.g. TensorFlow, CNTK, Theano, etc.).

You can see a list of all available backend functions here: https://keras.rstudio.com/articles/backend.html#backend-functions.

---

**k_one_hot**

*Computes the one-hot representation of an integer tensor.*

---

**Description**

Computes the one-hot representation of an integer tensor.

**Usage**

```
k_one_hot(indices, num_classes)
```

**Arguments**

- `indices` nD integer tensor of shape (batch_size, dim1, dim2, ... dim(n-1))
- `num_classes` Integer, number of classes to consider.

**Value**

(n + 1)D one hot representation of the input with shape (batch_size, dim1, dim2, ... dim(n-1), num_classes)

**Keras Backend**

This function is part of a set of Keras backend functions that enable lower level access to the core operations of the backend tensor engine (e.g. TensorFlow, CNTK, Theano, etc.).

You can see a list of all available backend functions here: https://keras.rstudio.com/articles/backend.html#backend-functions.
**k_permute_dimensions**  
*Permutes axes in a tensor.*

**Description**

Permutes axes in a tensor.

**Usage**

```r
k_permute_dimensions(x, pattern)
```

**Arguments**

- `x`  
  Tensor or variable.

- `pattern`  
  A list of dimension indices, e.g. (1, 3, 2). Dimension indices are 1-based.

**Value**

A tensor.

**Keras Backend**

This function is part of a set of Keras backend functions that enable lower level access to the core operations of the backend tensor engine (e.g. TensorFlow, CNTK, Theano, etc.).

You can see a list of all available backend functions here: [https://keras.rstudio.com/articles/backend.html#backend-functions](https://keras.rstudio.com/articles/backend.html#backend-functions).

---

**k_placeholder**  
*Instantiates a placeholder tensor and returns it.*

**Description**

Instantiates a placeholder tensor and returns it.

**Usage**

```r
k_placeholder(
    shape = NULL,
    ndim = NULL,
    dtype = NULL,
    sparse = FALSE,
    name = NULL
)
```
Arguments

- **shape**: Shape of the placeholder (integer list, may include NULL entries).
- **ndim**: Number of axes of the tensor. At least one of shape, ndim must be specified. If both are specified, shape is used.
- **dtype**: Placeholder type.
- **sparse**: Logical, whether the placeholder should have a sparse type.
- **name**: Optional name string for the placeholder.

Value

Tensor instance (with Keras metadata included).

Keras Backend

This function is part of a set of Keras backend functions that enable lower level access to the core operations of the backend tensor engine (e.g. TensorFlow, CNTK, Theano, etc.).

You can see a list of all available backend functions here: https://keras.rstudio.com/articles/backend.html#backend-functions.

---

**k_pool2d**

2D Pooling.

---

Description

2D Pooling.

Usage

```r
k_pool2d(
  x,
  pool_size,
  strides = c(1, 1),
  padding = "valid",
  data_format = NULL,
  pool_mode = "max"
)
```

Arguments

- **x**: Tensor or variable.
- **pool_size**: list of 2 integers.
- **strides**: list of 2 integers.
- **padding**: string, "same" or "valid".
- **data_format**: string, "channels_last" or "channels_first".
- **pool_mode**: string, "max" or "avg".
Value

A tensor, result of 2D pooling.

Keras Backend

This function is part of a set of Keras backend functions that enable lower level access to the core operations of the backend tensor engine (e.g. TensorFlow, CNTK, Theano, etc.). You can see a list of all available backend functions here: https://keras.rstudio.com/articles/backend.html#backend-functions.

---

**k_pool3d**

3D Pooling.

Description

3D Pooling.

Usage

```r
def k_pool3d(
    x, 
    pool_size, 
    strides = c(1, 1, 1), 
    padding = "valid", 
    data_format = NULL, 
    pool_mode = "max"
)
```

Arguments

- `x` Tensor or variable.
- `pool_size` list of 3 integers.
- `strides` list of 3 integers.
- `padding` string, "same" or "valid".
- `data_format` string, "channels_last" or "channels_first".
- `pool_mode` string, "max" or "avg".

Value

A tensor, result of 3D pooling.

Keras Backend

This function is part of a set of Keras backend functions that enable lower level access to the core operations of the backend tensor engine (e.g. TensorFlow, CNTK, Theano, etc.). You can see a list of all available backend functions here: https://keras.rstudio.com/articles/backend.html#backend-functions.
**k_pow**  
*Element-wise exponentiation.*

**Description**  
Element-wise exponentiation.

**Usage**  
\[k\_pow(x, a)\]

**Arguments**

- **x**  
  Tensor or variable.

- **a**  
  \(R\) integer.

**Value**  
A tensor.

**Keras Backend**  
This function is part of a set of Keras backend functions that enable lower level access to the core operations of the backend tensor engine (e.g. TensorFlow, CNTK, Theano, etc.).

You can see a list of all available backend functions here: [https://keras.rstudio.com/articles/backend.html#backend-functions](https://keras.rstudio.com/articles/backend.html#backend-functions).

---

**k_print_tensor**  
*Prints message and the tensor value when evaluated.*

**Description**  
Note that print_tensor returns a new tensor identical to \(x\) which should be used in the following code. Otherwise the print operation is not taken into account during evaluation.

**Usage**  
\[k\_print\_tensor(x, message = "")\]

**Arguments**

- **x**  
  Tensor to print.

- **message**  
  Message to print jointly with the tensor.
**Value**

The same tensor \( x \), unchanged.

**Keras Backend**

This function is part of a set of Keras backend functions that enable lower level access to the core operations of the backend tensor engine (e.g. TensorFlow, CNTK, Theano, etc.).

You can see a list of all available backend functions here: [https://keras.rstudio.com/articles/backend.html#backend-functions](https://keras.rstudio.com/articles/backend.html#backend-functions).

---

### k_prod

**Multiplies the values in a tensor, alongside the specified axis.**

---

**Description**

Multiplies the values in a tensor, alongside the specified axis.

**Usage**

```r
k_prod(x, axis = NULL, keepdims = FALSE)
```

**Arguments**

- `x` A tensor or variable.
- `axis` An integer, axis to compute the product over (axis indexes are 1-based).
- `keepdims` A boolean, whether to keep the dimensions or not. If `keepdims` is `FALSE`, the rank of the tensor is reduced by 1. If `keepdims` is `TRUE`, the reduced dimension is retained with length 1.

**Value**

A tensor with the product of elements of \( x \).

**Keras Backend**

This function is part of a set of Keras backend functions that enable lower level access to the core operations of the backend tensor engine (e.g. TensorFlow, CNTK, Theano, etc.).

You can see a list of all available backend functions here: [https://keras.rstudio.com/articles/backend.html#backend-functions](https://keras.rstudio.com/articles/backend.html#backend-functions).
k_random_binomial

Returns a tensor with random binomial distribution of values.

Description

Returns a tensor with random binomial distribution of values.

Usage

k_random_binomial(shape, p = 0, dtype = NULL, seed = NULL)

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>shape</td>
<td>A list of integers, the shape of tensor to create.</td>
</tr>
<tr>
<td>p</td>
<td>A float, 0. &lt;= p &lt;= 1, probability of binomial distribution.</td>
</tr>
<tr>
<td>dtype</td>
<td>String, dtype of returned tensor.</td>
</tr>
<tr>
<td>seed</td>
<td>Integer, random seed.</td>
</tr>
</tbody>
</table>

Value

A tensor.

Keras Backend

This function is part of a set of Keras backend functions that enable lower level access to the core operations of the backend tensor engine (e.g. TensorFlow, CNTK, Theano, etc.).

You can see a list of all available backend functions here: https://keras.rstudio.com/articles/backend.html#backend-functions.

k_random_normal

Returns a tensor with normal distribution of values.

Description

Returns a tensor with normal distribution of values.

Usage

k_random_normal(shape, mean = 0, stddev = 1, dtype = NULL, seed = NULL)
**Arguments**

- **shape**: A list of integers, the shape of tensor to create.
- **mean**: A float, mean of the normal distribution to draw samples.
- **stddev**: A float, standard deviation of the normal distribution to draw samples.
- **dtype**: String, dtype of returned tensor.
- **seed**: Integer, random seed.

**Value**

A tensor.

**Keras Backend**

This function is part of a set of Keras backend functions that enable lower level access to the core operations of the backend tensor engine (e.g. TensorFlow, CNTK, Theano, etc.).

You can see a list of all available backend functions here: [https://keras.rstudio.com/articles/backend.html#backend-functions](https://keras.rstudio.com/articles/backend.html#backend-functions).

---

**k_random_normal_variable**

*Instantiates a variable with values drawn from a normal distribution.*

---

**Description**

Instantiates a variable with values drawn from a normal distribution.

**Usage**

```python
k_random_normal_variable(
    shape,
    mean,
    scale,
    dtype = NULL,
    name = NULL,
    seed = NULL
)
```

**Arguments**

- **shape**: Tuple of integers, shape of returned Keras variable.
- **mean**: Float, mean of the normal distribution.
- **scale**: Float, standard deviation of the normal distribution.
- **dtype**: String, dtype of returned Keras variable.
- **name**: String, name of returned Keras variable.
- **seed**: Integer, random seed.
**Value**

A Keras variable, filled with drawn samples.

**Keras Backend**

This function is part of a set of Keras backend functions that enable lower level access to the core operations of the backend tensor engine (e.g. TensorFlow, CNTK, Theano, etc.).

You can see a list of all available backend functions here: https://keras.rstudio.com/articles/backend.html#backend-functions.

---

**k_random_uniform**  
*Returns a tensor with uniform distribution of values.*

---

**Description**

Returns a tensor with uniform distribution of values.

**Usage**

```
k_random_uniform(shape, minval = 0, maxval = 1, dtype = NULL, seed = NULL)
```

**Arguments**

- **shape**: A list of integers, the shape of tensor to create.
- **minval**: A float, lower boundary of the uniform distribution to draw samples.
- **maxval**: A float, upper boundary of the uniform distribution to draw samples.
- **dtype**: String, dtype of returned tensor.
- **seed**: Integer, random seed.

**Value**

A tensor.

**Keras Backend**

This function is part of a set of Keras backend functions that enable lower level access to the core operations of the backend tensor engine (e.g. TensorFlow, CNTK, Theano, etc.).

You can see a list of all available backend functions here: https://keras.rstudio.com/articles/backend.html#backend-functions.
Instantiates a variable with values drawn from a uniform distribution.

**Description**

Instantiates a variable with values drawn from a uniform distribution.

**Usage**

```python
k_random_uniform_variable(
    shape,
    low,
    high,
    dtype = NULL,
    name = NULL,
    seed = NULL
)
```

**Arguments**

- **shape**: Tuple of integers, shape of returned Keras variable.
- **low**: Float, lower boundary of the output interval.
- **high**: Float, upper boundary of the output interval.
- **dtype**: String, dtype of returned Keras variable.
- **name**: String, name of returned Keras variable.
- **seed**: Integer, random seed.

**Value**

A Keras variable, filled with drawn samples.

**Keras Backend**

This function is part of a set of Keras backend functions that enable lower level access to the core operations of the backend tensor engine (e.g. TensorFlow, CNTK, Theano, etc.).

You can see a list of all available backend functions here: https://keras.rstudio.com/articles/backend.html#backend-functions.
**k_relu**

*Rectified linear unit.*

**Description**

With default values, it returns element-wise \( \max(x, 0) \).

**Usage**

\[
k_{\text{relu}}(x, \alpha = 0, \text{max\_value} = \text{NULL})
\]

**Arguments**

- **x**
  A tensor or variable.
- **alpha**
  A scalar, slope of negative section (default=0.).
- **max\_value**
  Saturation threshold.

**Value**

A tensor.

**Keras Backend**

This function is part of a set of Keras backend functions that enable lower level access to the core operations of the backend tensor engine (e.g. TensorFlow, CNTK, Theano, etc.).

You can see a list of all available backend functions here: [https://keras.rstudio.com/articles/backend.html#backend-functions](https://keras.rstudio.com/articles/backend.html#backend-functions).

---

**k_repeat**

*Repeats a 2D tensor.*

**Description**

If x has shape (samples, dim) and n is 2, the output will have shape (samples, 2, dim).

**Usage**

\[
k_{\text{repeat}}(x, n)
\]

**Arguments**

- **x**
  Tensor or variable.
- **n**
  Integer, number of times to repeat.
k_repeat_elements

Value

A tensor

Keras Backend

This function is part of a set of Keras backend functions that enable lower level access to the core operations of the backend tensor engine (e.g. TensorFlow, CNTK, Theano, etc.).

You can see a list of all available backend functions here: https://keras.rstudio.com/articles/backend.html#backend-functions.

Description

If \( x \) has shape \( (s1, s2, s3) \) and \( \text{axis} \) is 2, the output will have shape \( (s1, s2 \times \text{rep}, s3) \).

Usage

\[
k\_repeat\_elements(x, \text{rep}, \text{axis})
\]

Arguments

- **x**: Tensor or variable.
- **rep**: Integer, number of times to repeat.
- **axis**: Axis along which to repeat (axis indexes are 1-based)

Value

A tensor.

Keras Backend

This function is part of a set of Keras backend functions that enable lower level access to the core operations of the backend tensor engine (e.g. TensorFlow, CNTK, Theano, etc.).

You can see a list of all available backend functions here: https://keras.rstudio.com/articles/backend.html#backend-functions.
**k_reset_uids**

*Reset graph identifiers.*

**Description**

Reset graph identifiers.

**Usage**

```python
k_reset_uids()
```

**Keras Backend**

This function is part of a set of Keras backend functions that enable lower level access to the core operations of the backend tensor engine (e.g. TensorFlow, CNTK, Theano, etc.).

You can see a list of all available backend functions here: [https://keras.rstudio.com/articles/backend.html#backend-functions](https://keras.rstudio.com/articles/backend.html#backend-functions).

---

**k_reshape**

*Reshapes a tensor to the specified shape.*

**Description**

Reshapes a tensor to the specified shape.

**Usage**

```python
k_reshape(x, shape)
```

**Arguments**

- `x`: Tensor or variable.
- `shape`: Target shape list.

**Value**

A tensor.

**Keras Backend**

This function is part of a set of Keras backend functions that enable lower level access to the core operations of the backend tensor engine (e.g. TensorFlow, CNTK, Theano, etc.).

You can see a list of all available backend functions here: [https://keras.rstudio.com/articles/backend.html#backend-functions](https://keras.rstudio.com/articles/backend.html#backend-functions).
**k_resize_images**

Resizes the images contained in a 4D tensor.

**Description**

Resizes the images contained in a 4D tensor.

**Usage**

\[ \text{k\_resize\_images}(x, \text{height\_factor}, \text{width\_factor}, \text{data\_format}) \]

**Arguments**

- \(x\) Tensor or variable to resize.
- \(\text{height\_factor}\) Positive integer.
- \(\text{width\_factor}\) Positive integer.
- \(\text{data\_format}\) string, "channels\_last" or "channels\_first".

**Value**

A tensor.

**Keras Backend**

This function is part of a set of Keras backend functions that enable lower level access to the core operations of the backend tensor engine (e.g. TensorFlow, CNTK, Theano, etc.).

You can see a list of all available backend functions here: [https://keras.rstudio.com/articles/backend.html#backend-functions](https://keras.rstudio.com/articles/backend.html#backend-functions).

---

**k_resize_volumes**

Resizes the volume contained in a 5D tensor.

**Description**

Resizes the volume contained in a 5D tensor.

**Usage**

\[ \text{k\_resize\_volumes}(x, \text{depth\_factor}, \text{height\_factor}, \text{width\_factor}, \text{data\_format}) \]

**Arguments**

- \(x\) Tensor or variable to resize.
- \(\text{depth\_factor}\) Positive integer.
- \(\text{height\_factor}\) Positive integer.
- \(\text{width\_factor}\) Positive integer.
- \(\text{data\_format}\) string, "channels\_last" or "channels\_first".

**Value**

A tensor.
Arguments

- **x**: Tensor or variable to resize.
- **depth_factor**: Positive integer.
- **height_factor**: Positive integer.
- **width_factor**: Positive integer.
- **data_format**: String, "channels_last" or "channels_first".

Value

A tensor.

Keras Backend

This function is part of a set of Keras backend functions that enable lower level access to the core operations of the backend tensor engine (e.g. TensorFlow, CNTK, Theano, etc.).

You can see a list of all available backend functions here: https://keras.rstudio.com/articles/backend.html#backend-functions.

---

**k_reverse**

Reverse a tensor along the specified axes.

---

Description

Reverse a tensor along the specified axes.

Usage

```
k_reverse(x, axes)
```

Arguments

- **x**: Tensor to reverse.
- **axes**: Integer or list of integers of axes to reverse (axis indexes are 1-based).

Value

A tensor.

Keras Backend

This function is part of a set of Keras backend functions that enable lower level access to the core operations of the backend tensor engine (e.g. TensorFlow, CNTK, Theano, etc.).

You can see a list of all available backend functions here: https://keras.rstudio.com/articles/backend.html#backend-functions.
**k_rnn**

Iterates over the time dimension of a tensor

**Description**

Iterates over the time dimension of a tensor

**Usage**

```r
k_rnn(
    step_function,
    inputs,
    initial_states,
    go_backwards = FALSE,
    mask = NULL,
    constants = NULL,
    unroll = FALSE,
    input_length = NULL
)
```

**Arguments**

- **step_function**: RNN step function.
- **inputs**: Tensor with shape (samples, ...) (no time dimension), representing input for the batch of samples at a certain time step.
- **initial_states**: Tensor with shape (samples, output_dim) (no time dimension), containing the initial values for the states used in the step function.
- **go_backwards**: Logical. If TRUE, do the iteration over the time dimension in reverse order and return the reversed sequence.
- **mask**: Binary tensor with shape (samples, time, 1), with a zero for every element that is masked.
- **constants**: A list of constant values passed at each step.
- **unroll**: Whether to unroll the RNN or to use a symbolic loop (while_loop or scan depending on backend).
- **input_length**: Not relevant in the TensorFlow implementation. Must be specified if using unrolling with Theano.

**Value**

A list with:

- **last_output**: the latest output of the rnn, of shape (samples, ...)
- **outputs**: tensor with shape (samples, time, ...) where each entry outputs[s, t] is the output of the step function at time t for sample s.
- **new_states**: list of tensors, latest states returned by the step function, of shape (samples, ...).
Keras Backend

This function is part of a set of Keras backend functions that enable lower level access to the core operations of the backend tensor engine (e.g. TensorFlow, CNTK, Theano, etc.).

You can see a list of all available backend functions here: https://keras.rstudio.com/articles/backend.html#backend-functions.

---

**k_round**

<table>
<thead>
<tr>
<th>Description</th>
<th>Element-wise rounding to the closest integer.</th>
</tr>
</thead>
</table>

**Description**

In case of tie, the rounding mode used is "half to even".

**Usage**

```
  k_round(x)
```

**Arguments**

- `x` Tensor or variable.

**Value**

A tensor.

---

Keras Backend

This function is part of a set of Keras backend functions that enable lower level access to the core operations of the backend tensor engine (e.g. TensorFlow, CNTK, Theano, etc.).

You can see a list of all available backend functions here: https://keras.rstudio.com/articles/backend.html#backend-functions.

---

**k_separable_conv2d**

<table>
<thead>
<tr>
<th>Description</th>
<th>2D convolution with separable filters.</th>
</tr>
</thead>
</table>

**Description**

2D convolution with separable filters.
**Usage**

\[
\begin{align*}
\text{k\_separable\_conv2d} & (x, \\
& \text{depthwise\_kernel}, \\
& \text{pointwise\_kernel}, \\
& \text{strides} = \text{c}(1, 1), \\
& \text{padding} = \text{"valid"}, \\
& \text{data\_format} = \text{NULL}, \\
& \text{dilation\_rate} = \text{c}(1, 1)
\end{align*}
\]

**Arguments**

- **x**: input tensor
- **depthwise\_kernel**: convolution kernel for the depthwise convolution.
- **pointwise\_kernel**: kernel for the 1x1 convolution.
- **strides**: strides list (length 2).
- **padding**: string, "same" or "valid".
- **data\_format**: string, "channels\_last" or "channels\_first".
- **dilation\_rate**: list of integers, dilation rates for the separable convolution.

**Value**

Output tensor.

**Keras Backend**

This function is part of a set of Keras backend functions that enable lower level access to the core operations of the backend tensor engine (e.g. TensorFlow, CNTK, Theano, etc.).

You can see a list of all available backend functions here: https://keras.rstudio.com/articles/backend.html#backend-functions.

---

**k\_set\_learning\_phase**  
*Sets the learning phase to a fixed value.*

**Description**

Sets the learning phase to a fixed value.

**Usage**

\[
\text{k\_set\_learning\_phase}(\text{value})
\]
**Arguments**

value  Learning phase value, either 0 or 1 (integers).

**Keras Backend**

This function is part of a set of Keras backend functions that enable lower level access to the core operations of the backend tensor engine (e.g. TensorFlow, CNTK, Theano, etc.).

You can see a list of all available backend functions here: https://keras.rstudio.com/articles/backend.html#backend-functions.

---

**k_set_value**  *Sets the value of a variable, from an R array.*

**Description**

Sets the value of a variable, from an R array.

**Usage**

```r
k_set_value(x, value)
```

**Arguments**

- **x**  Tensor to set to a new value.
- **value**  Value to set the tensor to, as an R array (of the same shape).

**Keras Backend**

This function is part of a set of Keras backend functions that enable lower level access to the core operations of the backend tensor engine (e.g. TensorFlow, CNTK, Theano, etc.).

You can see a list of all available backend functions here: https://keras.rstudio.com/articles/backend.html#backend-functions.

---

**k_shape**  *Returns the symbolic shape of a tensor or variable.*

**Description**

Returns the symbolic shape of a tensor or variable.

**Usage**

```r
k_shape(x)
```
**k_sigmoid**

**Arguments**

- **x**  
  A tensor or variable.

**Value**

A symbolic shape (which is itself a tensor).

**Keras Backend**

This function is part of a set of Keras backend functions that enable lower level access to the core operations of the backend tensor engine (e.g. TensorFlow, CNTK, Theano, etc.).

You can see a list of all available backend functions here: [https://keras.rstudio.com/articles/backend.html#backend-functions](https://keras.rstudio.com/articles/backend.html#backend-functions).

**Description**

Element-wise sigmoid.

**Usage**

```python
k_sigmoid(x)
```

**Arguments**

- **x**  
  A tensor or variable.

**Value**

A tensor.

**Keras Backend**

This function is part of a set of Keras backend functions that enable lower level access to the core operations of the backend tensor engine (e.g. TensorFlow, CNTK, Theano, etc.).

You can see a list of all available backend functions here: [https://keras.rstudio.com/articles/backend.html#backend-functions](https://keras.rstudio.com/articles/backend.html#backend-functions).
**k_sign**  
*Element-wise sign.*

**Description**  
Element-wise sign.

**Usage**  
```python  
k_sign(x)  
```  

**Arguments**  
- `x`: Tensor or variable.

**Value**  
A tensor.

**Keras Backend**  
This function is part of a set of Keras backend functions that enable lower level access to the core operations of the backend tensor engine (e.g. TensorFlow, CNTK, Theano, etc.).

You can see a list of all available backend functions here: [https://keras.rstudio.com/articles/backend.html#backend-functions](https://keras.rstudio.com/articles/backend.html#backend-functions).

**k_sin**  
*Computes sin of x element-wise.*

**Description**  
Computes sin of x element-wise.

**Usage**  
```python  
k_sin(x)  
```  

**Arguments**  
- `x`: Tensor or variable.

**Value**  
A tensor.
**Keras Backend**

This function is part of a set of Keras backend functions that enable lower level access to the core operations of the backend tensor engine (e.g. TensorFlow, CNTK, Theano, etc.).

You can see a list of all available backend functions here: [https://keras.rstudio.com/articles/backend.html#backend-functions](https://keras.rstudio.com/articles/backend.html#backend-functions).

---

**k_softmax**

Softmax of a tensor.

**Description**

Softmax of a tensor.

**Usage**

```python
k_softmax(x, axis = -1)
```

**Arguments**

- **x**: A tensor or variable.
- **axis**: The dimension softmax would be performed on. The default is -1 which indicates the last dimension.

**Value**

A tensor.
k_softplus

Softplus of a tensor.

Description
Softplus of a tensor.

Usage
k_softplus(x)

Arguments
x A tensor or variable.

Value
A tensor.

Keras Backend
This function is part of a set of Keras backend functions that enable lower level access to the core operations of the backend tensor engine (e.g. TensorFlow, CNTK, Theano, etc.).
You can see a list of all available backend functions here: https://keras.rstudio.com/articles/backend.html#backend-functions.

k_softsign

Softsign of a tensor.

Description
Softsign of a tensor.

Usage
k_softsign(x)

Arguments
x A tensor or variable.

Value
A tensor.
**k_sparse_categorical_crossentropy**

*Categorical crossentropy with integer targets.*

**Description**

Categorical crossentropy with integer targets.

**Usage**

```r
k_sparse_categorical_crossentropy(
  target,
  output,
  from_logits = FALSE,
  axis = -1
)
```

**Arguments**

- `target`: An integer tensor.
- `output`: A tensor resulting from a softmax (unless `from_logits` is TRUE, in which case output is expected to be the logits).
- `from_logits`: Boolean, whether output is the result of a softmax, or is a tensor of logits.
- `axis`: Axis (axis indexes are 1-based). Pass -1 (the default) to select the last axis.

**Value**

Output tensor.

**Keras Backend**

This function is part of a set of Keras backend functions that enable lower level access to the core operations of the backend tensor engine (e.g. TensorFlow, CNTK, Theano, etc.).

You can see a list of all available backend functions here: [https://keras.rstudio.com/articles/backend.html#backend-functions](https://keras.rstudio.com/articles/backend.html#backend-functions).
**k_spatial_2d_padding**  
Pads the 2nd and 3rd dimensions of a 4D tensor.

**Description**

Pads the 2nd and 3rd dimensions of a 4D tensor.

**Usage**

```r
k_spatial_2d_padding(
  x,
  padding = list(list(1, 1), list(1, 1)),
  data_format = NULL
)
```

**Arguments**

- `x` Tensor or variable.
- `padding` Tuple of 2 lists, padding pattern.
- `data_format` string, "channels_last" or "channels_first".

**Value**

A padded 4D tensor.

**Keras Backend**

This function is part of a set of Keras backend functions that enable lower level access to the core operations of the backend tensor engine (e.g. TensorFlow, CNTK, Theano, etc.).

You can see a list of all available backend functions here: [https://keras.rstudio.com/articles/backend.html#backend-functions.](https://keras.rstudio.com/articles/backend.html#backend-functions).

---

**k_spatial_3d_padding**  
Pads 5D tensor with zeros along the depth, height, width dimensions.

**Description**

Pads these dimensions with respectively padding[[1]], padding[[2]], and padding[[3]] zeros left and right. For 'channels_last' data_format, the 2nd, 3rd and 4th dimension will be padded. For 'channels_first' data_format, the 3rd, 4th and 5th dimension will be padded.
### k_sqrt

#### Usage

```python
ek_square_3d_padding(
    x,
    padding = list(list(1, 1), list(1, 1), list(1, 1)),
    data_format = NULL
)
```

#### Arguments

- `x` Tensor or variable.
- `padding` List of 3 lists, padding pattern.
- `data_format` string, "channels_last" or "channels_first".

#### Value

A padded 5D tensor.

#### Keras Backend

This function is part of a set of Keras backend functions that enable lower level access to the core operations of the backend tensor engine (e.g. TensorFlow, CNTK, Theano, etc.).

You can see a list of all available backend functions here: [https://keras.rstudio.com/articles/backend.html#backend-functions](https://keras.rstudio.com/articles/backend.html#backend-functions).

---

### k_sqrt

Element-wise square root.

#### Description

Element-wise square root.

#### Usage

```python
k_sqrt(x)
```

#### Arguments

- `x` Tensor or variable.

#### Value

A tensor.
Keras Backend

This function is part of a set of Keras backend functions that enable lower level access to the core operations of the backend tensor engine (e.g. TensorFlow, CNTK, Theano, etc.).

You can see a list of all available backend functions here: https://keras.rstudio.com/articles/backend.html#backend-functions.

---

k_square

**Element-wise square.**

**Description**

Element-wise square.

**Usage**

\[
k\_\text{square}(x)
\]

**Arguments**

\(x\)

Tensor or variable.

**Value**

A tensor.

---

Keras Backend

This function is part of a set of Keras backend functions that enable lower level access to the core operations of the backend tensor engine (e.g. TensorFlow, CNTK, Theano, etc.).

You can see a list of all available backend functions here: https://keras.rstudio.com/articles/backend.html#backend-functions.

---

k_squeeze

**Removes a 1-dimension from the tensor at index axis.**

**Description**

Removes a 1-dimension from the tensor at index axis.

**Usage**

\[
k\_\text{squeeze}(x, \text{axis})
\]
**k_stack**  

**Arguments**

- **x**: A tensor or variable.
- **axis**: Axis to drop (axis indexes are 1-based).

**Value**

A tensor with the same data as `x` but reduced dimensions.

**Keras Backend**

This function is part of a set of Keras backend functions that enable lower level access to the core operations of the backend tensor engine (e.g. TensorFlow, CNTK, Theano, etc.).

You can see a list of all available backend functions here: [https://keras.rstudio.com/articles/backend.html#backend-functions](https://keras.rstudio.com/articles/backend.html#backend-functions).

---

**Description**

Stacks a list of rank R tensors into a rank R+1 tensor.

**Usage**

```
k_stack(x, axis = 1)
```

**Arguments**

- **x**: List of tensors.
- **axis**: Axis along which to perform stacking (axis indexes are 1-based).

**Value**

A tensor.

**Keras Backend**

This function is part of a set of Keras backend functions that enable lower level access to the core operations of the backend tensor engine (e.g. TensorFlow, CNTK, Theano, etc.).

You can see a list of all available backend functions here: [https://keras.rstudio.com/articles/backend.html#backend-functions](https://keras.rstudio.com/articles/backend.html#backend-functions).
**k_std**

*Standard deviation of a tensor, alongside the specified axis.*

**Description**

Standard deviation of a tensor, alongside the specified axis.

**Usage**

```r
k_std(x, axis = NULL, keepdims = FALSE)
```

**Arguments**

- **x**
  A tensor or variable.

- **axis**
  An integer, the axis to compute the standard deviation over (axis indexes are 1-based).

- **keepdims**
  A boolean, whether to keep the dimensions or not. If `keepdims` is FALSE, the rank of the tensor is reduced by 1. If `keepdims` is TRUE, the reduced dimension is retained with length 1.

**Value**

A tensor with the standard deviation of elements of `x`.

**Keras Backend**

This function is part of a set of Keras backend functions that enable lower level access to the core operations of the backend tensor engine (e.g. TensorFlow, CNTK, Theano, etc.). You can see a list of all available backend functions here: https://keras.rstudio.com/articles/backend.html#backend-functions.

---

**k_stop_gradient**

*Returns variables but with zero gradient w.r.t. every other variable.*

**Description**

Returns variables but with zero gradient w.r.t. every other variable.

**Usage**

```r
k_stop_gradient(variables)
```

**Arguments**

- **variables**
  tensor or list of tensors to consider constant with respect to any other variable.
**Value**

A single tensor or a list of tensors (depending on the passed argument) that has constant gradient with respect to any other variable.

**Keras Backend**

This function is part of a set of Keras backend functions that enable lower level access to the core operations of the backend tensor engine (e.g. TensorFlow, CNTK, Theano, etc.).

You can see a list of all available backend functions here: https://keras.rstudio.com/articles/backend.html#backend-functions.

---

### k_sum

Sum of the values in a tensor, alongside the specified axis.

**Description**

Sum of the values in a tensor, alongside the specified axis.

**Usage**

```r
k_sum(x, axis = NULL, keepdims = FALSE)
```

**Arguments**

- `x` A tensor or variable.
- `axis` An integer, the axis to sum over (axis indexes are 1-based).
- `keepdims` A boolean, whether to keep the dimensions or not. If `keepdims` is FALSE, the rank of the tensor is reduced by 1. If `keepdims` is TRUE, the reduced dimension is retained with length 1.

**Value**

A tensor with sum of `x`.

**Keras Backend**

This function is part of a set of Keras backend functions that enable lower level access to the core operations of the backend tensor engine (e.g. TensorFlow, CNTK, Theano, etc.).

You can see a list of all available backend functions here: https://keras.rstudio.com/articles/backend.html#backend-functions.
k_switch

*Switches between two operations depending on a scalar value.*

**Description**

Note that both `then_expression` and `else_expression` should be symbolic tensors of the same shape.

**Usage**

```
k_switch(condition, then_expression, else_expression)
```

**Arguments**

- `condition`: tensor (int or bool).
- `then_expression`: either a tensor, or a function that returns a tensor.
- `else_expression`: either a tensor, or a function that returns a tensor.

**Value**

The selected tensor.

**Keras Backend**

This function is part of a set of Keras backend functions that enable lower level access to the core operations of the backend tensor engine (e.g. TensorFlow, CNTK, Theano, etc.).

You can see a list of all available backend functions here: https://keras.rstudio.com/articles/backend.html#backend-functions.

k_tanh

*Element-wise tanh.*

**Description**

Element-wise tanh.

**Usage**

```
k_tanh(x)
```

**Arguments**

- `x`: A tensor or variable.
**k_temporal_padding**

**Value**

A tensor.

**Keras Backend**

This function is part of a set of Keras backend functions that enable lower level access to the core operations of the backend tensor engine (e.g. TensorFlow, CNTK, Theano, etc.).

You can see a list of all available backend functions here: [https://keras.rstudio.com/articles/backend.html#backend-functions](https://keras.rstudio.com/articles/backend.html#backend-functions).

---

**k_temporal_padding**  
*Pads the middle dimension of a 3D tensor.*

**Description**

Pads the middle dimension of a 3D tensor.

**Usage**

```
k_temporal_padding(x, padding = c(1, 1))
```

**Arguments**

- `x`  
  Tensor or variable.
- `padding`  
  List of 2 integers, how many zeros to add at the start and end of dim 1.

**Value**

A padded 3D tensor.

**Keras Backend**

This function is part of a set of Keras backend functions that enable lower level access to the core operations of the backend tensor engine (e.g. TensorFlow, CNTK, Theano, etc.).

You can see a list of all available backend functions here: [https://keras.rstudio.com/articles/backend.html#backend-functions](https://keras.rstudio.com/articles/backend.html#backend-functions).
**k_tile**

*Creates a tensor by tiling x by n.*

**Description**

Creates a tensor by tiling x by n.

**Usage**

```
k_tile(x, n)
```

**Arguments**

- **x**: A tensor or variable
- **n**: A list of integers. The length must be the same as the number of dimensions in x.

**Value**

A tiled tensor.

**Keras Backend**

This function is part of a set of Keras backend functions that enable lower level access to the core operations of the backend tensor engine (e.g. TensorFlow, CNTK, Theano, etc.).

You can see a list of all available backend functions here: <https://keras.rstudio.com/articles/backend.html#backend-functions>.

---

**k_to_dense**

*Converts a sparse tensor into a dense tensor and returns it.*

**Description**

Converts a sparse tensor into a dense tensor and returns it.

**Usage**

```
k_to_dense(tensor)
```

**Arguments**

- **tensor**: A tensor instance (potentially sparse).

**Value**

A dense tensor.
**k_transpose**

Transposes a tensor and returns it.

**Description**

Transposes a tensor and returns it.

**Usage**

```python
k_transpose(x)
```

**Arguments**

- `x`: Tensor or variable.

**Value**

A tensor.

---

**Keras Backend**

This function is part of a set of Keras backend functions that enable lower level access to the core operations of the backend tensor engine (e.g. TensorFlow, CNTK, Theano, etc.).

You can see a list of all available backend functions here: [https://keras.rstudio.com/articles/backend.html#backend-functions](https://keras.rstudio.com/articles/backend.html#backend-functions).

---

**k_truncated_normal**

Returns a tensor with truncated random normal distribution of values.

**Description**

The generated values follow a normal distribution with specified mean and standard deviation, except that values whose magnitude is more than two standard deviations from the mean are dropped and re-picked.

**Usage**

```python
k_truncated_normal(shape, mean = 0, stddev = 1, dtype = NULL, seed = NULL)
```
Arguments

- **shape**: A list of integers, the shape of tensor to create.
- **mean**: Mean of the values.
- **stddev**: Standard deviation of the values.
- **dtype**: String, dtype of returned tensor.
- **seed**: Integer, random seed.

Value

A tensor.

Keras Backend

This function is part of a set of Keras backend functions that enable lower level access to the core operations of the backend tensor engine (e.g. TensorFlow, CNTK, Theano, etc.).

You can see a list of all available backend functions here: https://keras.rstudio.com/articles/backend.html#backend-functions.

---

**k_update**

*Update the value of x to new_x.*

Description

Update the value of x to new_x.

Usage

```
k_update(x, new_x)
```

Arguments

- **x**: A Variable.
- **new_x**: A tensor of same shape as x.

Value

The variable x updated.

Keras Backend

This function is part of a set of Keras backend functions that enable lower level access to the core operations of the backend tensor engine (e.g. TensorFlow, CNTK, Theano, etc.).

You can see a list of all available backend functions here: https://keras.rstudio.com/articles/backend.html#backend-functions.
**k_update_add**

Update the value of \( x \) by adding \( \text{increment} \).

**Description**

Update the value of \( x \) by adding \( \text{increment} \).

**Usage**

\[
k_{\text{update\_add}}(x, \text{increment})
\]

**Arguments**

\[
x \quad \text{A Variable.}
\]

\[
\text{increment} \quad \text{A tensor of same shape as } x.
\]

**Value**

The variable \( x \) updated.

**Keras Backend**

This function is part of a set of Keras backend functions that enable lower level access to the core operations of the backend tensor engine (e.g. TensorFlow, CNTK, Theano, etc.).

You can see a list of all available backend functions here: [https://keras.rstudio.com/articles/backend.html#backend-functions](https://keras.rstudio.com/articles/backend.html#backend-functions).

---

**k_update_sub**

Update the value of \( x \) by subtracting \( \text{decrement} \).

**Description**

Update the value of \( x \) by subtracting \( \text{decrement} \).

**Usage**

\[
k_{\text{update\_sub}}(x, \text{decrement})
\]

**Arguments**

\[
x \quad \text{A Variable.}
\]

\[
\text{decrement} \quad \text{A tensor of same shape as } x.
\]

**Value**

The variable \( x \) updated.
**k_var**

Variance of a tensor, alongside the specified axis.

### Description

Variance of a tensor, alongside the specified axis.

### Usage

```
k_var(x, axis = NULL, keepdims = FALSE)
```

### Arguments

- **x**: A tensor or variable.
- **axis**: An integer, the axis to compute the variance over (axis indexes are 1-based).
- **keepdims**: A boolean, whether to keep the dimensions or not. If `keepdims` is `FALSE`, the rank of the tensor is reduced by 1. If `keepdims` is `TRUE`, the reduced dimension is retained with length 1.

### Value

A tensor with the variance of elements of `x`.

---

**Keras Backend**

This function is part of a set of Keras backend functions that enable lower level access to the core operations of the backend tensor engine (e.g. TensorFlow, CNTK, Theano, etc.).

You can see a list of all available backend functions here: [https://keras.rstudio.com/articles/backend.html#backend-functions](https://keras.rstudio.com/articles/backend.html#backend-functions).

---

**Keras Backend**

This function is part of a set of Keras backend functions that enable lower level access to the core operations of the backend tensor engine (e.g. TensorFlow, CNTK, Theano, etc.).

You can see a list of all available backend functions here: [https://keras.rstudio.com/articles/backend.html#backend-functions](https://keras.rstudio.com/articles/backend.html#backend-functions).
**k_variable**

*Instantiates a variable and returns it.*

**Description**

Instantiates a variable and returns it.

**Usage**

```r
k_variable(value, dtype = NULL, name = NULL, constraint = NULL)
```

**Arguments**

- `value`  
  Numpy array, initial value of the tensor.
- `dtype`  
  Tensor type.
- `name`  
  Optional name string for the tensor.
- `constraint`  
  Optional projection function to be applied to the variable after an optimizer update.

**Value**

A variable instance (with Keras metadata included).

**Keras Backend**

This function is part of a set of Keras backend functions that enable lower level access to the core operations of the backend tensor engine (e.g. TensorFlow, CNTK, Theano, etc.).

You can see a list of all available backend functions here: [https://keras.rstudio.com/articles/backend.html#backend-functions](https://keras.rstudio.com/articles/backend.html#backend-functions).

**k_zeros**

*Instantiates an all-zeros variable and returns it.*

**Description**

Instantiates an all-zeros variable and returns it.

**Usage**

```r
k_zeros(shape, dtype = NULL, name = NULL)
```

**Arguments**

- `shape`  
  Tuple of integers, shape of returned Keras variable
- `dtype`  
  String, data type of returned Keras variable
- `name`  
  String, name of returned Keras variable
**k_zeros_like**

**Value**

A variable (including Keras metadata), filled with 0.0.

**Keras Backend**

This function is part of a set of Keras backend functions that enable lower level access to the core operations of the backend tensor engine (e.g. TensorFlow, CNTK, Theano, etc.).

You can see a list of all available backend functions here: https://keras.rstudio.com/articles/backend.html#backend-functions.

**Description**

Instantiates an all-zeros variable of the same shape as another tensor.

**Usage**

```
k_zeros_like(x, dtype = NULL, name = NULL)
```

**Arguments**

- **x**: Keras variable or Keras tensor.
- **dtype**: String, dtype of returned Keras variable. NULL uses the dtype of x.
- **name**: String, name for the variable to create.

**Value**

A Keras variable with the shape of x filled with zeros.

**Keras Backend**

This function is part of a set of Keras backend functions that enable lower level access to the core operations of the backend tensor engine (e.g. TensorFlow, CNTK, Theano, etc.).

You can see a list of all available backend functions here: https://keras.rstudio.com/articles/backend.html#backend-functions.
Layer

Create a custom Layer

Description

Create a custom Layer

Usage

Layer(
    classname,
    initialize,
    build = NULL,
    call = NULL,
    compute_output_shape = NULL,
    ...
    inherit = tensorflow::tf$keras$layers$Layer
)

Arguments

classname the name of the custom Layer.
initialize a function. This is where you define the arguments used to further build your
layer. For example, a dense layer would take the units argument. You should
always call super()$\_\_init\_\_$( ) to initialize the base inherited layer.

build a function that takes input_shape as argument. This is where you will define
your weights. Note that if your layer doesn’t define trainable weights then you
need not implement this method.

call This is where the layer’s logic lives. Unless you want your layer to support
masking, you only have to care about the first argument passed to call (the
input tensor).

compute_output_shape a function that takes input_shape as an argument. In case your layer modifies
the shape of its input, you should specify here the shape transformation logic.
This allows Keras to do automatic shape inference. If you don’t modify the
shape of the input then you need not implement this method.

... Any other methods and/or attributes can be specified using named arguments.
They will be added to the layer class.

inherit the Keras layer to inherit from

Value

A function that wraps create_layer, similar to keras::layer_dense.
Examples

## Not run:

layer_dense2 <- Layer(
  "Dense2",
  initialize = function(units) {
    super()$`__init__`()
    self$units <- as.integer(units)
  },
  build = function(input_shape) {
    print(class(input_shape))
    self$kernel <- self$add_weight(
      name = "kernel",
      shape = list(input_shape[[2]], self$units),
      initializer = "uniform",
      trainable = TRUE
    )
  },
  call = function(x) {
    tensorflow::tf$matmul(x, self$kernel)
  },
  compute_output_shape = function(input_shape) {
    list(input_shape[[1]], self$units)
  }
)

l <- layer_dense2(units = 10)
l(matrix(runif(10), ncol = 1))

## End(Not run)

layer_activation

Apply an activation function to an output.

Description

Apply an activation function to an output.

Usage

layer_activation
layer_activation_elu

object,
activation,
input_shape = NULL,
batch_input_shape = NULL,
batch_size = NULL,
dtype = NULL,
name = NULL,
trainable = NULL,
weights = NULL
)

Arguments

object Model or layer object
activation Name of activation function to use. If you don’t specify anything, no activation is applied (ie. "linear" activation: a(x) = x).
input_shape Input shape (list of integers, does not include the samples axis) which is required when using this layer as the first layer in a model.
batch_input_shape Shapes, including the batch size. For instance, batch_input_shape=c(10,32) indicates that the expected input will be batches of 10 32-dimensional vectors. batch_input_shape=list(NULL,32) indicates batches of an arbitrary number of 32-dimensional vectors.
batch_size Fixed batch size for layer
dtype The data type expected by the input, as a string (float32, float64, int32...)
name An 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.
trainable Whether the layer weights will be updated during training.
weights Initial weights for layer.

See Also

Other core layers: layer_activity_regularization(), layer_attention(), layer_dense_features(), layer_dense(), layer_dropout(), layer_flatten(), layer_input(), layer_lambda(), layer_masking(), layer_permute(), layer_repeat_vector(), layer_reshape()

Other activation layers: layer_activation_elu(), layer_activation_leaky_relu(), layer_activation_parametric(), layer_activation_relu(), layer_activation_selu(), layer_activation_softmax(), layer_activation_thresholded_relu()

layer_activation_elu  Exponential Linear Unit.

Description

It follows: f(x) = alpha * (exp(x) -1.0) for x < 0, f(x) = x for x >= 0.
Usage

```r
layer_activation_elu(
    object,
    alpha = 1,
    input_shape = NULL,
    batch_input_shape = NULL,
    batch_size = NULL,
    dtype = NULL,
    name = NULL,
    trainable = NULL,
    weights = NULL
)
```

Arguments

- **object**: Model or layer object
- **alpha**: Scale for the negative factor.
- **input_shape**: Input shape (list of integers, does not include the samples axis) which is required when using this layer as the first layer in a model.
- **batch_input_shape**: Shapes, including the batch size. For instance, `batch_input_shape=c(10,32)` indicates that the expected input will be batches of 10 32-dimensional vectors. `batch_input_shape=list(NULL,32)` indicates batches of an arbitrary number of 32-dimensional vectors.
- **batch_size**: Fixed batch size for layer
- **dtype**: The data type expected by the input, as a string (float32, float64, int32...)
- **name**: An 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.
- **trainable**: Whether the layer weights will be updated during training.
- **weights**: Initial weights for layer.

See Also

- Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs).
- Other activation layers: `layer_activation_leaky_relu()`, `layer_activation_parametric_relu()`, `layer_activation relu()`, `layer_activation_selu()`, `layer_activation_softmax()`, `layer_activation_thresholded_relu()`
layer_activation_leaky_relu

*Leaky version of a Rectified Linear Unit.*

**Description**

Allows a small gradient when the unit is not active: \( f(x) = \alpha x \) for \( x < 0 \), \( f(x) = x \) for \( x \geq 0 \).

**Usage**

```r
layer_activation_leaky_relu(
    object,
    alpha = 0.3,
    input_shape = NULL,
    batch_input_shape = NULL,
    batch_size = NULL,
    dtype = NULL,
    name = NULL,
    trainable = NULL,
    weights = NULL
)
```

**Arguments**

- **object**: Model or layer object
- **alpha**: `float >= 0`. Negative slope coefficient.
- **input_shape**: Input shape (list of integers, does not include the samples axis) which is required when using this layer as the first layer in a model.
- **batch_input_shape**: Shapes, including the batch size. For instance, `batch_input_shape=c(10, 32)` indicates that the expected input will be batches of 10 32-dimensional vectors. `batch_input_shape=list(NULL, 32)` indicates batches of an arbitrary number of 32-dimensional vectors.
- **batch_size**: Fixed batch size for layer
- **dtype**: The data type expected by the input, as a string (`float32`, `float64`, `int32`...)
- **name**: An 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.
- **trainable**: Whether the layer weights will be updated during training.
- **weights**: Initial weights for layer.
See Also

Rectifier Nonlinearities Improve Neural Network Acoustic Models.
Other activation layers: `layer_activation_elu()`, `layer_activation_parametric_relu()`, `layer_activation_relu()`, `layer_activation_selu()`, `layer_activation_softmax()`, `layer_activation_thresholded_relu()`, `layer_activation()`

---

**layer_activation_parametric_relu**

*Parametric Rectified Linear Unit.*

**Description**

It follows: \( f(x) = \alpha \times x \) for \( x < 0 \), \( f(x) = x \) for \( x \geq 0 \), where \( \alpha \) is a learned array with the same shape as \( x \).

**Usage**

```r
layer_activation_parametric_relu(
    object,
    alpha_initializer = "zeros",
    alpha_regularizer = NULL,
    alpha_constraint = NULL,
    shared_axes = NULL,
    input_shape = NULL,
    batch_input_shape = NULL,
    batch_size = NULL,
    dtype = NULL,
    name = NULL,
    trainable = NULL,
    weights = NULL
)
```

**Arguments**

- `object` Model or layer object
- `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=c(1, 2)`.
layer_activation_relu

| input_shape | Input shape (list of integers, does not include the samples axis) which is required when using this layer as the first layer in a model. |
| batch_input_shape | Shapes, including the batch size. For instance, batch_input_shape=c(10, 32) indicates that the expected input will be batches of 10 32-dimensional vectors. batch_input_shape=list(NULL, 32) indicates batches of an arbitrary number of 32-dimensional vectors. |
| batch_size | Fixed batch size for layer |
| dtype | The data type expected by the input, as a string (float32, float64, int32...) |
| name | An 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. |
| trainable | Whether the layer weights will be updated during training. |
| weights | Initial weights for layer. |

See Also

Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification.

Other activation layers: layer_activation_elu(), layer_activation_leaky_relu(), layer_activation_relu(), layer_activation_selu(), layer_activation_softmax(), layer_activation_thresholded_relu(), layer_activation()
layer_activation_selu

Arguments

- **object**: Model or layer object
- **max_value**: float, the maximum output value.
- **negative_slope**: float >= 0 Negative slope coefficient.
- **threshold**: float. Threshold value for thresholded activation.
- **input_shape**: Input shape (list of integers, does not include the samples axis) which is required when using this layer as the first layer in a model.
- **batch_input_shape**: Shapes, including the batch size. For instance, batch_input_shape=c(10,32) indicates that the expected input will be batches of 10 32-dimensional vectors. batch_input_shape=list(NULL,32) indicates batches of an arbitrary number of 32-dimensional vectors.
- **batch_size**: Fixed batch size for layer
- **dtype**: The data type expected by the input, as a string (float32, float64, int32...)
- **name**: An 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.
- **trainable**: Whether the layer weights will be updated during training.
- **weights**: Initial weights for layer.

See Also

Other activation layers: layer_activation_elu(), layer_activation_leaky_relu(), layer_activation_parametric_relu(), layer_activation_selu(), layer_activation_softmax(), layer_activation_thresholded_relu(), layer_activation()

layer_activation_selu  Scaled Exponential Linear Unit.

Description

SELU is equal to: scale * elu(x, alpha), where alpha and scale are pre-defined constants.

Usage

layer_activation_selu(
  object,
  input_shape = NULL,
  batch_input_shape = NULL,
  batch_size = NULL,
  dtype = NULL,
  name = NULL,
  trainable = NULL,
  weights = NULL
)
layer_activation_softmax

Arguments

object Model or layer object
input_shape Input shape (list of integers, does not include the samples axis) which is required when using this layer as the first layer in a model.
batch_input_shape Shapes, including the batch size. For instance, batch_input_shape=c(10, 32) indicates that the expected input will be batches of 10 32-dimensional vectors. batch_input_shape=list(NULL, 32) indicates batches of an arbitrary number of 32-dimensional vectors.
batch_size Fixed batch size for layer
dtype The data type expected by the input, as a string (float32, float64, int32...)
name An 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.
trainable Whether the layer weights will be updated during training.
weights Initial weights for layer.

Details

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 inputs is "large enough" (see article for more information).

Note:

- To be used together with the initialization "lecun_normal".
- To be used together with the dropout variant "AlphaDropout".

See Also

Self-Normalizing Neural Networks, initializer_lecun_normal, layer_alpha_dropout

Other activation layers: layer_activation_elu(), layer_activation_leaky_relu(), layer_activation_parametric_relu(), layer_activation_relu(), layer_activation_softmax(), layer_activation_thresholded_relu(), layer_activation()
Usage

layer_activation_softmax(
  object,
  axis = -1,
  input_shape = NULL,
  batch_input_shape = NULL,
  batch_size = NULL,
  dtype = NULL,
  name = NULL,
  trainable = NULL,
  weights = NULL
)

Arguments

object       Model or layer object
axis         Integer, axis along which the softmax normalization is applied.
input_shape  Input shape (list of integers, does not include the samples axis) which is required when using this layer as the first layer in a model.
batch_input_shape
  Shapes, including the batch size. For instance, batch_input_shape=c(10,32) indicates that the expected input will be batches of 10 32-dimensional vectors.
  batch_input_shape=list(NULL,32) indicates batches of an arbitrary number of 32-dimensional vectors.
batch_size   Fixed batch size for layer
dtype        The data type expected by the input, as a string (float32, float64, int32...)
name         An 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.
trainable    Whether the layer weights will be updated during training.
weights      Initial weights for layer.

See Also

Other activation layers: layer_activation_elu(), layer_activation_leaky_relu(), layer_activation_parametric_relu(), layer_activation_relu(), layer_activation_selu(), layer_activation_thresholded_relu(), layer_activation()

layer_activation_thresholded_relu

Thresholded Rectified Linear Unit.

Description

It follows: \( f(x) = x \) for \( x > \theta \), \( f(x) = 0 \) otherwise.
layer_activity_regularization

Usage

layer_activation_thresholded_relu(
    object,
    theta = 1,
    input_shape = NULL,
    batch_input_shape = NULL,
    batch_size = NULL,
    dtype = NULL,
    name = NULL,
    trainable = NULL,
    weights = NULL
)

Arguments

object Model or layer object
theta float >= 0. Threshold location of activation.
input_shape Input shape (list of integers, does not include the samples axis) which is required when using this layer as the first layer in a model.
batch_input_shape Shapes, including the batch size. For instance, batch_input_shape=c(10, 32) indicates that the expected input will be batches of 10 32-dimensional vectors. batch_input_shape=list(NULL, 32) indicates batches of an arbitrary number of 32-dimensional vectors.
batch_size Fixed batch size for layer
dtype The data type expected by the input, as a string (float32, float64, int32...)
name An 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.
trainable Whether the layer weights will be updated during training.
weights Initial weights for layer.

See Also

*Zero-bias autoencoders and the benefits of co-adapting features.*

Other activation layers: layer_activation_elu(), layer_activation_leaky_relu(), layer_activation_parametric_relu(), layer_activation_relu(), layer_activation_selu(), layer_activation_softmax(), layer_activation()
Usage

```r
layer_activity_regularization(
  object,
  l1 = 0,
  l2 = 0,
  input_shape = NULL,
  batch_input_shape = NULL,
  batch_size = NULL,
  dtype = NULL,
  name = NULL,
  trainable = NULL,
  weights = NULL
)
```

Arguments

- **object** Model or layer object
- **l1** L1 regularization factor (positive float).
- **l2** L2 regularization factor (positive float).
- **input_shape** Dimensionality of the input (integer) not including the samples axis. This argument is required when using this layer as the first layer in a model.
- **batch_input_shape** Shapes, including the batch size. For instance, `batch_input_shape=c(10, 32)` indicates that the expected input will be batches of 10 32-dimensional vectors. `batch_input_shape=list(NULL, 32)` indicates batches of an arbitrary number of 32-dimensional vectors.
- **batch_size** Fixed batch size for layer
- **dtype** The data type expected by the input, as a string (`float32`, `float64`, `int32`...)
- **name** An 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.
- **trainable** Whether the layer weights will be updated during training.
- **weights** Initial weights for layer.

Input shape

Arbitrary. Use the keyword argument `input_shape` (list 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

Other core layers: `layer_activation()`, `layer_attention()`, `layer_dense_features()`, `layer_dense()`, `layer_dropout()`, `layer_flatten()`, `layer_input()`, `layer_lambda()`, `layer_masking()`, `layer_permute()`, `layer_repeat_vector()`, `layer_reshape()`
**layer_add**  

*Layer that adds 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**

```python
layer_add(
    inputs,
    batch_size = NULL,
    dtype = NULL,
    name = NULL,
    trainable = NULL,
    weights = NULL
)
```

**Arguments**

- **inputs**: A list of input tensors (at least 2).
- **batch_size**: Fixed batch size for layer.
- **dtype**: The data type expected by the input, as a string (float32, float64, int32...)
- **name**: An 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.
- **trainable**: Whether the layer weights will be updated during training.
- **weights**: Initial weights for layer.

**Value**

A tensor, the sum of the inputs.

**See Also**

Other merge layers: `layer_average()`, `layer_concatenate()`, `layer_dot()`, `layer_maximum()`, `layer_minimum()`, `layer_multiply()`, `layer_subtract()`
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.

**Usage**

```r
layer_alpha_dropout(
    object,
    rate,
    noise_shape = NULL,
    seed = NULL,
    input_shape = NULL,
    batch_input_shape = NULL,
    batch_size = NULL,
    dtype = NULL,
    name = NULL,
    trainable = NULL,
    weights = NULL
)
```

**Arguments**

- **object**: Model or layer object.
- **rate**: float, drop probability (as with `layer_dropout()`). The multiplicative noise will have standard deviation `sqrt(rate / (1 - rate))`.
- **noise_shape**: Noise shape.
- **seed**: An integer to use as random seed.
- **input_shape**: Dimensionality of the input (integer) not including the samples axis. This argument is required when using this layer as the first layer in a model.
- **batch_input_shape**: Shapes, including the batch size. For instance, `batch_input_shape=c(10, 32)` indicates that the expected input will be batches of 10 32-dimensional vectors. `batch_input_shape=list(NULL, 32)` indicates batches of an arbitrary number of 32-dimensional vectors.
- **batch_size**: Fixed batch size for layer.
- **dtype**: The data type expected by the input, as a string (`float32`, `float64`, `int32`...).
- **name**: An 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.
- **trainable**: Whether the layer weights will be updated during training.
- **weights**: Initial weights for layer.
Details

Alpha Dropout fits well to Scaled Exponential Linear Units by randomly setting activations to the negative saturation value.

Input shape

Arbitrary. Use the keyword argument `input_shape` (list 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.

References

- Self-Normalizing Neural Networks

See Also

Other noise layers: `layer_gaussian_dropout()`, `layer_gaussian_noise()`

Description

Dot-product attention layer, a.k.a. Luong-style attention.

Usage

```r
layer_attention(
  inputs,
  use_scale = FALSE,
  causal = FALSE,
  batch_size = NULL,
  dtype = NULL,
  name = NULL,
  trainable = NULL,
  weights = NULL
)
```
layer_average

Arguments

- **inputs**: a list of inputs first should be the query tensor, the second the value tensor.
- **use_scale**: If True, will create a scalar variable to scale the attention scores.
- **causal**: 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.
- **batch_size**: Fixed batch size for layer.
- **dtype**: The data type expected by the input, as a string (float32, float64, int32...)
- **name**: An 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.
- **trainable**: Whether the layer weights will be updated during training.
- **weights**: Initial weights for layer.

See Also

Other core layers: layer_activation(), layer_activity_regularizer(), layer_dense_features(), layer_dense(), layer_dropout(), layer_flatten(), layer_input(), layer_lambda(), layer_masking(), layer_permute(), layer_repeat_vector(), layer_reshape()

**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**

```r
layer_average(  
  inputs,  
  batch_size = NULL,  
  dtype = NULL,  
  name = NULL,  
  trainable = NULL,  
  weights = NULL  
)
```

Arguments

- **inputs**: A list of input tensors (at least 2).
- **batch_size**: Fixed batch size for layer.
- **dtype**: The data type expected by the input, as a string (float32, float64, int32...).
layer_average_pooling_1d

An 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.

Whether the layer weights will be updated during training.

Initial weights for layer.

Value

A tensor, the average of the inputs.

See Also

Other merge layers: `layer_add()`, `layer_concatenate()`, `layer_dot()`, `layer_maximum()`, `layer_minimum()`, `layer_multiply()`, `layer_subtract()`

layer_average_pooling_1d

Average pooling for temporal data.

Description

Average pooling for temporal data.

Usage

```r
layer_average_pooling_1d(
  object,
  pool_size = 2L,
  strides = NULL,
  padding = "valid",
  data_format = "channels_last",
  batch_size = NULL,
  name = NULL,
  trainable = NULL,
  weights = NULL
)
```

Arguments

- **object**: Model or layer object
- **pool_size**: Integer, size of the average pooling windows.
- **strides**: Integer, or NULL. Factor by which to downscale. E.g. 2 will halve the input. If NULL, it will default to pool_size.
- **padding**: One of "valid" or "same" (case-insensitive).
- **data_format**: One of channels_last (default) or channels_first. The ordering of the dimensions in the inputs.
layer_average_pooling_2d

batch_size
Fixed batch size for layer

name
An 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.

trainable
Whether the layer weights will be updated during training.

weights
Initial weights for layer.

Input shape

3D tensor with shape: (batch_size, steps, features).

Output shape

3D tensor with shape: (batch_size, downsampled_steps, features).

See Also

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()

layer_average_pooling_2d

Average pooling operation for spatial data.

Description

Average pooling operation for spatial data.

Usage

layer_average_pooling_2d(
  object,
  pool_size = c(2L, 2L),
  strides = NULL,
  padding = "valid",
  data_format = NULL,
  batch_size = NULL,
  name = NULL,
  trainable = NULL,
  weights = NULL
)
**layer_average_pooling_2d**

**Arguments**

- **object**
  Model or layer object

- **pool_size**
  Integer or list of 2 integers, factors by which to downscale (vertical, horizontal). (2, 2) will halve the input in both spatial dimension. If only one integer is specified, the same window length will be used for both dimensions.

- **strides**
  Integer, list of 2 integers, or NULL. Strides values. If NULL, it will default to pool_size.

- **padding**
  One of "valid" or "same" (case-insensitive).

- **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, height, width, channels) while channels_first corresponds to inputs with shape (batch, channels, height, width). If data_format value found in your Keras config file at ~/.keras/keras.json. If you never set it, then it will be "channels_last".

- **batch_size**
  Fixed batch size for layer

- **name**
  An 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.

- **trainable**
  Whether the layer weights will be updated during training.

- **weights**
  Initial weights for layer.

**Input shape**

- If data_format='channels_last': 4D tensor with shape: (batch_size, rows, cols, channels)
- If data_format='channels_first': 4D tensor with shape: (batch_size, channels, rows, cols)

**Output shape**

- If data_format='channels_last': 4D tensor with shape: (batch_size, pooled_rows, pooled_cols, channels)
- If data_format='channels_first': 4D tensor with shape: (batch_size, channels, pooled_rows, pooled_cols)

**See Also**

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()
layer_average_pooling_3d

Average pooling operation for 3D data (spatial or spatio-temporal).

Description

Average pooling operation for 3D data (spatial or spatio-temporal).

Usage

layer_average_pooling_3d(
  object,
  pool_size = c(2L, 2L, 2L),
  strides = NULL,
  padding = "valid",
  data_format = NULL,
  batch_size = NULL,
  name = NULL,
  trainable = NULL,
  weights = NULL
)

Arguments

object Model or layer object

pool_size list of 3 integers, factors by which to downscale (dim1, dim2, dim3). (2, 2, 2) will halve the size of the 3D input in each dimension.

strides list of 3 integers, or NULL. Strides values.

padding One of "valid" or "same" (case-insensitive).

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, spatial_dim1, spatial_dim2, spatial_dim3, channels) while channels_first corresponds to inputs with shape (batch, channels, spatial_dim1, spatial_dim2, spatial_dim3). 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".

batch_size Fixed batch size for layer

name An 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.

trainable Whether the layer weights will be updated during training.

weights Initial weights for layer.
layer_batch_normalization

Input shape

- If data_format='channels_last': 5D tensor with shape: (batch_size, spatial_dim1, spatial_dim2, spatial_dim3, channels)
- If data_format='channels_first': 5D tensor with shape: (batch_size, channels, spatial_dim1, spatial_dim2, spatial_dim3)

Output shape

- If data_format='channels_last': 5D tensor with shape: (batch_size, pooled_dim1, pooled_dim2, pooled_dim3, channels)
- If data_format='channels_first': 5D tensor with shape: (batch_size, channels, pooled_dim1, pooled_dim2, pooled_dim3)

See Also

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()

layer_batch_normalization

Batch normalization layer (Ioffe and Szegedy, 2014).

Description

Normalize the activations of the previous layer at each batch, i.e. applies a transformation that maintains the mean activation close to 0 and the activation standard deviation close to 1.

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,
    renorm = FALSE,
)
renorm_clipping = NULL,
renorm_momentum = 0.99,
fused = NULL,
virtual_batch_size = NULL,
adjustment = NULL,
input_shape = NULL,
batch_input_shape = NULL,
batch_size = NULL,
dtype = NULL,
name = NULL,
trainable = NULL,
weights = NULL
)

Arguments

object  Model or layer object
axis   Integer, the axis that should be normalized (typically the features axis). For instance, after a Conv2D layer with data_format="channels_first", set axis=1 in BatchNormalization.
momentum Momentum for the moving mean and the moving variance.
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 (also e.g. nn.relu), 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.
renorm   Whether to use Batch Renormalization (https://arxiv.org/abs/1702.03275). This adds extra variables during training. The inference is the same for either value of this parameter.
layer_batch_normalization  221

renorm_clipping
A named list or dictionary that may map keys rmax, rmin, dmax to scalar Tensors
used to clip the renorm correction. The correction (r, d) is used as corrected_value
= normalized_value * r + d, with r clipped to [rmin, rmax], and d to [-dmax, dmax].
Missing rmax, rmin, dmax are set to Inf, 0, Inf, respectively.

renorm_momentum
Momentum used to update the moving means and standard deviations with
renorm. Unlike momentum, this affects training and should be neither too small
(which would add noise) nor too large (which would give stale estimates). Note
that momentum is still applied to get the means and variances for inference.

fused
TRUE, use a faster, fused implementation, or raise a ValueError if the fused im-
plementation cannot be used. If NULL, use the faster implementation if possible.
If FALSE, do not use the fused implementation.

virtual_batch_size
An integer. By default, virtual_batch_size is NULL, which means batch normal-
ization is performed across the whole batch. When virtual_batch_size is not
NULL, instead perform "Ghost Batch Normalization", which creates virtual sub-
batches which are each normalized separately (with shared gamma, beta, and
moving statistics). Must divide the actual batch size during execution.

adjustment
A function taking the Tensor containing the (dynamic) shape of the input ten-
sor and returning a pair (scale, bias) to apply to the normalized values (be-
fore gamma and beta), only during training. For example, if axis==-1, adjustment
<-function(shape) { tuple(tf$random$uniform(shape[-1:NULL,style = "python"],0.93,1.07)
= "python"),-0.1,0.1)) } will scale the normalized value by up to 7% up or
down, then shift the result by up to 0.1 (with independent scaling and bias for
each feature but shared across all examples), and finally apply gamma and/or
beta. If NULL, no adjustment is applied. Cannot be specified if virtual_batch_size
is specified.

input_shape
Dimensionality of the input (integer) not including the samples axis. This argu-
ment is required when using this layer as the first layer in a model.

batch_input_shape
Shapes, including the batch size. For instance, batch_input_shape=c(10,32)
indicates that the expected input will be batches of 10 32-dimensional vectors.
batch_input_shape=list(NULL,32) indicates batches of an arbitrary number
of 32-dimensional vectors.

batch_size
Fixed batch size for layer

dtype
The data type expected by the input, as a string (float32, float64, int32...)

name
An 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.

trainable
Whether the layer weights will be updated during training.

weights
Initial weights for layer.

Input shape
Arbitrary. Use the keyword argument input_shape (list 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.

References
- Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift

layer_concatenate
Layer that concatenates a list of inputs.

Description
It takes as input a list of tensors, all of the same shape expect for the concatenation axis, and returns a single tensor, the concatenation of all inputs.

Usage
layer_concatenate(
    inputs,
    axis = -1,
    batch_size = NULL,
    dtype = NULL,
    name = NULL,
    trainable = NULL,
    weights = NULL
)

Arguments
- inputs: A list of input tensors (at least 2).
- axis: Concatenation axis.
- batch_size: Fixed batch size for layer.
- dtype: The data type expected by the input, as a string (float32, float64, int32...)
- name: An 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.
- trainable: Whether the layer weights will be updated during training.
- weights: Initial weights for layer.

Value
A tensor, the concatenation of the inputs alongside axis axis.

See Also
Other merge layers: layer_add(), layer_average(), layer_dot(), layer_maximum(), layer_minimum(), layer_multiply(), layer_subtract()
**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. When using this layer as the first layer in a model, provide an `input_shape` argument (list of integers or NULL, e.g. (10, 128) for sequences of 10 vectors of 128-dimensional vectors, or (NULL, 128) for variable-length sequences of 128-dimensional vectors.

**Usage**

```r
layer_conv_1d(
  object,
  filters,
  kernel_size,
  strides = 1L,
  padding = "valid",
  data_format = "channels_last",
  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,
  input_shape = NULL,
  batch_input_shape = NULL,
  batch_size = NULL,
  dtype = NULL,
  name = NULL,
  trainable = NULL,
  weights = NULL
)
```

**Arguments**

- **object** Model or layer object
- **filters** Integer, the dimensionality of the output space (i.e. the number of output filters in the convolution).
**kernel_size**
An integer or list of a single integer, specifying the length of the 1D convolution window.

**strides**
An integer or list of a single integer, specifying the stride length of the convolution. Specifying any stride value != 1 is incompatible with specifying any dilation_rate value != 1.

**padding**
One of "valid", "causal" or "same" (case-insensitive). "valid" means "no padding". "same" results in padding the input such that the output has the same length as the original input. "causal" results in causal (dilated) convolutions, e.g. output[t] does not depend on 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, section 2.1.

**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, length, channels) (default format for temporal data in Keras) while "channels_first" corresponds to inputs with shape (batch, channels, length).

**dilation_rate**
an integer or list of a single integer, specifying the dilation rate to use for dilated convolution. Currently, specifying any dilation_rate value != 1 is incompatible with specifying any strides value != 1.

**groups**
A positive integer 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 to use. If you don’t specify anything, no activation is applied (i.e. "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 matrix.

**bias_constraint**
Constraint function applied to the bias vector.

**input_shape**
Dimensionality of the input (integer) not including the samples axis. This argument is required when using this layer as the first layer in a model.

**batch_input_shape**
Shapes, including the batch size. For instance, batch_input_shape=c(10, 32) indicates that the expected input will be batches of 10 32-dimensional vectors.
layer_conv_1d_transpose

batch_input_shape=list(NULL,32) indicates batches of an arbitrary number of 32-dimensional vectors.

batch_size Fixed batch size for layer
dtype The data type expected by the input, as a string (float32, float64, int32...)
name An 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.
trainable Whether the layer weights will be updated during training.
weights Initial weights for layer.

Input shape
3D tensor with shape: (batch_size, steps, input_dim)

Output shape
3D tensor with shape: (batch_size, new_steps, filters) steps value might have changed due to padding or strides.

See Also
Other convolutional layers: layer_conv_1d_transpose(), layer_conv_2d_transpose(), layer_conv_2d(), layer_conv_3d_transpose(), layer_conv_3d(), layer_conv_lstm_2d(), layer_cropping_1d(), layer_cropping_2d(), layer_cropping_3d(), layer_depthwise_conv_2d(), layer_separable_conv_1d(), layer_separable_conv_2d(), layer_upsampling_1d(), layer_upsampling_2d(), layer_upsampling_3d(), layer_zero_padding_1d(), layer_zero_padding_2d(), layer_zero_padding_3d()

layer_conv_1d_transpose

Transposed 1D convolution layer (sometimes called Deconvolution).

Description
The need for transposed convolutions generally arises 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. When using this layer as the first layer in a model, provide the keyword argument input_shape (tuple of integers, does not include the sample axis), e.g. input_shape=(128, 3) for data with 128 time steps and 3 channels.

Usage
layer_conv_1d_transpose(
    object,
    filters,
    kernel_size,
    strides = 1,
padding = "valid",
output_padding = NULL,
data_format = NULL,
dilation_rate = 1,
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,
input_shape = NULL,
batch_input_shape = NULL,
batch_size = NULL,
dtype = NULL,
name = NULL,
trainable = NULL,
weights = NULL
)

Arguments

object    Model or layer object
filters   Integer, the dimensionality of the output space (i.e. the number of output filters in the convolution).
kernel_size   An integer or list of a single integer, specifying the length of the 1D convolution window.
strides   An integer or list of a single integer, specifying the stride length of the convolution. Specifying any stride value != 1 is incompatible with specifying any dilation_rate value != 1.
padding    one of "valid" or "same" (case-insensitive).
output_padding An integer specifying the amount of padding along the time dimension of the output tensor. The amount of output padding must be lower than the stride. If set to NULL (default), the output shape is inferred.
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, length, channels) (default format for temporal data in Keras) while "channels_first" corresponds to inputs with shape (batch, channels, length).
dilation_rate an integer or list of a single integer, specifying the dilation rate to use for dilated convolution. Currently, specifying any dilation_rate value != 1 is incompatible with specifying any strides value != 1.
activation Activation function to use. If you don’t specify anything, no activation is applied (i.e. "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 matrix.
bias_constraint          Constraint function applied to the bias vector.
input_shape              Dimensionality of the input (integer) not including the samples axis. This argument is required when using this layer as the first layer in a model.
batch_input_shape        Shapes, including the batch size. For instance, batch_input_shape=c(10, 32) indicates that the expected input will be batches of 10 32-dimensional vectors. batch_input_shape=list(NULL, 32) indicates batches of an arbitrary number of 32-dimensional vectors.
batch_size               Fixed batch size for layer
dtype                    The data type expected by the input, as a string (float32, float64, int32...)
name                     An 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.
trainable                Whether the layer weights will be updated during training.
weights                  Initial weights for layer.

Input shape

3D tensor with shape: (batch, steps, channels)

Output shape

3D tensor with shape: (batch, new_steps, filters) If output_padding is specified:

\[
new\_steps = ((timesteps - 1) \times strides + kernel\_size - 2 \times padding + output\_padding)
\]

References

- A guide to convolution arithmetic for deep learning
layer_conv_2d

2D convolution layer (e.g. spatial convolution over images).

Description

This layer creates a convolution kernel that is convolved with the layer input 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. When using this layer as the first layer in a model, provide the keyword argument input_shape (list of integers, does not include the sample axis), e.g. input_shape=c(128,128,3) for 128x128 RGB pictures in data_format="channels_last".

Usage

layer_conv_2d(
  object,
  filters,
  kernel_size,
  strides = c(1L, 1L),
  padding = "valid",
  data_format = NULL,
  dilation_rate = c(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,
  input_shape = NULL,
  batch_input_shape = NULL,
  batch_size = NULL,
  dtype = NULL,
  name = NULL,
  trainable = NULL,
  weights = NULL
)
layer_conv_2d

Arguments

- **object**: Model or layer object
- **filters**: Integer, the dimensionality of the output space (i.e. the number of output filters in the convolution).
- **kernel_size**: An integer or list of 2 integers, specifying the width and height of the 2D convolution window. Can be a single integer to specify the same value for all spatial dimensions.
- **strides**: An integer or list of 2 integers, specifying the strides of the convolution along the width and height. Can be a single integer to specify the same value for all spatial dimensions. Specifying any stride value != 1 is incompatible with specifying any dilation_rate value != 1.
- **padding**: one of "valid" or "same" (case-insensitive). Note that "same" is slightly inconsistent across backends with strides != 1, as described [here](#).
- **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, 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**: an integer or list of 2 integers, specifying the dilation rate to use for dilated convolution. Can be a single integer to specify the same value for all spatial dimensions. Currently, specifying any dilation_rate value != 1 is incompatible with specifying any stride value != 1.
- **groups**: A positive integer 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 to use. If you don’t specify anything, no activation is applied (i.e. "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 matrix.
- **bias_constraint**: Constraint function applied to the bias vector.
layer_conv_2d_transpose

input_shape  Dimensionality of the input (integer) not including the samples axis. This argument is required when using this layer as the first layer in a model.

batch_input_shape  Shapes, including the batch size. For instance, batch_input_shape=c(10, 32) indicates that the expected input will be batches of 10 32-dimensional vectors. batch_input_shape=list(NULL, 32) indicates batches of an arbitrary number of 32-dimensional vectors.

batch_size  Fixed batch size for layer

dtype  The data type expected by the input, as a string (float32, float64, int32...)

name  An 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.

trainable  Whether the layer weights will be updated during training.

weights  Initial weights for layer.

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

4D tensor with shape: (samples, filters, new_rows, new_cols) if data_format='channels_first' or 4D tensor with shape: (samples, new_rows, new_cols, filters) if data_format='channels_last'. rows and cols values might have changed due to padding.

See Also

Other convolutional layers: layer_conv_1d_transpose(), layer_conv_1d(), layer_conv_2d_transpose(), layer_conv_3d_transpose(), layer_lstm_2d(), layer_separable_conv_2d(), layer_cropping_2d(), layer_encoding_2d(), layer_depthwise_conv_2d(), layer_separable_conv_1d(), layer_encoding_1d(), layer_depthwise_conv_1d(), layer_zero_padding_1d(), layer_zero_padding_2d(), layer_zero_padding_3d()
layer_conv_2d_transpose

Usage

layer_conv_2d_transpose(
    object,
    filters,
    kernel_size,
    strides = c(1, 1),
    padding = "valid",
    output_padding = NULL,
    data_format = NULL,
    dilation_rate = c(1, 1),
    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,
    input_shape = NULL,
    batch_input_shape = NULL,
    batch_size = NULL,
    dtype = NULL,
    name = NULL,
    trainable = NULL,
    weights = NULL
)

Arguments

object Model or layer object
filters Integer, the dimensionality of the output space (i.e. the number of output filters in the convolution).
kernel_size An integer or list of 2 integers, specifying the width and height of the 2D convolution window. Can be a single integer to specify the same value for all spatial dimensions.
strides An integer or list of 2 integers, specifying the strides of the convolution along the width and height. Can be a single integer to specify the same value for all spatial dimensions. Specifying any stride value != 1 is incompatible with specifying any dilation_rate value != 1.
padding one of "valid" or "same" (case-insensitive).
output_padding An integer or list of 2 integers, 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, one of channels_last (default) 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  Dialation rate.

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 matrix.

bias_constraint  Constraint function applied to the bias vector.

input_shape  Dimensionality of the input (integer) not including the samples axis. This argument is required when using this layer as the first layer in a model.

batch_input_shape  Shapes, including the batch size. For instance, batch_input_shape=c(10,32) indicates that the expected input will be batches of 10 32-dimensional vectors. batch_input_shape=list(NULL,32) indicates batches of an arbitrary number of 32-dimensional vectors.

batch_size  Fixed batch size for layer

dtype  The data type expected by the input, as a string (float32, float64, int32...)

name  An 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.

trainable  Whether the layer weights will be updated during training.

weights  Initial weights for layer.

**Input shape**

4D tensor with shape: (batch, channels, rows, cols) if data_format='channels_first' or 4D tensor with shape: (batch, rows, cols, channels) if data_format='channels_last'.
**Output shape**

4D tensor with shape: (batch, filters, new_rows, new_cols) if data_format='channels_first' or 4D tensor with shape: (batch, new_rows, new_cols, filters) if data_format='channels_last'. rows and cols values might have changed due to padding.

**References**

- A guide to convolution arithmetic for deep learning

**See Also**

Other convolutional layers: `layer_conv_1d_transpose()`, `layer_conv_1d()`, `layer_conv_2d()`, `layer_conv_3d_transpose()`, `layer_conv_3d()`, `layer_conv_lstm_2d()`, `layer_cropping_1d()`, `layer_cropping_2d()`, `layer_cropping_3d()`, `layer_depthwise_conv_2d()`, `layer_separable_conv_1d()`, `layer_separable_conv_2d()`, `layer_upsampling_1d()`, `layer_upsampling_2d()`, `layer_upsampling_3d()`, `layer_zero_padding_1d()`, `layer_zero_padding_2d()`, `layer_zero_padding_3d()`

---

**Description**

This layer creates a convolution kernel that is convolved with the layer input 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. When using this layer as the first layer in a model, provide the keyword argument input_shape (list of integers, does not include the sample axis), e.g. input_shape=c(128L, 128L, 128L, 3L) for 128x128x128 volumes with a single channel, in data_format="channels_last".

**Usage**

```r
layer_conv_3d(
  object,
  filters,
  kernel_size,
  strides = c(1L, 1L, 1L),
  padding = "valid",
  data_format = NULL,
  dilation_rate = c(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,
)```
layer_conv_3d

    kernel_constraint = NULL,
    bias_constraint = NULL,
    input_shape = NULL,
    batch_input_shape = NULL,
    batch_size = NULL,
    dtype = NULL,
    name = NULL,
    trainable = NULL,
    weights = NULL

Arguments

object        Model or layer object
filters       Integer, the dimensionality of the output space (i.e. the number of output filters in the convolution).
kernel_size   An integer or list of 3 integers, specifying the depth, height, and width of the 3D convolution window. Can be a single integer to specify the same value for all spatial dimensions.
strides       An integer or list of 3 integers, specifying the strides of the convolution along each spatial dimension. Can be a single integer to specify the same value for all spatial dimensions. Specifying any stride value != 1 is incompatible with specifying any dilation_rate value != 1.
padding       one of "valid" or "same" (case-insensitive).
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, spatial_dim1, spatial_dim2, spatial_dim3, channels) while channels_first corresponds to inputs with shape (batch, channels, spatial_dim1, spatial_dim2, spatial_dim3). 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 an integer or list of 3 integers, specifying the dilation rate to use for dilated convolution. Can be a single integer to specify the same value for all spatial dimensions. Currently, specifying any dilation_rate value != 1 is incompatible with specifying any stride value != 1.
groups        A positive integer 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 to use. If you don’t specify anything, no activation is applied (i.e. "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.
layer_conv_3d

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 matrix.

bias_constraint
Constraint function applied to the bias vector.

input_shape
Dimensionality of the input (integer) not including the samples axis. This argument is required when using this layer as the first layer in a model.

batch_input_shape
Shapes, including the batch size. For instance, batch_input_shape=c(10, 32) indicates that the expected input will be batches of 10 32-dimensional vectors. batch_input_shape=list(NULL, 32) indicates batches of an arbitrary number of 32-dimensional vectors.

batch_size
Fixed batch size for layer

dtype
The data type expected by the input, as a string (float32, float64, int32...)

name
An 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.

trainable
Whether the layer weights will be updated during training.

weights
Initial weights for layer.

Input shape
5D tensor with shape: (samples, channels, conv_dim1, conv_dim2, conv_dim3) if data_format='channels_first' or 5D tensor with shape: (samples, conv_dim1, conv_dim2, conv_dim3, channels) if data_format='channels_last'.

Output shape
5D tensor with shape: (samples, filters, new_conv_dim1, new_conv_dim2, new_conv_dim3) if data_format='channels_first' or 5D tensor with shape: (samples, new_conv_dim1, new_conv_dim2, new_conv_dim3, filters) if data format='channels_last'. new_conv_dim1, new_conv_dim2 and new_conv_dim3 values might have changed due to padding.

See Also
Other convolutional layers: layer_conv_1d_transpose(), layer_conv_1d(), layer_conv_2d_transpose(), layer_conv_2d(), layer_conv_3d_transpose(), layer_conv_lstm_2d(), layer_cropping_1d(), layer_cropping_2d(), layer_cropping_3d(), layer_depthwise_conv_2d(), layer_separable_conv_1d(), layer_separable_conv_2d(), layer_upsampling_1d(), layer_upsampling_2d(), layer_upsampling_3d(), layer_zero_padding_1d(), layer_zero_padding_2d(), layer_zero_padding_3d()
Transposed 3D convolution layer (sometimes called Deconvolution).

Description

The need for transposed convolutions generally arises 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

```r
layer_conv_3d_transpose(
  object,
  filters,
  kernel_size,
  strides = c(1, 1, 1),
  padding = "valid",
  output_padding = NULL,
  data_format = NULL,
  dilation_rate = c(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,
  input_shape = NULL,
  batch_input_shape = NULL,
  batch_size = NULL,
  dtype = NULL,
  name = NULL,
  trainable = NULL,
  weights = NULL
)
```

Arguments

- **object**: Model or layer object
- **filters**: Integer, the dimensionality of the output space (i.e. the number of output filters in the convolution).
layer_conv_3d_transpose

kernel_size
An integer or list of 3 integers, specifying the depth, height, and width of the 3D convolution window. Can be a single integer to specify the same value for all spatial dimensions.

strides
An integer or list of 3 integers, specifying the strides of the convolution along the depth, height and width. Can be a single integer to specify the same value for all spatial dimensions. Specifying any stride value != 1 is incompatible with specifying any dilation_rate value != 1.

padding
one of "valid" or "same" (case-insensitive).

output_padding
An integer or list of 3 integers, specifying the amount of padding along the depth, 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, one of channels_last (default) or channels_first. The ordering of the dimensions in the inputs. channels_last corresponds to inputs with shape (batch, depth, height, width, channels) while channels_first corresponds to inputs with shape (batch, channels, depth, 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
An integer or vector of 3 integers, specifying the dilation rate to use for dilated convolution. Can be a single integer to specify the same value for all spatial dimensions.

activation
Activation function to use. If you don’t specify anything, no activation is applied (i.e. "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 matrix.

bias_constraint
Constraint function applied to the bias vector.

input_shape
Dimensionality of the input (integer) not including the samples axis. This argument is required when using this layer as the first layer in a model.

batch_input_shape
Shapes, including the batch size. For instance, batch_input_shape=c(10, 32) indicates that the expected input will be batches of 10 32-dimensional vectors. batch_input_shape=list(NULL, 32) indicates batches of an arbitrary number of 32-dimensional vectors.
layer_conv_lstm_2d

batch_size  Fixed batch size for layer

dtype       The data type expected by the input, as a string (float32, float64, int32...)

name        An 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.

trainable   Whether the layer weights will be updated during training.

weights     Initial weights for layer.

Details

When using this layer as the first layer in a model, provide the keyword argument input_shape
(list of integers, does not include the sample axis), e.g. input_shape = list(128,128,128,3) for
a 128x128x128 volume with 3 channels if data_format="channels_last".

References

• A guide to convolution arithmetic for deep learning

See Also

Other convolutional layers: layer_conv_1d_transpose(), layer_conv_1d(), layer_conv_2d_transpose()
layer_conv_2d(), layer_conv_3d(), layer_conv_lstm_2d(), layer_cropping_1d(), layer_cropping_2d()
layer_cropping_3d(), layer_depthwise_conv_2d(), layer_separable_conv_1d(), layer_separable_conv_2d()
layer_upsampling_1d(), layer_upsampling_2d(), layer_upsampling_3d(), layer_zero_padding_1d()
layer_zero_padding_2d(), layer_zero_padding_3d()

layer_conv_lstm_2d  Convolutional LSTM.

Description

It is 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 = c(1L, 1L),
    padding = "valid",
    data_format = NULL,
    dilation_rate = c(1L, 1L),
    activation = "tanh",
    recurrent_activation = "hard_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,
return_sequences = FALSE,
return_state = FALSE,
going_backwards = FALSE,
stateful = FALSE,
dropout = 0,
recurrent_dropout = 0,
batch_size = NULL,
name = NULL,
trainable = NULL,
weights = NULL,
input_shape = NULL
)

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>object</td>
<td>Model or layer object</td>
</tr>
<tr>
<td>filters</td>
<td>Integer, the dimensionality of the output space (i.e. the number of output filters in the convolution).</td>
</tr>
<tr>
<td>kernel_size</td>
<td>An integer or list of n integers, specifying the dimensions of the convolution window.</td>
</tr>
<tr>
<td>strides</td>
<td>An integer or list of n integers, specifying the strides of the convolution. Specifying any stride value != 1 is incompatible with specifying any dilation_rate value != 1.</td>
</tr>
<tr>
<td>padding</td>
<td>One of &quot;valid&quot; or &quot;same&quot; (case-insensitive).</td>
</tr>
<tr>
<td>data_format</td>
<td>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, time, ..., channels) while channels_first corresponds to inputs with shape (batch, time, channels, ...). 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 &quot;channels_last&quot;.</td>
</tr>
<tr>
<td>dilation_rate</td>
<td>An integer or list of n integers, specifying the dilation rate to use for dilated convolution. Currently, specifying any dilation_rate value != 1 is incompatible with specifying any strides value != 1.</td>
</tr>
<tr>
<td>activation</td>
<td>Activation function to use. If you don’t specify anything, no activation is applied (ie. &quot;linear&quot; activation: ( a(x) = x )).</td>
</tr>
</tbody>
</table>
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 trans-
    formation 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.
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 the output of the layer (its "activation").
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.
return_sequences
    Boolean. Whether to return the last output in the output sequence, or the full
    sequence.
return_state
    Boolean. Whether to return the last state in addition to the output.
go_backwards
    Boolean (default FALSE). If TRUE, rocess the input sequence backwards.
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.
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.
batch_size
    Fixed batch size for layer
name
    An 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.
layer_cropping_1d

trainable Whether the layer weights will be updated during training.
weights Initial weights for layer.
input_shape Dimensionality of the input (integer) not including the samples axis. This argument is required when using this layer as the first layer in a model.

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)

References

- Convolutional LSTM Network: A Machine Learning Approach for Precipitation Nowcasting
  The current implementation does not include the feedback loop on the cells output

See Also

Other convolutional layers: layer_conv_1d_transpose(), layer_conv_1d(), layer_conv_2d_transpose(),
layer_conv_2d(), layer_conv_3d_transpose(), layer_conv_3d(), layer_cropping_1d(), layer_cropping_2d(),
layer_cropping_3d(), layer_depthwise_conv_2d(), layer_separable_conv_1d(), layer_separable_conv_2d(),
layer_upsampling_1d(), layer_upsampling_2d(), layer_upsampling_3d(), layer_zero_padding_1d(),
layer_zero_padding_2d(), layer_zero_padding_3d()
Arguments

- **object**: Model or layer object
- **cropping**: int or list of int (length 2) How many units should be trimmed off at the beginning and end of the cropping dimension (axis 1). If a single int is provided, the same value will be used for both.
- **batch_size**: Fixed batch size for layer
- **name**: An 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.
- **trainable**: Whether the layer weights will be updated during training.
- **weights**: Initial weights for layer.

Input shape

- 3D tensor with shape (batch, axis_to_crop, features)

Output shape

- 3D tensor with shape (batch, cropped_axis, features)

See Also

- Other convolutional layers: `layer_conv_1d_transpose()`, `layer_conv_1d()`, `layer_conv_2d_transpose()`, `layer_conv_2d()`, `layer_conv_3d_transpose()`, `layer_conv_3d()`, `layer_conv_lstm_2d()`, `layer_cropping_2d()`, `layer_cropping_3d()`, `layer_depthwise_conv_2d()`, `layer_separable_conv_1d()`, `layer_separable_conv_2d()`, `layer_upsampling_1d()`, `layer_upsampling_2d()`, `layer_upsampling_3d()`, `layer_zero_padding_1d()`, `layer_zero_padding_2d()`, `layer_zero_padding_3d()`

---

**layer_cropping_2d**  
*Cropping layer for 2D input (e.g. picture).*

**Description**

It crops along spatial dimensions, i.e. width and height.

**Usage**

```r
layer_cropping_2d(
  object,
  cropping = list(c(0L, 0L), c(0L, 0L)),
  data_format = NULL,
  batch_size = NULL,
  name = NULL,
  trainable = NULL,
  weights = NULL
)
```
layer_cropping_2d

Arguments

- **object**: Model or layer object
- **cropping**: int, or list of 2 ints, or list of 2 lists of 2 ints.
  - If int: the same symmetric cropping is applied to width and height.
  - 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, 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".

- **batch_size**: Fixed batch size for layer
- **name**: An 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.
- **trainable**: Whether the layer weights will be updated during training.
- **weights**: Initial weights for layer.

Input shape

4D tensor with shape:

- If `data_format` is "channels_last": (batch, rows, cols, channels)
- If `data_format` is "channels_first": (batch, channels, rows, cols)

Output shape

4D tensor with shape:

- If `data_format` is "channels_last": (batch, cropped_rows, cropped_cols, channels)
- If `data_format` is "channels_first": (batch, channels, cropped_rows, cropped_cols)

See Also

Other convolutional layers: layer_conv_1d_transpose(), layer_conv_1d(), layer_conv_2d_transpose(), layer_conv_2d(), layer_conv_3d_transpose(), layer_conv_3d(), layer_conv_lstm_2d(), layer_cropping_1d(), layer_cropping_3d(), layer_depthwise_conv_2d(), layer_separable_conv_1d(), layer_separable_conv_2d(), layer_upsampling_1d(), layer_upsampling_2d(), layer_upsampling_3d(), layer_zero_padding_1d(), layer_zero_padding_2d(), layer_zero_padding_3d()
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

```r
layer_cropping_3d(
  object,
  cropping = list(c(1L, 1L), c(1L, 1L), c(1L, 1L)),
  data_format = NULL,
  batch_size = NULL,
  name = NULL,
  trainable = NULL,
  weights = NULL
)
```

Arguments

- **object**: Model or layer object
- **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 two different symmetric cropping values for depth, height, and width: (symmetric_dim1_crop, symmetric_dim2_crop, symmetric_dim3_crop).
  - If list of 3 list of 2 ints: interpreted as ((left_dim1_crop, right_dim1_crop), (left_dim2_crop, right_dim2_crop), (left_dim3_crop, right_dim3_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, spatial_dim1, spatial_dim2, spatial_dim3, channels) while channels_first corresponds to inputs with shape (batch, channels, spatial_dim1, spatial_dim2, spatial_dim3). 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".
- **batch_size**: Fixed batch size for layer
- **name**: An 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.
- **trainable**: Whether the layer weights will be updated during training.
- **weights**: Initial weights for layer.

Input shape

5D tensor with shape:

- If data_format is "channels_last": (batch, first_axis_to_crop, second_axis_to_crop, third_axis_to_crop, depth)
- If data_format is "channels_first": (batch, depth, first_axis_to_crop, second_axis_to_crop, third_axis_to_crop)
**layer_cudnn_gru**

**Output shape**

5D tensor with shape:
- If `data_format` is "channels_last": (batch, first_cropped_axis, second_cropped_axis, third_cropped_axis, depth)
- If `data_format` is "channels_first": (batch, depth, first_cropped_axis, second_cropped_axis, third_cropped_axis)

**See Also**

Other convolutional layers: `layer_conv_1d_transpose()`, `layer_conv_1d()`, `layer_conv_2d_transpose()`, `layer_conv_2d()`, `layer_conv_3d_transpose()`, `layer_conv_3d()`, `layer_conv_lstm_2d()`, `layer_cropping_1d()`, `layer_cropping_2d()`, `layer_depthwise_conv_2d()`, `layer_separable_conv_1d()`, `layer_separable_conv_2d()`, `layer_upsampling_1d()`, `layer_upsampling_2d()`, `layer_upsampling_3d()`, `layer_zero_padding_1d()`, `layer_zero_padding_2d()`, `layer_zero_padding_3d()`

---

**Description**

Can only be run on GPU, with the TensorFlow backend.

**Usage**

```r
layer_cudnn_gru(
  object,
  units,
  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,
  return_sequences = FALSE,
  return_state = FALSE,
  stateful = FALSE,
  input_shape = NULL,
  batch_input_shape = NULL,
  batch_size = NULL,
  dtype = NULL,
  name = NULL,
  trainable = NULL,
  weights = NULL
)
```
Arguments

- **object**
  - Model or layer object

- **units**
  - Positive integer, dimensionality of the output space.

- **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.

- **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 the output of the layer (its "activation").

- **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.

- **return_sequences**
  - Boolean. 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.

- **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.

- **input_shape**
  - Dimensionality of the input (integer) not including the samples axis. This argument is required when using this layer as the first layer in a model.

- **batch_input_shape**
  - Shapes, including the batch size. For instance, batch_input_shape=c(10,32) indicates that the expected input will be batches of 10 32-dimensional vectors. batch_input_shape=list(NULL,32) indicates batches of an arbitrary number of 32-dimensional vectors.

- **batch_size**
  - Fixed batch size for layer

- **dtype**
  - The data type expected by the input, as a string (float32, float64, int32...)

- **name**
  - An 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.

- **trainable**
  - Whether the layer weights will be updated during training.

- **weights**
  - Initial weights for layer.
layer_cudnn_lstm

References

- On the Properties of Neural Machine Translation: Encoder-Decoder Approaches
- Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling
- A Theoretically Grounded Application of Dropout in Recurrent Neural Networks

See Also

Other recurrent layers: layer_cudnn_lstm(), layer_gru(), layer_lstm(), layer_simple_rnn()

---

layer_cudnn_lstm  Fast  LSTM implementation backed by
Rhrefhttps://developer.nvidia.com/cudnnCuDNN.

Description

Can only be run on GPU, with the TensorFlow backend.

Usage

layer_cudnn_lstm(
    object,
    units,
    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,
    return_sequences = FALSE,
    return_state = FALSE,
    stateful = FALSE,
    input_shape = NULL,
    batch_input_shape = NULL,
    batch_size = NULL,
    dtype = NULL,
    name = NULL,
    trainable = NULL,
    weights = NULL
)
Arguments

object Model or layer object
units Positive integer, dimensionality of the output space.
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. 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.
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 the output of the layer (its "activation").
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.
return_sequences Boolean. 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.
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.
input_shape Dimensionality of the input (integer) not including the samples axis. This argument is required when using this layer as the first layer in a model.
batch_input_shape Shapes, including the batch size. For instance, batch_input_shape=c(10, 32) indicates that the expected input will be batches of 10 32-dimensional vectors. batch_input_shape=list(NULL, 32) indicates batches of an arbitrary number of 32-dimensional vectors.
batch_size Fixed batch size for layer
layer_dense

<table>
<thead>
<tr>
<th>dtype</th>
<th>The data type expected by the input, as a string (float32, float64, int32...)</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>An 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.</td>
</tr>
<tr>
<td>trainable</td>
<td>Whether the layer weights will be updated during training.</td>
</tr>
<tr>
<td>weights</td>
<td>Initial weights for layer.</td>
</tr>
</tbody>
</table>

References

- Long short-term memory (original 1997 paper)
- Supervised sequence labeling with recurrent neural networks
- A Theoretically Grounded Application of Dropout in Recurrent Neural Networks

See Also

Other recurrent layers: layer_cudnn_gru(), layer_gru(), layer_lstm(), layer_simple_rnn()

layer_dense  

Add a densely-connected NN layer to an output

Description

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). Note: if the input to the layer has a rank greater than 2, then it is flattened prior to the initial dot product with kernel.

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,
    input_shape = NULL,
    batch_input_shape = NULL,
    batch_size = NULL,
    dtype = NULL,
    name = NULL,
)
layer dense

```r
trainable = NULL,
weights = NULL
```

**Arguments**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>object</code></td>
<td>Model or layer object</td>
</tr>
<tr>
<td><code>units</code></td>
<td>Positive integer, dimensionality of the output space.</td>
</tr>
<tr>
<td><code>activation</code></td>
<td>Name of activation function to use. If you don’t specify anything, no activation is applied (i.e. &quot;linear&quot; activation: ( a(x) = x )).</td>
</tr>
<tr>
<td><code>use_bias</code></td>
<td>Whether the layer uses a bias vector.</td>
</tr>
<tr>
<td><code>kernel_initializer</code></td>
<td>Initializer for the kernel weights matrix.</td>
</tr>
<tr>
<td><code>bias_initializer</code></td>
<td>Initializer for the bias vector.</td>
</tr>
<tr>
<td><code>kernel_regularizer</code></td>
<td>Regularizer function applied to the kernel weights matrix.</td>
</tr>
<tr>
<td><code>bias_regularizer</code></td>
<td>Regularizer function applied to the bias vector.</td>
</tr>
<tr>
<td><code>activity_regularizer</code></td>
<td>Regularizer function applied to the output of the layer (its &quot;activation&quot;).</td>
</tr>
<tr>
<td><code>kernel_constraint</code></td>
<td>Constraint function applied to the kernel weights matrix.</td>
</tr>
<tr>
<td><code>bias_constraint</code></td>
<td>Constraint function applied to the bias vector.</td>
</tr>
<tr>
<td><code>input_shape</code></td>
<td>Dimensionality of the input (integer) not including the samples axis. This argument is required when using this layer as the first layer in a model.</td>
</tr>
<tr>
<td><code>batch_input_shape</code></td>
<td>Shapes, including the batch size. For instance, <code>batch_input_shape=c(10,32)</code> indicates that the expected input will be batches of 10 32-dimensional vectors. <code>batch_input_shape=list(NULL,32)</code> indicates batches of an arbitrary number of 32-dimensional vectors.</td>
</tr>
<tr>
<td><code>batch_size</code></td>
<td>Fixed batch size for layer</td>
</tr>
<tr>
<td><code>dtype</code></td>
<td>The data type expected by the input, as a string (<code>float32</code>, <code>float64</code>, <code>int32</code>...)</td>
</tr>
<tr>
<td><code>name</code></td>
<td>An 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.</td>
</tr>
<tr>
<td><code>trainable</code></td>
<td>Whether the layer weights will be updated during training.</td>
</tr>
<tr>
<td><code>weights</code></td>
<td>Initial weights for layer</td>
</tr>
</tbody>
</table>

**Input and Output Shapes**

Input shape: nD tensor with shape: `(batch_size, ..., input_dim)`. The most common situation would be a 2D input with shape `(batch_size, input_dim)`.

Output shape: nD 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, unit)`.
See Also

Other core layers: `layer_activation()`, `layer_activity_regularizer()`, `layer_attention()`, `layer_dense_features()`, `layer_dropout()`, `layer_flatten()`, `layer_input()`, `layer_lambda()`, `layer_masking()`, `layer_permute()`, `layer_repeat_vector()`, `layer_reshape()`

Description

A layer that produces a dense Tensor based on given feature_columns.

Usage

```
layer_dense_features(
  object,  
  feature_columns, 
  name = NULL, 
  trainable = NULL, 
  input_shape = NULL, 
  batch_input_shape = NULL, 
  batch_size = NULL, 
  dtype = NULL, 
  weights = NULL
)
```

Arguments

- **object**: Model or layer object
- **feature_columns**: An iterable containing the FeatureColumns to use as inputs to your model. All items should be instances of classes derived from `DenseColumn` such as `numeric_column`, `embedding_column`, `bucketized_column`, `indicator_column`. If you have categorical features, you can wrap them with an `embedding_column` or `indicator_column`. See `tfestimators::feature_columns()`.
- **name**: An 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.
- **trainable**: Whether the layer weights will be updated during training.
- **input_shape**: Dimensionality of the input (integer) not including the samples axis. This argument is required when using this layer as the first layer in a model.
- **batch_input_shape**: Shapes, including the batch size. For instance, `batch_input_shape=c(10,32)` indicates that the expected input will be batches of 10 32-dimensional vectors. `batch_input_shape=list(NULL,32)` indicates batches of an arbitrary number of 32-dimensional vectors.
layer_depthwise_conv_2d

Description

Depthwise Separable convolutions consists in performing just the first step in a depthwise spatial convolution (which acts on each input channel separately). The depth_multiplier argument controls how many output channels are generated per input channel in the depthwise step.

Usage

layer_depthwise_conv_2d(
    object,
    kernel_size,
    strides = c(1, 1),
    padding = "valid",
    depth_multiplier = 1,
    data_format = NULL,
    dilation_rate = c(1, 1),
    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,
    input_shape = NULL,
    batch_input_shape = NULL,
    batch_size = NULL,
    dtype = NULL,
    name = NULL,
    trainable = NULL,
    weights = NULL
)

See Also

Other core layers: layer_activation(), layer_activity_regularization(), layer_attention(), layer_dense(), layer_dropout(), layer_flatten(), layer_input(), layer_lambda(), layer_masking(), layer_permute(), layer_repeat_vector(), layer_reshape()
layer_depthwise_conv_2d

Arguments

object
Model or layer object

kernel_size
An integer or list of 2 integers, specifying the width and height of the 2D convolution window. Can be a single integer to specify the same value for all spatial dimensions.

strides
An integer or list of 2 integers, specifying the strides of the convolution along the width and height. Can be a single integer to specify the same value for all spatial dimensions. Specifying any stride value != 1 is incompatible with specifying any dilation_rate value != 1.

padding
one of "valid" or "same" (case-insensitive).

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 filters_in * depth_multiplier.

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, 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
an integer or list of 2 integers, specifying the dilation rate to use for dilated convolution. Can be a single integer to specify the same value for all spatial dimensions. Currently, specifying any dilation_rate value != 1 is incompatible with specifying any stride value != 1.

activation
Activation function to use. If you don't specify anything, no activation is applied (i.e. "linear" activation: \( a(x) = x \)).

use_bias
Boolean, whether the layer uses a bias vector.

depthwise_initializer
Initializer for the depthwise kernel matrix.

bias_initializer
Initializer for the bias vector.

depthwise_regularizer
Regularizer function applied to the depthwise kernel matrix.

bias_regularizer
Regularizer function applied to the bias vector.

activity_regularizer
Regularizer function applied to the output of the layer (its "activation")

depthwise_constraint
Constraint function applied to the depthwise kernel matrix.

bias_constraint
Constraint function applied to the bias vector.

input_shape
Dimensionality of the input (integer) not including the samples axis. This argument is required when using this layer as the first layer in a model.
**layer_dot**

Layer that computes a dot product between samples in two tensors.

**Description**

Layer that computes a dot product between samples in two tensors.

**Usage**

```r
layer_dot(
  inputs,
  axes,
  normalize = FALSE,
  batch_size = NULL,
  dtype = NULL,
  name = NULL,
  trainable = NULL,
  weights = NULL
)
```

**Arguments**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>inputs</td>
<td>A list of input tensors (at least 2).</td>
</tr>
<tr>
<td>axes</td>
<td>Integer or list of integers, axis or axes along which to take the dot product.</td>
</tr>
</tbody>
</table>
layer_dropout

Applies Dropout to the input.

Description

Dropout consists in randomly setting a fraction rate of input units to 0 at each update during training time, which helps prevent overfitting.

Usage

```r
layer_dropout(
  object,
  rate,
  noise_shape = NULL,
  seed = NULL,
  input_shape = NULL,
  batch_input_shape = NULL,
  batch_size = NULL,
  name = NULL,
  trainable = NULL,
  weights = NULL
)
```
Arguments

- **object**: Model or layer object
- **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=c(batch_size,1,features).
- **seed**: integer to use as random seed.
- **input_shape**: Dimensionality of the input (integer) not including the samples axis. This argument is required when using this layer as the first layer in a model.
- **batch_input_shape**: Shapes, including the batch size. For instance, batch_input_shape=c(10, 32) indicates that the expected input will be batches of 10 32-dimensional vectors. batch_input_shape=list(NULL, 32) indicates batches of an arbitrary number of 32-dimensional vectors.
- **batch_size**: Fixed batch size for layer
- **name**: An 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.
- **trainable**: Whether the layer weights will be updated during training.
- **weights**: Initial weights for layer.

See Also

- Other core layers: `layer_activation()`, `layer_activity_regularizer()`, `layer_attention()`, `layer_dense_features()`, `layer_dense()`, `layer_flatten()`, `layer_input()`, `layer_lambda()`, `layer_masking()`, `layer_permute()`, `layer_repeat_vector()`, `layer_reshape()`
- Other dropout layers: `layer_spatial_dropout_1d()`, `layer_spatial_dropout_2d()`, `layer_spatial_dropout_3d()`

---

**layer_embedding**

Turns positive integers (indexes) into dense vectors of fixed size.

Description

For example, list(4L,20L) -> list(c(0.25,0.1),c(0.6,-0.2)) This layer can only be used as the first layer in a model.

Usage

```r
layer_embedding(
  object,
  input_dim,
  output_dim,
  embeddings_initializer = "uniform",
  embeddings_regularizer = NULL,
)```
layer_embedding

```r
activity_regularizer = NULL,
embeddings_constraint = NULL,
mask_zero = FALSE,
input_length = NULL,
batch_size = NULL,
name = NULL,
trainable = NULL,
weights = NULL
)
```

**Arguments**

- `object`  
  Model or layer object

- `input_dim`  
  int > 0. Size of the vocabulary, i.e. maximum integer index + 1.

- `output_dim`  
  int >= 0. Dimension of the dense embedding.

- `embeddings_initializer`  
  Initializer for the embeddings matrix.

- `embeddings_regularizer`  
  Regularizer function applied to the embeddings matrix.

- `activity_regularizer`  
  activity_regularizer

- `embeddings_constraint`  
  Constraint function applied to the embeddings matrix.

- `mask_zero`  
  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 inputs. 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).

- `input_length`  
  Length of input sequences, when it is constant. This argument is required if you are going to connect Flatten then Dense layers upstream (without it, the shape of the dense outputs cannot be computed).

- `batch_size`  
  Fixed batch size for layer

- `name`  
  An 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.

- `trainable`  
  Whether the layer weights will be updated during training.

- `weights`  
  Initial weights for layer.

**Input shape**

2D tensor with shape: (batch_size, sequence_length).

**Output shape**

3D tensor with shape: (batch_size, sequence_length, output_dim).
References

- A Theoretically Grounded Application of Dropout in Recurrent Neural Networks

layer_flatten  Flattens an input

Description

Flatten a given input, does not affect the batch size.

Usage

layer_flatten(
  object,
  data_format = NULL,
  input_shape = NULL,
  dtype = NULL,
  name = NULL,
  trainable = NULL,
  weights = NULL
)

Arguments

object  Model or layer object

data_format  A string, one of channels_last (default) or channels_first. The ordering of the dimensions in the inputs. The purpose of this argument is to preserve weight ordering when switching a model from one data format to another. channels_last corresponds to inputs with shape (batch, ..., channels) while channels_first corresponds to inputs with shape (batch, channels, ...). 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".

input_shape  Input shape (list of integers, does not include the samples axis) which is required when using this layer as the first layer in a model.

dtype  The data type expected by the input, as a string (float32, float64, int32...)

name  An 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.

trainable  Whether the layer weights will be updated during training.

weights  Initial weights for layer.

See Also

Other core layers: layer_activation(), layer_activity_regularization(), layer_attention(), layer_dense_features(), layer_dense(), layer_dropout(), layer_input(), layer_lambda(), layer_masking(), layer_permute(), layer_repeat_vector(), layer_reshape()
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,
    input_shape = NULL,
    batch_input_shape = NULL,
    batch_size = NULL,
    dtype = NULL,
    name = NULL,
    trainable = NULL,
    weights = NULL
)

Arguments

object Model or layer object
rate float, drop probability (as with Dropout). The multiplicative noise will have standard deviation sqrt(rate / (1 - rate)).
input_shape Dimensionality of the input (integer) not including the samples axis. This argument is required when using this layer as the first layer in a model.
batch_input_shape Shapes, including the batch size. For instance, batch_input_shape=c(10,32) indicates that the expected input will be batches of 10 32-dimensional vectors. batch_input_shape=list(NULL,32) indicates batches of an arbitrary number of 32-dimensional vectors.
batch_size Fixed batch size for layer
dtype The data type expected by the input, as a string (float32, float64, int32...)
name An 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.
trainable Whether the layer weights will be updated during training.
weights Initial weights for layer.

Input shape

Arbitrary. Use the keyword argument input_shape (list of integers, does not include the samples axis) when using this layer as the first layer in a model.
**layer_gaussian_noise**

### Output shape

Same shape as input.

### References


### See Also

Other noise layers: `layer_alpha_dropout()`, `layer_gaussian_noise()`

---

**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

```r
layer_gaussian_noise(
    object,
    stddev,
    input_shape = NULL,
    batch_input_shape = NULL,
    batch_size = NULL,
    dtype = NULL,
    name = NULL,
    trainable = NULL,
    weights = NULL
)
```

### Arguments

- **object**
  Model or layer object

- **stddev**
  float, standard deviation of the noise distribution.

- **input_shape**
  Dimensionality of the input (integer) not including the samples axis. This argument is required when using this layer as the first layer in a model.

- **batch_input_shape**
  Shapes, including the batch size. For instance, `batch_input_shape=c(10,32)` indicates that the expected input will be batches of 10 32-dimensional vectors. `batch_input_shape=list(NULL,32)` indicates batches of an arbitrary number of 32-dimensional vectors.
layer_global_average_pooling_1d

batch_size: Fixed batch size for layer

dtype: The data type expected by the input, as a string (float32, float64, int32...)

name: An 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.

trainable: Whether the layer weights will be updated during training.

weights: Initial weights for layer.

Input shape

Arbitrary. Use the keyword argument input_shape (list 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

Other noise layers: layer_alpha_dropout(), layer_gaussian_dropout()

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 = "channels_last",
    batch_size = NULL,
    name = NULL,
    trainable = NULL,
    weights = NULL
)

Arguments

object: Model or layer object

data_format: One of channels_last (default) or channels_first. The ordering of the di-
dimensions in the inputs.

batch_size: Fixed batch size for layer
name
An 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.

trainable
Whether the layer weights will be updated during training.

weights
Initial weights for layer.

Input shape
3D tensor with shape: (batch_size, steps, features).

Output shape
2D tensor with shape: (batch_size, channels)

See Also
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()

layer_global_average_pooling_2d
Global average pooling operation for spatial data.

Description
Global average pooling operation for spatial data.

Usage
layer_global_average_pooling_2d(
    object,
    data_format = NULL,
    batch_size = NULL,
    name = NULL,
    trainable = NULL,
    weights = NULL
)

Arguments
object
Model or layer object

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, 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".
**layer_global_average_pooling_3d**

batch_size  
Fixed batch size for layer

name  
An 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.

trainable  
Whether the layer weights will be updated during training.

weights  
Initial weights for layer.

**Input shape**

- If `data_format='channels_last'`: 4D tensor with shape: (batch_size, rows, cols, channels)
- If `data_format='channels_first'`: 4D tensor with shape: (batch_size, channels, rows, cols)

**Output shape**

2D tensor with shape: (batch_size, channels)

**See Also**

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()`

```r
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,
  batch_size = NULL,
  name = NULL,
  trainable = NULL,
  weights = NULL
)```
Arguments

- **object**: Model or layer object
- **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, spatial_dim1, spatial_dim2, spatial_dim3, channels)` while `channels_first` corresponds to inputs with shape `(batch, channels, spatial_dim1, spatial_dim2, spatial_dim3)`. 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".
- **batch_size**: Fixed batch size for layer
- **name**: An 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.
- **trainable**: Whether the layer weights will be updated during training.
- **weights**: Initial weights for layer.

Input shape

- If `data_format='channels_last'`: 5D tensor with shape: (batch_size, spatial_dim1, spatial_dim2, spatial_dim3, channels)
- If `data_format='channels_first'`: 5D tensor with shape: (batch_size, channels, spatial_dim1, spatial_dim2, spatial_dim3)

Output shape

2D tensor with shape: (batch_size, channels)

See Also

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()`

---

```
layer_global_max_pooling_1d

Global max pooling operation for temporal data.
```

Description

Global max pooling operation for temporal data.
layer_global_max_pooling_2d

Usage

layer_global_max_pooling_2d(
    object,
    data_format = "channels_last",
    batch_size = NULL,
    name = NULL,
    trainable = NULL,
    weights = NULL
)

Arguments

object Model or layer object

data_format One of channels_last (default) or channels_first. The ordering of the dimensions in the inputs.
batch_size Fixed batch size for layer
name An 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.
trainable Whether the layer weights will be updated during training.
weights Initial weights for layer.

Input shape

3D tensor with shape: (batch_size, steps, features).

Output shape

2D tensor with shape: (batch_size, channels)

See Also

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()

layer_global_max_pooling_2d

Global max pooling operation for spatial data.

Description

Global max pooling operation for spatial data.
layer_global_max_pooling_2d

Usage

layer_global_max_pooling_2d(
    object,
    data_format = NULL,
    batch_size = NULL,
    name = NULL,
    trainable = NULL,
    weights = NULL
)

Arguments

object Model or layer object

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, 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".

batch_size Fixed batch size for layer

name An 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.

trainable Whether the layer weights will be updated during training.

weights Initial weights for layer.

Input shape

- If data_format=’channels_last’: 4D tensor with shape: (batch_size, rows, cols, channels)
- If data_format=’channels_first’: 4D tensor with shape: (batch_size, channels, rows, cols)

Output shape

2D tensor with shape: (batch_size, channels)

See Also

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(), layer_max_pooling_3d()
Global Max pooling operation for 3D data.

**Description**

Global Max pooling operation for 3D data.

**Usage**

```python
layer_global_max_pooling_3d(
    object,
    data_format = NULL,
    batch_size = NULL,
    name = NULL,
    trainable = NULL,
    weights = NULL
)
```

**Arguments**

- **object**: Model or layer object
- **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, spatial_dim1, spatial_dim2, spatial_dim3, channels) while `channels_first` corresponds to inputs with shape (batch, channels, spatial_dim1, spatial_dim2, spatial_dim3). 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".
- **batch_size**: Fixed batch size for layer
- **name**: An 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.
- **trainable**: Whether the layer weights will be updated during training.
- **weights**: Initial weights for layer.

**Input shape**

- If `data_format='channels_last'`: 5D tensor with shape: (batch_size, spatial_dim1, spatial_dim2, spatial_dim3, channels)
- If `data_format='channels_first'`: 5D tensor with shape: (batch_size, channels, spatial_dim1, spatial_dim2, spatial_dim3)

**Output shape**

2D tensor with shape: (batch_size, channels)
See Also

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()

---

layer_gru

Gated Recurrent Unit - Cho et al.

---

Description

There are two variants. The default one is based on 1406.1078v3 and has reset gate applied to
hidden state before matrix multiplication. The other one is based on original 1406.1078v1 and has
the order reversed.

Usage

layer_gru(
    object,
    units,
    activation = "tanh",
    recurrent_activation = "hard_sigmoid",
    use_bias = TRUE,
    return_sequences = FALSE,
    return_state = FALSE,
    go_backwards = FALSE,
    stateful = FALSE,
    unroll = FALSE,
    time_major = FALSE,
    reset_after = FALSE,
    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,
    input_shape = NULL,
    batch_input_shape = NULL,
    batch_size = NULL,
    dtype = NULL,
layer_gru

name = NULL, 
trainable = NULL, 
weights = NULL )

Arguments

object Model or layer object
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.
use_bias Boolean, whether the layer uses a bias vector.
return_sequences Boolean. 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.
time_major If True, the inputs and outputs will be in shape [timesteps, batch, feature], whereas in the False case, it will be [batch, timesteps, feature]. Using time_major = TRUE is a bit more efficient because it avoids transposes at the beginning and end of the RNN calculation. However, most TensorFlow data is batch-major, so by default this function accepts input and emits output in batch-major form.
reset_after GRU convention (whether to apply reset gate after or before matrix multiplication). FALSE = "before" (default), TRUE = "after" (CuDNN compatible).
kernel_initializer
recurrernt_initializer
bias_initializer
kernel_regularizer
Recurrent_regularizer

Regularizer function applied to the kernel weights matrix.

Regularizer function applied to the recurrent_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.
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.
input_shape
   Dimensionality of the input (integer) not including the samples axis. This argument is required when using this layer as the first layer in a model.
batch_input_shape
   Shapes, including the batch size. For instance, batch_input_shape=c(10, 32) indicates that the expected input will be batches of 10 32-dimensional vectors. batch_input_shape=list(NULL, 32) indicates batches of an arbitrary number of 32-dimensional vectors.
batch_size
   Fixed batch size for layer
dtype
   The data type expected by the input, as a string (float32, float64, int32...)
name
   An 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.
trainable
   Whether the layer weights will be updated during training.
weights
   Initial weights for layer.

Details

The second variant is compatible with CuDNNGRU (GPU-only) and allows inference on CPU. Thus it has separate biases for kernel and recurrent_kernel. Use reset_after = TRUE and recurrent_activation = "sigmoid".

Input shapes

3D tensor with shape (batch_size, timesteps, input_dim), (Optional) 2D tensors with shape (batch_size, output_dim).

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, units).
- if return_sequences: 3D tensor with shape (batch_size, timesteps, units).
- else, 2D tensor with shape (batch_size, units).
Masking

This layer supports masking for input data with a variable number of timesteps. To introduce masks to your data, use an embedding layer with the mask_zero parameter set to TRUE.

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. For intuition behind statefulness, there is a helpful blog post here: https://philipperemy.github.io/keras-stateful-lstm/

To enable statefulness:

- Specify stateful = TRUE in the layer constructor.
- Specify a fixed batch size for your model. For sequential models, pass batch_input_shape = c(...) to the first layer in your model. For functional models with 1 or more Input layers, pass batch_shape = c(...) to all the first layers in your model. This is the expected shape of your inputs including the batch size. It should be a vector of integers, e.g. c(32, 10, 100). For dimensions which can vary (are not known ahead of time), use NULL in place of an integer, e.g. c(32, NULL, NULL).
- Specify shuffle = FALSE when calling fit().

To reset the states of your model, call reset_states() on either a specific layer, or on your entire model.

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.

You can specify the initial state of RNN layers numerically by calling reset_states with the keyword argument states. The value of states should be a numpy array or list of numpy arrays representing the initial state of the RNN layer.

References

- Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation
- On the Properties of Neural Machine Translation: Encoder-Decoder Approaches
- Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling
- A Theoretically Grounded Application of Dropout in Recurrent Neural Networks

See Also

Other recurrent layers: layer_cudnn_gru(), layer_cudnn_lstm(), lstm(), layer_simple_rnn()
layer_input  

**Input layer**

**Description**

Layer to be used as an entry point into a graph.

**Usage**

```r
layer_input(
    shape = NULL,
    batch_shape = NULL,
    name = NULL,
    dtype = NULL,
    sparse = FALSE,
    tensor = NULL,
    ragged = FALSE
)
```

**Arguments**

- `shape`  
  Shape, not including the batch size. For instance, `shape=c(32)` indicates that the expected input will be batches of 32-dimensional vectors.

- `batch_shape`  
  Shape, including the batch size. For instance, `shape = c(10, 32)` indicates that the expected input will be batches of 10 32-dimensional vectors. `batch_shape = list(NULL, 32)` indicates batches of an arbitrary number of 32-dimensional vectors.

- `name`  
  An 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.

- `dtype`  
  The data type expected by the input, as a string (`float32`, `float64`, `int32`...)

- `sparse`  
  Boolean, whether the placeholder created is meant to be sparse.

- `tensor`  
  Existing tensor to wrap into the Input layer. If set, the layer will not create a placeholder tensor.

- `ragged`  
  A boolean specifying whether the placeholder to be created is ragged. Only one of `ragged` and `sparse` can be `TRUE`. In this case, values of `NULL` in the `shape` argument represent ragged dimensions.

**Value**

A tensor

**See Also**

Other core layers: `layer_activation()`, `layer_activity_regularization()`, `layer_attention()`, `layer_dense_features()`, `layer_dense()`, `layer_dropout()`, `layer_flatten()`, `layer_lambda()`, `layer_masking()`, `layer_permute()`, `layer_repeat_vector()`, `layer_reshape()`
layer_lambda

<table>
<thead>
<tr>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wraps arbitrary expression as a layer</td>
</tr>
</tbody>
</table>

**Usage**

```r
layer_lambda(
    object,
    f,
    output_shape = NULL,
    mask = NULL,
    arguments = NULL,
    input_shape = NULL,
    batch_input_shape = NULL,
    batch_size = NULL,
    dtype = NULL,
    name = NULL,
    trainable = NULL,
    weights = NULL
)
```

**Arguments**

<table>
<thead>
<tr>
<th>object</th>
<th>Model or layer object</th>
</tr>
</thead>
<tbody>
<tr>
<td>f</td>
<td>The function to be evaluated. Takes input tensor as first argument.</td>
</tr>
<tr>
<td>output_shape</td>
<td>Expected output shape from the function (not required when using TensorFlow back-end).</td>
</tr>
<tr>
<td>mask</td>
<td>mask</td>
</tr>
<tr>
<td>arguments</td>
<td>optional named list of keyword arguments to be passed to the function.</td>
</tr>
<tr>
<td>input_shape</td>
<td>Dimensionality of the input (integer) not including the samples axis. This argument is required when using this layer as the first layer in a model.</td>
</tr>
<tr>
<td>batch_input_shape</td>
<td>Shapes, including the batch size. For instance, batch_input_shape=c(10, 32) indicates that the expected input will be batches of 10 32-dimensional vectors. batch_input_shape=list(NULL, 32) indicates batches of an arbitrary number of 32-dimensional vectors.</td>
</tr>
<tr>
<td>batch_size</td>
<td>Fixed batch size for layer</td>
</tr>
<tr>
<td>dtype</td>
<td>The data type expected by the input, as a string (float32, float64, int32...)</td>
</tr>
<tr>
<td>name</td>
<td>An 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.</td>
</tr>
<tr>
<td>trainable</td>
<td>Whether the layer weights will be updated during training.</td>
</tr>
<tr>
<td>weights</td>
<td>Initial weights for layer.</td>
</tr>
</tbody>
</table>
layer_layer_normalization

Input shape

Arbitrary. Use the keyword argument input_shape (list of integers, does not include the samples axis) when using this layer as the first layer in a model.

Output shape

Arbitrary (based on tensor returned from the function)

See Also

Other core layers: layer_activation(), layer_activity_regularizer(), layer_attention(), layer_dense_features(), layer_dense(), layer_dropout(), layer_flatten(), layer_input(), layer_masking(), layer_permute(), layer_repeat_vector(), layer_reshape()

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.

Usage

layer_layer_normalization(
    object,
    axis = -1,
    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,
    trainable = TRUE,
    name = NULL
)

Arguments

object Model or layer object
layer_locally_connected_1d

**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. This argument defaults to -1, the last dimension in the input.

**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. Defaults to True. When the next layer is linear (also e.g. nn.relu), this can be disabled since the scaling will be done by the next layer.

**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. None by default.

**gamma_regularizer**
Optional regularizer for the gamma weight. None by default.

**beta_constraint**
Optional constraint for the beta weight. None by default.

**gamma_constraint**
Optional constraint for the gamma weight. None by default.

**trainable**
Boolean, if True the variables will be marked as trainable. Defaults to True.

**name**
An 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.

**Details**
Given a tensor inputs, moments are calculated and normalization is performed across the axes specified in axis.

---

**layer_locally_connected_1d**

Locally-connected layer for 1D inputs.

---

**Description**

`layer_locally_connected_1d()` works similarly to `layer_conv_1d()`, except that weights are unshared, that is, a different set of filters is applied at each different patch of the input.
layer_locally_connected_1d(
    object,
    filters,
    kernel_size,
    strides = 1L,
    padding = "valid",
    data_format = NULL,
    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,
    implementation = 1L,
    batch_size = NULL,
    name = NULL,
    trainable = NULL,
    weights = NULL
)

Arguments

object        Model or layer object
filters       Integer, the dimensionality of the output space (i.e. the number output of filters
              in the convolution).
kernel_size   An integer or list of a single integer, specifying the length of the 1D convolution
              window.
strides       An integer or list of a single integer, specifying the stride length of the con-
              volution. Specifying any stride value != 1 is incompatible with specifying any
              dilation_rate value != 1.
padding       Currently only supports "valid" (case-insensitive). "same" may be supported
              in the future.
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, 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".
activation    Activation function to use. If you don’t specify anything, no activation is applied
              (i.e. "linear" activation: a(x) = x).
use_bias      Boolean, whether the layer uses a bias vector.
layer_locally_connected_1d

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 matrix.
bias_constraint
Constraint function applied to the bias vector.
implementation
either 1, 2, or 3. 1 loops over input spatial locations to perform the forward pass. It is memory-efficient but performs a lot of (small) ops. 2 stores layer weights in a dense but sparsely-populated 2D matrix and implements the forward pass as a single matrix-multiply. It uses a lot of RAM but performs few (large) ops. 3 stores layer weights in a sparse tensor and implements the forward pass as a single sparse matrix-multiply. How to choose: 1: large, dense models, 2: small models, 3: large, sparse models, where "large" stands for large input/output activations (i.e. many filters, input_filters, large input_size, output_size), and "sparse" stands for few connections between inputs and outputs, i.e. small ratio filters * input_filters * kernel_size / (input_size * strides), where inputs to and outputs of the layer are assumed to have shapes (input_size, input_filters), (output_size, filters) respectively. It is recommended to benchmark each in the setting of interest to pick the most efficient one (in terms of speed and memory usage). Correct choice of implementation can lead to dramatic speed improvements (e.g. 50X), potentially at the expense of RAM. Also, only padding="valid" is supported by implementation=1.
batch_size
Fixed batch size for layer
name
An 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.
trainable
Whether the layer weights will be updated during training.
weights
Initial weights for layer.

Input shape
3D tensor with shape: (batch_size, steps, input_dim)

Output shape
3D tensor with shape: (batch_size, new_steps, filters) steps value might have changed due to padding or strides.

See Also
Other locally connected layers: layer_locally_connected_2d()
layer_locally_connected_2d

Locally-connected layer for 2D inputs.

Description

layer_locally_connected_2d works similarly to layer_conv_2d(), except that weights are un-shared, that is, a different set of filters is applied at each different patch of the input.

Usage

layer_locally_connected_2d(
  object,
  filters,
  kernel_size,
  strides = c(1L, 1L),
  padding = "valid",
  data_format = NULL,
  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,
  implementation = 1L,
  batch_size = NULL,
  name = NULL,
  trainable = NULL,
  weights = NULL
)

Arguments

object Model or layer object
filters Integer, the dimensionality of the output space (i.e. the number output of filters in the convolution).
kernel_size An integer or list of 2 integers, specifying the width and height of the 2D convolution window. Can be a single integer to specify the same value for all spatial dimensions.
strides An integer or list of 2 integers, specifying the strides of the convolution along the width and height. Can be a single integer to specify the same value for all spatial dimensions. Specifying any stride value != 1 is incompatible with specifying any dilation_rate value != 1.
padding: Currently only supports "valid" (case-insensitive). "same" may be supported in the future.

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, width, height, channels) while channels_first corresponds to inputs with shape (batch, channels, width, height). 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".

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 matrix.

bias_constraint: Constraint function applied to the bias vector.

implementation: either 1, 2, or 3. 1 loops over input spatial locations to perform the forward pass. It is memory-efficient but performs a lot of (small) ops. 2 stores layer weights in a dense but sparsely-populated 2D matrix and implements the forward pass as a single matrix-multiply. It uses a lot of RAM but performs few (large) ops. 3 stores layer weights in a sparse tensor and implements the forward pass as a single sparse matrix-multiply. How to choose: 1: large, dense models, 2: small models, 3: large, sparse models, where "large" stands for large input/output activations (i.e. many filters, input filters, large input size, output size), and "sparse" stands for few connections between inputs and outputs, i.e. small ratio filters * input filters * kernel size / (input size * strides), where inputs to and outputs of the layer are assumed to have shapes (input size, input filters, output size, filters) respectively. It is recommended to benchmark each in the setting of interest to pick the most efficient one (in terms of speed and memory usage). Correct choice of implementation can lead to dramatic speed improvements (e.g. 50X), potentially at the expense of RAM. Also, only padding="valid" is supported by implementation=1.

batch_size: Fixed batch size for layer,

name: An 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.

trainable: Whether the layer weights will be updated during training.

weights: Initial weights for layer.
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
4D tensor with shape: (samples, filters, new_rows, new_cols) if data_format='channels_first' or 4D tensor with shape: (samples, new_rows, new_cols, filters) if data_format='channels_last'. rows and cols values might have changed due to padding.

See Also
Other locally connected layers: layer_locally_connected_1d()
layer_lstm

recurrent_dropout = 0,
input_shape = NULL,
batch_input_shape = NULL,
batch_size = NULL,
dtype = NULL,
name = NULL,
trainable = NULL,
weights = NULL
)

Arguments

- **object**: Model or layer object
- **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 (i.e. "linear" activation: \( a(x) = x \)).
- **recurrent_activation**: Activation function to use for the recurrent step.
- **use_bias**: Boolean, whether the layer uses a bias vector.
- **return_sequences**: Boolean. 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.
- **time_major**: If True, the inputs and outputs will be in shape [timesteps, batch, feature], whereas in the False case, it will be [batch, timesteps, feature]. Using time_major = TRUE is a bit more efficient because it avoids transposes at the beginning and end of the RNN calculation. However, most TensorFlow data is batch-major, so by default this function accepts input and emits output in batch-major form.
- **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.
- **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. 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.

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 the output of the layer (its "activation").

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.

input_shape
   Dimensionality of the input (integer) not including the samples axis. This argument is required when using this layer as the first layer in a model.

batch_input_shape
   Shapes, including the batch. For instance, batch_input_shape=c(10,32) indicates that the expected input will be batches of 10 32-dimensional vectors.
   batch_input_shape=list(NULL,32) indicates batches of an arbitrary number of 32-dimensional vectors.

batch_size
   Fixed batch size for layer

dtype
   The data type expected by the input, as a string (float32, float64, int32...)

name
   An 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.

trainable
   Whether the layer weights will be updated during training.

weights
   Initial weights for layer.

**Input shapes**

3D tensor with shape (batch_size, timesteps, input_dim), (Optional) 2D tensors with shape (batch_size, output_dim).
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, units).
- if return_sequences: 3D tensor with shape (batch_size, timesteps, units).
- else, 2D tensor with shape (batch_size, units).

Masking

This layer supports masking for input data with a variable number of timesteps. To introduce masks to your data, use an embedding layer with the mask_zero parameter set to TRUE.

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. For intuition behind statefulness, there is a helpful blog post here: https://philipperemy.github.io/keras-stateful-lstm/

To enable statefulness:

- Specify stateful = TRUE in the layer constructor.
- Specify a fixed batch size for your model. For sequential models, pass batch_input_shape = c(...) to the first layer in your model. For functional models with 1 or more Input layers, pass batch_shape = c(...) to all the first layers in your model. This is the expected shape of your inputs including the batch size. It should be a vector of integers, e.g. c(32,10,100). For dimensions which can vary (are not known ahead of time), use NULL in place of an integer, e.g. c(32,NULL,NULL).
- Specify shuffle = FALSE when calling fit().

To reset the states of your model, call reset_states() on either a specific layer, or on your entire model.

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.

You can specify the initial state of RNN layers numerically by calling reset_states with the keyword argument states. The value of states should be a numpy array or list of numpy arrays representing the initial state of the RNN layer.

References

- Long short-term memory (original 1997 paper)
- Supervised sequence labeling with recurrent neural networks
- A Theoretically Grounded Application of Dropout in Recurrent Neural Networks
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,  
    input_shape = NULL,  
    batch_input_shape = NULL,  
    batch_size = NULL,  
    dtype = NULL,  
    name = NULL,  
    trainable = NULL,  
    weights = NULL
)

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>object</td>
<td>Model or layer object</td>
</tr>
<tr>
<td>mask_value</td>
<td>float, mask value</td>
</tr>
<tr>
<td>input_shape</td>
<td>Dimensionality of the input (integer) not including the samples axis. This argument is required when using this layer as the first layer in a model.</td>
</tr>
<tr>
<td>batch_input_shape</td>
<td>Shapes, including the batch size. For instance, batch_input_shape=c(10,32) indicates that the expected input will be batches of 10 32-dimensional vectors. batch_input_shape=list(NULL,32) indicates batches of an arbitrary number of 32-dimensional vectors.</td>
</tr>
<tr>
<td>batch_size</td>
<td>Fixed batch size for layer</td>
</tr>
<tr>
<td>dtype</td>
<td>The data type expected by the input, as a string (float32, float64, int32...)</td>
</tr>
<tr>
<td>name</td>
<td>An 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.</td>
</tr>
<tr>
<td>trainable</td>
<td>Whether the layer weights will be updated during training.</td>
</tr>
<tr>
<td>weights</td>
<td>Initial weights for layer</td>
</tr>
</tbody>
</table>

See Also

Other recurrent layers: layer_cudnn_gru(), layer_cudnn_lstm(), layer_gru(), layer_simple_rnn()
See Also

Other core layers: `layer_activation()`, `layer_activity_regularization()`, `layer_attention()`, `layer_dense_features()`, `layer_dense()`, `layer_dropout()`, `layer_flatten()`, `layer_input()`, `layer_lambda()`, `layer_permute()`, `layer_repeat_vector()`, `layer_reshape()`

---

**layer_maximum**  
*Layer that computes the maximum (element-wise) 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**

```r
layer_maximum(
inputs,
batch_size = NULL,
dtype = NULL,
name = NULL,
trainable = NULL,
weights = NULL
)
```

**Arguments**

- **inputs**: A list of input tensors (at least 2).
- **batch_size**: Fixed batch size for layer.
- **dtype**: The data type expected by the input, as a string (`float32`, `float64`, `int32`...)
- **name**: An 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.
- **trainable**: Whether the layer weights will be updated during training.
- **weights**: Initial weights for layer.

**Value**

A tensor, the element-wise maximum of the inputs.

**See Also**

Other merge layers: `layer_add()`, `layer_average()`, `layer_concatenate()`, `layer_dot()`, `layer_minimum()`, `layer_multiply()`, `layer_subtract()`
**layer_max_pooling_1d**  
*Max pooling operation for temporal data.*

**Description**
Max pooling operation for temporal data.

**Usage**
```r
layer_max_pooling_1d(
  object,
  pool_size = 2L,
  strides = NULL,
  padding = "valid",
  data_format = "channels_last",
  batch_size = NULL,
  name = NULL,
  trainable = NULL,
  weights = NULL
)
```

**Arguments**
- **object**: Model or layer object
- **pool_size**: Integer, size of the max pooling windows.
- **strides**: Integer, or NULL. Factor by which to downscale. E.g. 2 will halve the input. If NULL, it will default to pool_size.
- **padding**: One of "valid" or "same" (case-insensitive).
- **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, steps, features) while channels_first corresponds to inputs with shape (batch, features, steps).
- **batch_size**: Fixed batch size for layer
- **name**: An 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.
- **trainable**: Whether the layer weights will be updated during training.
- **weights**: Initial weights for layer.

**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).
**layer_max_pooling_2d**

**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).

**See Also**

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()`

---

**layer_max_pooling_2d**  
*Max pooling operation for spatial data.*

---

**Description**

Max pooling operation for spatial data.

**Usage**

```r
layer_max_pooling_2d(
  object,
  pool_size = c(2L, 2L),
  strides = NULL,
  padding = "valid",
  data_format = NULL,
  batch_size = NULL,
  name = NULL,
  trainable = NULL,
  weights = NULL
)
```

**Arguments**

- **object**: Model or layer object
- **pool_size**: Integer or list of 2 integers, factors by which to downscale (vertical, horizontal). (2, 2) will halve the input in both spatial dimension. If only one integer is specified, the same window length will be used for both dimensions.
- **strides**: Integer, list of 2 integers, or NULL. Strides values. If NULL, it will default to `pool_size`.
- **padding**: One of "valid" or "same" (case-insensitive).
- **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, 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".
layer_max_pooling_3d

batch_size  
Fixed batch size for layer

name  
An 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.

trainable  
Whether the layer weights will be updated during training.

weights  
Initial weights for layer.

Input shape

- If `data_format='channels_last'`: 4D tensor with shape: (batch_size, rows, cols, channels)
- If `data_format='channels_first'`: 4D tensor with shape: (batch_size, channels, rows, cols)

Output shape

- If `data_format='channels_last'`: 4D tensor with shape: (batch_size, pooled_rows, pooled_cols, channels)
- If `data_format='channels_first'`: 4D tensor with shape: (batch_size, channels, pooled_rows, pooled_cols)

See Also

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()

layer_max_pooling_3d  
Max pooling operation for 3D data (spatial or spatio-temporal).

Description

Max pooling operation for 3D data (spatial or spatio-temporal).

Usage

layer_max_pooling_3d(
  object,
  pool_size = c(2L, 2L, 2L),
  strides = NULL,
  padding = "valid",
  data_format = NULL,
  batch_size = NULL,
  name = NULL,
  trainable = NULL,
  weights = NULL
)
layer_max_pooling_3d

Arguments

object  
Model or layer object

pool_size  
list of 3 integers, factors by which to downscale (dim1, dim2, dim3). (2, 2, 2) will halve the size of the 3D input in each dimension.

strides  
list of 3 integers, or NULL. Strides values.

padding  
One of "valid" or "same" (case-insensitive).

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, spatial_dim1, spatial_dim2, spatial_dim3, channels) while channels_first corresponds to inputs with shape (batch, channels, spatial_dim1, spatial_dim2, spatial_dim3). 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".

batch_size  
Fixed batch size for layer

name  
An 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.

trainable  
Whether the layer weights will be updated during training.

weights  
Initial the layer weights will be updated during training.

Input shape

- If data_format='channels_last': 5D tensor with shape: (batch_size, spatial_dim1, spatial_dim2, spatial_dim3, channels)
- If data_format='channels_first': 5D tensor with shape: (batch_size, channels, spatial_dim1, spatial_dim2, spatial_dim3)

Output shape

- If data_format='channels_last': 5D tensor with shape: (batch_size, pooled_dim1, pooled_dim2, pooled_dim3, channels)
- If data_format='channels_first': 5D tensor with shape: (batch_size, channels, pooled_dim1, pooled_dim2, pooled_dim3)

See Also

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()
layer_minimum

Layer that computes the minimum (element-wise) 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,
  batch_size = NULL,
  dtype = NULL,
  name = NULL,
  trainable = NULL,
  weights = NULL
)

Arguments

inputs A list of input tensors (at least 2).
batch_size Fixed batch size for layer
dtype The data type expected by the input, as a string (float32, float64, int32...)
name An 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.
trainable Whether the layer weights will be updated during training.
weights Initial weights for layer.

Value

A tensor, the element-wise maximum of the inputs.

See Also

Other merge layers: layer_add(), layer_average(), layer_concatenate(), layer_dot(), layer_maximum(), layer_multiply(), layer_subtract()
**layer_multiply**  
*Layer that multiplies (element-wise) 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**

```python
layer_multiply(
    inputs,
    batch_size = NULL,
    dtype = NULL,
    name = NULL,
    trainable = NULL,
    weights = NULL
)
```

**Arguments**

- **inputs**: A list of input tensors (at least 2).
- **batch_size**: Fixed batch size for layer.
- **dtype**: The data type expected by the input, as a string (float32, float64, int32...)
- **name**: An 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.
- **trainable**: Whether the layer weights will be updated during training.
- **weights**: Initial weights for layer.

**Value**

A tensor, the element-wise product of the inputs.

**See Also**

Other merge layers: `layer_add()`, `layer_average()`, `layer_concatenate()`, `layer_dot()`, `layer_maximum()`, `layer_minimum()`, `layer_subtract()`
layer_multi_head_attention

*MultiHeadAttention layer*

Description

This is an implementation of multi-headed attention based on "Attention is all you Need". 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.

Usage

```r
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**
  - a list of inputs first should be the query tensor, the second the value tensor
- **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 key feature dim.
- **attention_axes**
  - axes over which the attention is applied. None 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.

... Other arguments passed to the layer. Eg, name, training.

Details

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, , key_dim], [batch_size, , key_dim], [batch_size, , 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 an linear projection and return.

Value

- 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 None. Otherwise, the multi-head outputs are project to the shape specified by output_shape.

- attention_scores: (Optional) multi-head attention coefficients over attention axes.

Call arguments

- query: Query Tensor of shape [B, T, dim].
- value: Value Tensor of shape [B, S, dim].
- 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.
- return_attention_scores: A boolean to indicate whether the output should be attention output if TRUE, or (attention_output, attention_scores) 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). Defaults to either using the training mode of the parent layer/model, or FALSE (inference) if there is no parent layer.
layer_permute

Permute the dimensions of an input according to a given pattern

Description

Permute the dimensions of an input according to a given pattern

Usage

layer_permute(
  object,
  dims,
  input_shape = NULL,
  batch_input_shape = NULL,
  batch_size = NULL,
  dtype = NULL,
  name = NULL,
  trainable = NULL,
  weights = NULL
)

Arguments

object      Model or layer object
 dims       List of integers. Permutation pattern, does not include the samples dimension. Indexing starts at 1. For instance, (2, 1) permutes the first and second dimension of the input.
 input_shape Input shape (list of integers, does not include the samples axis) which is required when using this layer as the first layer in a model.
 batch_input_shape Shapes, including the batch size. For instance, batch_input_shape=c(10,32) indicates that the expected input will be batches of 10 32-dimensional vectors. batch_input_shape=list(NULL,32) indicates batches of an arbitrary number of 32-dimensional vectors.
 batch_size   Fixed batch size for layer
 dtype       The data type expected by the input, as a string (float32, float64, int32...)
 name        An 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.
 trainable   Whether the layer weights will be updated during training.
 weights     Initial weights for layer.

Input and Output Shapes

Input shape: Arbitrary
Output shape: Same as the input shape, but with the dimensions re-ordered according to the specified pattern.
Note

Useful for e.g. connecting RNNs and convnets together.

See Also

Other core layers: layer_activation(), layer_activity_regularization(), layer_attention(), layer_dense_features(), layer_dense(), layer_dropout(), layer_flatten(), layer_input(), layer_lambda(), layer_masking(), layer_repeat_vector(), layer_reshape()

layer_repeat_vector

Repeats the input n times.

Description

Repeats the input n times.

Usage

layer_repeat_vector(
    object,
    n,
    batch_size = NULL,
    name = NULL,
    trainable = NULL,
    weights = NULL
)

Arguments

object Model or layer object
n integer, repetition factor.
batch_size Fixed batch size for layer
name An 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.
trainable Whether the layer weights will be updated during training.
weights Initial weights for layer.

Input shape

2D tensor of shape (num_samples, features).

Output shape

3D tensor of shape (num_samples, n, features).
See Also

Other core layers: layer_activation(), layer_activity_regularization(), layer_attention(), layer_dense_features(), layer_dense(), layer_dropout(), layer_flatten(), layer_input(), layer_lambda(), layer_masking(), layer_permute(), layer_reshape()

layer_reshape

Reshapes an output to a certain shape.

Description

Reshapes an output to a certain shape.

Usage

layer_reshape(
  object,
  target_shape,
  input_shape = NULL,
  batch_input_shape = NULL,
  batch_size = NULL,
  dtype = NULL,
  name = NULL,
  trainable = NULL,
  weights = NULL
)

Arguments

object Model or layer object

target_shape List of integers, does not include the samples dimension (batch size).

input_shape Input shape (list of integers, does not include the samples axis) which is required when using this layer as the first layer in a model.

batch_input_shape Shapes, including the batch size. For instance, batch_input_shape=c(10, 32) indicates that the expected input will be batches of 10 32-dimensional vectors. batch_input_shape=list(NULL, 32) indicates batches of an arbitrary number of 32-dimensional vectors.

batch_size Fixed batch size for layer

dtype The data type expected by the input, as a string (float32, float64, int32...)

name An 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.

trainable Whether the layer weights will be updated during training.

weights Initial weights for layer.
Input and Output Shapes

Input shape: Arbitrary, although all dimensions in the input shaped must be fixed.
Output shape: (batch_size,) + target_shape.

See Also

Other core layers: layer_activation(), layer_activity_regularization(), layer_attention(), layer_dense_features(), layer_dense(), layer_dropout(), layer_flatten(), layer_input(), layer_lambda(), layer_masking(), layer_permute(), layer_repeat_vector()

Description

Separable convolutions consist in first performing a depthwise spatial convolution (which acts on each input channel separately) followed by a pointwise convolution which mixes together the resulting output channels. The depth_multiplier argument controls how many output channels are generated per input channel in the depthwise step. Intuitively, separable convolutions can be understood as a way to factorize a convolution kernel into two smaller kernels, or as an extreme version of an Inception block.

Usage

layer_separable_conv_1d(
    object,
    filters,
    kernel_size,
    strides = 1,
    padding = "valid",
    data_format = "channels_last",
    dilation_rate = 1,
    depth_multiplier = 1,
    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  
Model or layer object

filters  
Integer, the dimensionality of the output space (i.e. the number of output filters in the convolution).

kernel_size  
An integer or list of 2 integers, specifying the width and height of the 2D convolution window. Can be a single integer to specify the same value for all spatial dimensions.

strides  
An integer or list of 2 integers, specifying the strides of the convolution along the width and height. Can be a single integer to specify the same value for all spatial dimensions. Specifying any stride value != 1 is incompatible with specifying any dilation_rate value != 1.

padding  
one of "valid" or "same" (case-insensitive).

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, 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  
an integer or list of 2 integers, specifying the dilation rate to use for dilated convolution. Can be a single integer to specify the same value for all spatial dimensions. Currently, specifying any dilation_rate value != 1 is incompatible with specifying any stride value != 1.

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 filters_in * depth_multiplier.

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.

depthwise_initializer  
Initializer for the depthwise kernel matrix.

pointwise_initializer  
Initializer for the pointwise kernel matrix.

bias_initializer  
Initializer for the bias vector.
layer_separable_conv_1d

depthwise_regularizer
   Regularizer function applied to the depthwise kernel matrix.
pointwise_regularizer
   Regularizer function applied to the pointwise kernel matrix.
bias_regularizer
   Regularizer function applied to the bias vector.
activity_regularizer
   Regularizer function applied to the output of the layer (its "activation").
depthwise_constraint
   Constraint function applied to the depthwise kernel matrix.
pointwise_constraint
   Constraint function applied to the pointwise kernel matrix.
bias_constraint
   Constraint function applied to the bias vector.
input_shape
   Dimensionality of the input (integer) not including the samples axis. This argument is required when using this layer as the first layer in a model.
batch_input_shape
   Shapes, including the batch size. For instance, batch_input_shape=c(10, 32) indicates that the expected input will be batches of 10 32-dimensional vectors. batch_input_shape=list(NULL, 32) indicates batches of an arbitrary number of 32-dimensional vectors.
batch_size
   Fixed batch size for layer
dtype
   The data type expected by the input, as a string (float32, float64, int32...)
name
   An 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.
trainable
   Whether the layer weights will be updated during training.
weights
   Initial weights for layer.

Input shape
   3D tensor with shape: (batch, channels, steps) if data_format=’channels_first’ or 3D tensor with shape: (batch, steps, channels) if data_format=’channels_last’.

Output shape
   3D tensor with shape: (batch, filters, new_steps) if data_format=’channels_first’ or 3D tensor with shape: (batch, new_steps, filters) if data_format=’channels_last’. new_steps values might have changed due to padding or strides.

See Also
   Other convolutional layers: layer_conv_1d_transpose(), layer_conv_1d(), layer_conv_2d_transpose(), layer_conv_2d(), layer_conv_3d_transpose(), layer_conv_3d(), layer_conv_lstm_2d(), layer_cropping_1d(), layer_cropping_2d(), layer_cropping_3d(), layer_depthwise_conv_2d(), layer_separable_conv_2d(), layer_upsampling_1d(), layer_upsampling_2d(), layer_upsampling_3d(), layer_zero_padding_1d(), layer_zero_padding_2d(), layer_zero_padding_3d()
layer_separable_conv_2d

Separable 2D convolution.

Description

Separable convolutions consist in first performing a depthwise spatial convolution (which acts on each input channel separately) followed by a pointwise convolution which mixes together the resulting output channels. The depth_multiplier argument controls how many output channels are generated per input channel in the depthwise step. Intuitively, separable convolutions can be understood as a way to factorize a convolution kernel into two smaller kernels, or as an extreme version of an Inception block.

Usage

layer_separable_conv_2d(
    object,
    filters,
    kernel_size,
    strides = c(1, 1),
    padding = "valid",
    data_format = NULL,
    dilation_rate = 1,
    depth_multiplier = 1,
    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,
    input_shape = NULL,
    batch_input_shape = NULL,
    batch_size = NULL,
    dtype = NULL,
    name = NULL,
    trainable = NULL,
    weights = NULL
)
**Arguments**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>object</td>
<td>Model or layer object</td>
</tr>
<tr>
<td>filters</td>
<td>Integer, the dimensionality of the output space (i.e. the number of output filters in the convolution).</td>
</tr>
<tr>
<td>kernel_size</td>
<td>An integer or list of 2 integers, specifying the width and height of the 2D convolution window. Can be a single integer to specify the same value for all spatial dimensions.</td>
</tr>
<tr>
<td>strides</td>
<td>An integer or list of 2 integers, specifying the strides of the convolution along the width and height. Can be a single integer to specify the same value for all spatial dimensions. Specifying any stride value (!= 1) is incompatible with specifying any (\text{dilation} _\text{rate}) value (!= 1).</td>
</tr>
<tr>
<td>padding</td>
<td>one of &quot;valid&quot; or &quot;same&quot; (case-insensitive).</td>
</tr>
<tr>
<td>data_format</td>
<td>A string, one of \text{channels} _\text{last} (default) or \text{channels} _\text{first}. The ordering of the dimensions in the inputs. \text{channels} _\text{last} corresponds to inputs with shape (batch, height, width, channels) while \text{channels} _\text{first} corresponds to inputs with shape (batch, channels, height, width). It defaults to the \text{image} _\text{data} _\text{format} value found in your Keras config file at \texttt{~/.keras/keras.json}. If you never set it, then it will be &quot;channels_last&quot;.</td>
</tr>
<tr>
<td>dilation_rate</td>
<td>an integer or list of 2 integers, specifying the dilation rate to use for dilated convolution. Can be a single integer to specify the same value for all spatial dimensions. Currently, specifying any (\text{dilation} _\text{rate}) value (!= 1) is incompatible with specifying any stride value (!= 1).</td>
</tr>
<tr>
<td>depth_multiplier</td>
<td>The number of depthwise convolution output channels for each input channel. The total number of depthwise convolution output channels will be equal to (\text{filters} _\text{in} \ast \text{depth} _\text{multiplier}).</td>
</tr>
<tr>
<td>activation</td>
<td>Activation function to use. If you don’t specify anything, no activation is applied (i.e. &quot;linear&quot; activation: (a(x) = x)).</td>
</tr>
<tr>
<td>use_bias</td>
<td>Boolean, whether the layer uses a bias vector.</td>
</tr>
<tr>
<td>depthwise_initializer</td>
<td>Initializer for the depthwise kernel matrix.</td>
</tr>
<tr>
<td>pointwise_initializer</td>
<td>Initializer for the pointwise kernel matrix.</td>
</tr>
<tr>
<td>bias_initializer</td>
<td>Initializer for the bias vector.</td>
</tr>
<tr>
<td>depthwise_regularizer</td>
<td>Regularizer function applied to the depthwise kernel matrix.</td>
</tr>
<tr>
<td>pointwise_regularizer</td>
<td>Regularizer function applied to the pointwise kernel matrix.</td>
</tr>
<tr>
<td>bias_regularizer</td>
<td>Regularizer function applied to the bias vector.</td>
</tr>
<tr>
<td>activity_regularizer</td>
<td>Regularizer function applied to the output of the layer (its &quot;activation&quot;).</td>
</tr>
</tbody>
</table>
depthwise_constraint
Constraint function applied to the depthwise kernel matrix.

pointwise_constraint
Constraint function applied to the pointwise kernel matrix.

bias_constraint
Constraint function applied to the bias vector.

input_shape
Dimensionality of the input (integer) not including the samples axis. This argument is required when using this layer as the first layer in a model.

batch_input_shape
Shapes, including the batch size. For instance, batch_input_shape=c(10, 32) indicates that the expected input will be batches of 10 32-dimensional vectors. batch_input_shape=list(NULL, 32) indicates batches of an arbitrary number of 32-dimensional vectors.

batch_size
Fixed batch size for layer.

dtype
The data type expected by the input, as a string (float32, float64, int32...)

name
An 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.

trainable
Whether the layer weights will be updated during training.

weights
Initial weights for layer.

Input shape

4D tensor with shape: (batch, channels, rows, cols) if data_format=’channels_first’ or 4D tensor with shape: (batch, rows, cols, channels) if data_format=’channels_last’.

Output shape

4D tensor with shape: (batch, filters, new_rows, new_cols) if data_format=’channels_first’ or 4D tensor with shape: (batch, new_rows, new_cols, filters) if data_format=’channels_last’. rows and cols values might have changed due to padding.

See Also

Other convolutional layers: layer_conv_1d_transpose(), layer_conv_1d(), layer_conv_2d_transpose(), layer_conv_2d(), layer_conv_3d_transpose(), layer_conv_3d(), layer_conv_lstm_2d(), layer_cropping_1d(), layer_cropping_2d(), layer_cropping_3d(), layer_depthwise_conv_2d(), layer_separable_conv_1d(), layer_upsampling_1d(), layer_upsampling_2d(), layer_upsampling_3d(), layer_zero_padding_1d(), layer_zero_padding_2d(), layer_zero_padding_3d()
layer_simple_rnn  

**Fully-connected RNN where the output is to be fed back to input.**

**Description**

Fully-connected RNN where the output is to be fed back to input.

**Usage**

```r
layer_simple_rnn(
  object,
  units,
  activation = "tanh",
  use_bias = TRUE,
  return_sequences = FALSE,
  return_state = FALSE,
  go_backwards = FALSE,
  stateful = FALSE,
  unroll = FALSE,
  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,
  input_shape = NULL,
  batch_input_shape = NULL,
  batch_size = NULL,
  dtype = NULL,
  name = NULL,
  trainable = NULL,
  weights = NULL
)
```

**Arguments**

- **object**: Model or layer object
- **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, whether the layer uses a bias vector.

return_sequences
  Boolean. 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.

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.

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 the output of the layer (its "activation").

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.

input_shape
  Dimensionality of the input (integer) not including the samples axis. This argument is required when using this layer as the first layer in a model.

batch_input_shape
  Shapes, including the batch size. For instance, batch_input_shape=c(10, 32) indicates that the expected input will be batches of 10 32-dimensional vectors.
layer_simple_rnn

batch_input_shape=list(NULL,32) indicates batches of an arbitrary number of 32-dimensional vectors.

**batch_size** Fixed batch size for layer

**dtype** The data type expected by the input, as a string (float32, float64, int32...)

**name** An 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.

**trainable** Whether the layer weights will be updated during training.

**weights** Initial weights for layer.

**Input shapes**

3D tensor with shape (batch_size, timesteps, input_dim), (Optional) 2D tensors with shape (batch_size, output_dim).

**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, units).
- if `return_sequences`: 3D tensor with shape (batch_size, timesteps, units).
- else, 2D tensor with shape (batch_size, units).

**Masking**

This layer supports masking for input data with a variable number of timesteps. To introduce masks to your data, use an embedding layer with the `mask_zero` parameter set to TRUE.

**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. For intuition behind statefulness, there is a helpful blog post here: https://philipperemy.github.io/keras-stateful-lstm/

To enable statefulness:

- Specify `stateful = TRUE` in the layer constructor.
- Specify a fixed batch size for your model. For sequential models, pass `batch_input_shape = c(\ldots)` to the first layer in your model. For functional models with 1 or more Input layers, pass `batch_shape = c(\ldots)` to all the first layers in your model. This is the expected shape of your inputs including the batch size. It should be a vector of integers, e.g. `c(32, 10, 100)`. For dimensions which can vary (are not known ahead of time), use `NULL` in place of an integer, e.g. `c(32, NULL, NULL).
- Specify `shuffle = FALSE` when calling `fit()`.

To reset the states of your model, call `reset_states()` on either a specific layer, or on your entire model.
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.

You can specify the initial state of RNN layers numerically by calling reset_states with the keyword argument states. The value of states should be a numpy array or list of numpy arrays representing the initial state of the RNN layer.

References

- A Theoretically Grounded Application of Dropout in Recurrent Neural Networks

See Also

Other recurrent layers: layer_cudnn_gru(), layer_cudnn_lstm(), layer_gru(), layer_lstm()

layer_spatial_dropout_1d

Spatial 1D version of Dropout.

Description

This version 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, layer_spatial_dropout_1d will help promote independence between feature maps and should be used instead.

Usage

layer_spatial_dropout_1d(
    object,
    rate,
    batch_size = NULL,
    name = NULL,
    trainable = NULL,
    weights = NULL
)

Arguments

object     Model or layer object
rate       float between 0 and 1. Fraction of the input units to drop.
batch_size Fixed batch size for layer
name       An 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.
layer_spatial_dropout_2d

trainable  Whether the layer weights will be updated during training.
weights    Initial weights for layer.

Input shape

3D tensor with shape: (samples, timesteps, channels)

Output shape

Same as input

References

- Efficient Object Localization Using Convolutional Networks

See Also

Other dropout layers: layer_dropout(), layer_spatial_dropout_2d(), layer_spatial_dropout_3d()

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, layer_spatial_dropout_2d will help promote independence between feature maps and should be used instead.

Usage

layer_spatial_dropout_2d(
  object,
  rate,
  data_format = NULL,
  batch_size = NULL,
  name = NULL,
  trainable = NULL,
  weights = NULL
)
layer_spatial_dropout_3d

Arguments

- object: Model or layer object
- 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".
- batch_size: Fixed batch size for layer
- name: An 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.
- trainable: Whether the layer weights will be updated during training.
- weights: Initial weights for layer.

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

References

- Efficient Object Localization Using Convolutional Networks

See Also

Other dropout layers: layer_dropout(), layer_spatial_dropout_1d(), layer_spatial_dropout_3d()

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, layer_spatial_dropout_3d will help promote independence between feature maps and should be used instead.
layer_spatial_dropout_3d

Usage

layer_spatial_dropout_3d(
    object,
    rate,
    data_format = NULL,
    batch_size = NULL,
    name = NULL,
    trainable = NULL,
    weights = NULL
)

Arguments

object Model or layer object
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".
batch_size Fixed batch size for layer
name An 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.
trainable Whether the layer weights will be updated during training.
weights Initial weights for layer.

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

References

- Efficient Object Localization Using Convolutional Networks

See Also

Other dropout layers: layer_dropout(), layer_spatial_dropout_1d(), layer_spatial_dropout_2d()
layer_subtract  

**Layer that subtracts two inputs.**

**Description**

It takes as input a list of tensors of size 2, both of the same shape, and returns a single tensor, \((\text{inputs}[1] - \text{inputs}[2])\), also of the same shape.

**Usage**

```python
layer_subtract(
    inputs,
    batch_size = NULL,
    dtype = NULL,
    name = NULL,
    trainable = NULL,
    weights = NULL
)
```

**Arguments**

- **inputs**: A list of input tensors (exactly 2).
- **batch_size**: Fixed batch size for layer.
- **dtype**: The data type expected by the input, as a string (float32, float64, int32...)
- **name**: An 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.
- **trainable**: Whether the layer weights will be updated during training.
- **weights**: Initial weights for layer.

**Value**

A tensor, the difference of the inputs.

**See Also**

Other merge layers: `layer_add()`, `layer_average()`, `layer_concatenate()`, `layer_dot()`, `layer_maximum()`, `layer_minimum()`, `layer_multiply()`
### Description

This layer has basic options for managing text in a Keras model. It transforms a batch of strings (one sample = one string) into either a list of token indices (one sample = 1D tensor of integer token indices) or a dense representation (one sample = 1D tensor of float values representing data about the sample’s tokens).

### Usage

```r
layer_text_vectorization(
  object,
  max_tokens = NULL,
  standardize = "lower_and_strip_punctuation",
  split = "whitespace",
  ngrams = NULL,
  output_mode = c("int", "binary", "count", "tf-idf"),
  output_sequence_length = NULL,
  pad_to_max_tokens = tf_version() < "2.6",
  vocabulary = NULL,
  ...
)
```

### Arguments

- **object**: Model or layer object
- **max_tokens**: The maximum size of the vocabulary for this layer. If NULL, there is no cap on the size of the vocabulary.
- **standardize**: Optional specification for standardization to apply to the input text. Values can be NULL (no standardization), "lower_and_strip_punctuation" (lowercase and remove punctuation) or a Callable. Default is "lower_and_strip_punctuation".
- **split**: Optional specification for splitting the input text. Values can be NULL (no splitting), "split_on_whitespace" (split on ASCII whitespace), or a Callable. Default is "split_on_whitespace".
- **ngrams**: Optional specification for ngrams to create from the possibly-split input text. Values can be NULL, an integer or a 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", "binary", "count" or "tfidf", which control the outputs as follows:
  - "int": Outputs integer indices, one integer index per split string token.
"binary": Outputs a single int array per batch, of either \texttt{vocab\_size} or \texttt{max\_tokens} size, containing 1s in all elements where the token mapped to that index exists at least once in the batch item.

- "count": As "binary", but the int array contains a count of the number of times the token at that index appeared in the batch item.

- "tfidf": As "binary", but the TF-IDF algorithm is applied to find the value in each token slot.

\texttt{output\_sequence\_length}

Only valid in "int" mode. If set, the output will have its time dimension padded or truncated to exactly \texttt{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.

\texttt{pad\_to\_max\_tokens}

Only valid in "binary", "count", and "tfidf" modes. If \texttt{TRUE}, the output will have its feature axis padded to \texttt{max\_tokens} even if the number of unique tokens in the vocabulary is less than \texttt{max\_tokens}, resulting in a tensor of shape (batch\_size, max\_tokens) regardless of vocabulary size. Defaults to \texttt{FALSE} in TF 2.6+, \texttt{TRUE} in prior version.

\texttt{vocabulary}

An optional list of vocabulary terms, or a path to a text file containing a vocabulary to load into this layer. The file should contain one token per line. If the list or file contains the same token multiple times, an error will be thrown.

... Not used.

\textbf{Details}

The processing of each sample contains the following steps:

1. standardize each sample (usually lowercasing + punctuation stripping)
2. split each sample 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 sample using this index, either into a vector of ints or a dense float vector.

\textbf{Description}

Repeats each temporal step \texttt{size} times along the time axis.
Usage

layer_upsampling_2d(
    object,
    size = 2L,
    batch_size = NULL,
    name = NULL,
    trainable = NULL,
    weights = NULL
)

Arguments

- **object**: Model or layer object
- **size**: integer. Upsampling factor.
- **batch_size**: Fixed batch size for layer
- **name**: An 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.
- **trainable**: Whether the layer weights will be updated during training.
- **weights**: Initial weights for layer.

Input shape

3D tensor with shape: (batch, steps, features).

Output shape

3D tensor with shape: (batch, upsampled_steps, features).

See Also

Other convolutional layers: layer_conv_1d_transpose(), layer_conv_1d(), layer_conv_2d_transpose(),
layer_conv_2d(), layer_conv_3d_transpose(), layer_conv_3d(), layer_conv_lstm_2d(),
layer_cropping_1d(), layer_cropping_2d(), layer_cropping_3d(), layer_depthwise_conv_2d(),
layer_separable_conv_1d(), layer_separable_conv_2d(), layer_upsampling_2d(), layer_upsampling_3d(),
layer_zero_padding_1d(), layer_zero_padding_2d(), layer_zero_padding_3d()

layer_upsampling_2d  Upsampling layer for 2D inputs.

Description

Repeats the rows and columns of the data by size[[0]] and size[[1]] respectively.
Usage

```r
layer_upsampling_2d(
  object,
  size = c(2L, 2L),
  data_format = NULL,
  interpolation = "nearest",
  batch_size = NULL,
  name = NULL,
  trainable = NULL,
  weights = NULL
)
```

Arguments

- `object`: Model or layer object.
- `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, 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".
- `interpolation`: A string, one of `nearest` or `bilinear`. Note that CNTK does not support yet the bilinear upsampling and that with Theano, only size=(2, 2) is possible.
- `batch_size`: Fixed batch size for layer.
- `name`: An 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.
- `trainable`: Whether the layer weights will be updated during training.
- `weights`: Initial weights for layer.

Input shape

4D tensor with shape:

- If `data_format` is "channels_last": `(batch, rows, cols, channels)`
- If `data_format` is "channels_first": `(batch, channels, rows, cols)`

Output shape

4D tensor with shape:

- If `data_format` is "channels_last": `(batch, upsampled_rows, upsampled_cols, channels)`
- If `data_format` is "channels_first": `(batch, channels, upsampled_rows, upsampled_cols)`
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 = c(2L, 2L, 2L),
  data_format = NULL,
  batch_size = NULL,
  name = NULL,
  trainable = NULL,
  weights = NULL
)

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>object</td>
<td>Model or layer object</td>
</tr>
<tr>
<td>size</td>
<td>int, or list of 3 integers. The upsampling factors for dim1, dim2 and dim3.</td>
</tr>
<tr>
<td>data_format</td>
<td>A string, one of channels_last (default) or channels_first. The ordering of</td>
</tr>
<tr>
<td></td>
<td>the dimensions in the inputs. channels_last corresponds to inputs with shape</td>
</tr>
<tr>
<td></td>
<td>(batch, spatial_dim1, spatial_dim2, spatial_dim3, channels) while</td>
</tr>
<tr>
<td></td>
<td>channels_first corresponds to inputs with shape (batch, channels, spatial_dim1,</td>
</tr>
<tr>
<td></td>
<td>spatial_dim2, spatial_dim3). It defaults to the image_data_format value found in</td>
</tr>
<tr>
<td></td>
<td>your Keras config file at ~/.keras/keras.json. If you never set it, then it</td>
</tr>
<tr>
<td></td>
<td>will be &quot;channels_last&quot;.</td>
</tr>
<tr>
<td>batch_size</td>
<td>Fixed batch size for layer</td>
</tr>
<tr>
<td>name</td>
<td>An optional name string for the layer. Should be unique in a model (do not</td>
</tr>
<tr>
<td></td>
<td>reuse the same name twice). It will be autogenerated if it isn’t provided.</td>
</tr>
<tr>
<td>trainable</td>
<td>Whether the layer weights will be updated during training.</td>
</tr>
<tr>
<td>weights</td>
<td>Initial weights for layer</td>
</tr>
</tbody>
</table>
layer_zero_padding_1d

Input shape

5D tensor with shape:

- If `data_format` is "channels_last": (batch, dim1, dim2, dim3, channels)
- If `data_format` is "channels_first": (batch, channels, dim1, dim2, dim3)

Output shape

5D tensor with shape:

- If `data_format` is "channels_last": (batch, upsampled_dim1, upsampled_dim2, upsampled_dim3, channels)
- If `data_format` is "channels_first": (batch, channels, upsampled_dim1, upsampled_dim2, upsampled_dim3)

See Also

Other convolutional layers: `layer_conv_1d_transpose()`, `layer_conv_1d()`, `layer_conv_2d_transpose()`, `layer_conv_2d()`, `layer_conv_3d_transpose()`, `layer_conv_3d()`, `layer_conv_lstm_2d()`, `layer_cropping_1d()`, `layer_cropping_2d()`, `layer_cropping_3d()`, `layer_depthwise_conv_2d()`, `layer_separable_conv_1d()`, `layer_separable_conv_2d()`, `layer_upsampling_1d()`, `layer_upsampling_2d()`, `layer_zero_padding_1d()`, `layer_zero_padding_2d()`, `layer_zero_padding_3d()`

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

```python
layer_zero_padding_1d(
    object,
    padding = 1L,
    batch_size = NULL,
    name = NULL,
    trainable = NULL,
    weights = NULL
)
```
layer_zero_padding_2d

Arguments

object: Model or layer object

padding: int, or list of int (length 2)
  • If int: How many zeros to add at the beginning and end of the padding dimension (axis 1).
  • If list of int (length 2): How many zeros to add at the beginning and at the end of the padding dimension ((left_pad, right_pad)).

batch_size: Fixed batch size for layer

name: An 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.

trainable: Whether the layer weights will be updated during training.

weights: Initial weights for layer.

Input shape

3D tensor with shape (batch, axis_to_pad, features)

Output shape

3D tensor with shape (batch, padded_axis, features)

See Also

Other convolutional layers: layer_conv_1d_transpose(), layer_conv_1d(), layer_conv_2d_transpose(), layer_conv_2d(), layer_conv_3d_transpose(), layer_conv_3d(), layer_conv_lstm_2d(), layer_cropping_1d(), layer_cropping_2d(), layer_cropping_3d(), layer_depthwise_conv_2d(), layer_separable_conv_1d(), layer_separable_conv_2d(), layer_upsampling_1d(), layer_upsampling_2d(), layer_upsampling_3d(), layer_zero_padding_2d(), layer_zero_padding_3d()

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 = c(1L, 1L),
  data_format = NULL,
  batch_size = NULL,
  name = NULL,
layer_zero_padding_2d

trainable = NULL,
weights = NULL
)

Arguments

object Model or layer object
padding int, or list of 2 ints, or list of 2 lists of 2 ints.
  • If int: the same symmetric padding is applied to width and height.
  • 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, 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".
batch_size Fixed batch size for layer
name An 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.
trainable Whether the layer weights will be updated during training.
weights Initial weights for layer.

Input shape

4D tensor with shape:
  • If data_format is "channels_last": (batch, rows, cols, channels)
  • If data_format is "channels_first": (batch, channels, rows, cols)

Output shape

4D tensor with shape:
  • If data_format is "channels_last": (batch, padded_rows, padded_cols, channels)
  • If data_format is "channels_first": (batch, channels, padded_rows, padded_cols)

See Also

Other convolutional layers: layer_conv_1d_transpose(), layer_conv_1d(), layer_conv_2d_transpose(),
layer_conv_2d(), layer_conv_3d_transpose(), layer_conv_3d(), layer_conv_lstm_2d(),
layer_cropping_1d(), layer_cropping_2d(), layer_cropping_3d(), layer_depthwise_conv_2d(),
layer_separable_conv_1d(), layer_separable_conv_2d(), layer_upsampling_1d(), layer_upsampling_2d(),
layer_upsampling_3d(), layer_zero_padding_1d(), layer_zero_padding_2d(), layer_zero_padding_3d()
**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**

```r
layer_zero_padding_3d(
  object,
  padding = c(1L, 1L, 1L),
  data_format = NULL,
  batch_size = NULL,
  name = NULL,
  trainable = NULL,
  weights = NULL
)
```

**Arguments**

- **object**: Model or layer object
- **padding**: int, or list of 3 ints, or list of 3 lists of 2 ints.
  - If int: the same symmetric padding is applied to width and height.
  - If list of 3 ints: interpreted as three different symmetric padding values:
    (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, right_dim2_pad), (left_dim3_pad, right_dim3_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, spatial_dim1, spatial_dim2, spatial_dim3, channels) while `channels_first` corresponds to inputs with shape (batch, channels, spatial_dim1, spatial_dim2, spatial_dim3). 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".
- **batch_size**: Fixed batch size for layer
- **name**: An 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.
- **trainable**: Whether the layer weights will be updated during training.
- **weights**: Initial weights for layer.

**Input shape**

5D tensor with shape:

- If `data_format` is "channels_last": (batch, first_axis_to_pad, second_axis_to_pad, third_axis_to_pad, depth)
- If `data_format` is "channels_first": (batch, depth, first_axis_to_pad, second_axis_to_pad, third_axis_to_pad)
Output shape

5D tensor with shape:

- If `data_format` is "channels_last": (batch, first_padded_axis, second_padded_axis, third_axis_to_pad, depth)
- If `data_format` is "channels_first": (batch, depth, first_padded_axis, second_padded_axis, third_axis_to_pad)

See Also

Other convolutional layers: `layer_conv_1d_transpose()`, `layer_conv_1d()`, `layer_conv_2d_transpose()`, `layer_conv_2d()`, `layer_conv_3d_transpose()`, `layer_conv_3d()`, `layer_conv_lstm_2d()`, `layer_cropping_1d()`, `layer_cropping_2d()`, `layer_cropping_3d()`, `layer_separable_conv_1d()`, `layer_separable_conv_2d()`, `layer_depthwise_conv_2d()`, `layer_upsampling_1d()`, `layer_upsampling_2d()`, `layer_upsampling_3d()`, `layer_zero_padding_1d()`, `layer_zero_padding_2d()`

---

### Description

Loss functions

### Usage

```r
loss_binary_crossentropy(
  y_true,
  y_pred,
  from_logits = FALSE,
  label_smoothing = 0,
  axis = -1L,
  ...,
  reduction = "auto",
  name = "binary_crossentropy"
)

loss_categorical_crossentropy(
  y_true,
  y_pred,
  from_logits = FALSE,
  label_smoothing = 0L,
  axis = -1L,
  ...,
  reduction = "auto",
  name = "categorical_crossentropy"
)

loss_categorical_hinge(
  y_true,
  ...,
  ...)```
loss-functions

```python
    y_pred,
    ...,  
    reduction = "auto",
    name = "categorical_hinge"
)

loss_cosine_similarity(
    y_true,
    y_pred,
    axis = -1L,
    ...,  
    reduction = "auto",
    name = "cosine_similarity"
)

loss_hinge(y_true, y_pred, ..., reduction = "auto", name = "hinge")

loss_huber(
    y_true,
    y_pred,
    delta = 1,
    ...,  
    reduction = "auto",
    name = "huber_loss"
)

loss_kullback_leibler_divergence(
    y_true,
    y_pred,
    ...,  
    reduction = "auto",
    name = "kl_divergence"
)

loss_kl_divergence(
    y_true,
    y_pred,
    ...,  
    reduction = "auto",
    name = "kl_divergence"
)

loss_logcosh(y_true, y_pred, ..., reduction = "auto", name = "log_cosh")

loss_mean_absolute_error(
    y_true,
    y_pred,
    ...,  
```


```r
loss_mean_absolute_error(  
  reduction = "auto",  
  name = "mean_absolute_error"
)

loss_mean_absolute_percentage_error(  
  y_true,  
  y_pred,  
  ...,  
  reduction = "auto",  
  name = "mean_absolute_percentage_error"
)

loss_mean_squared_error(  
  y_true,  
  y_pred,  
  ...,  
  reduction = "auto",  
  name = "mean_squared_error"
)

loss_mean_squared_logarithmic_error(  
  y_true,  
  y_pred,  
  ...,  
  reduction = "auto",  
  name = "mean_squared_logarithmic_error"
)

loss_poisson(y_true, y_pred, ..., reduction = "auto", name = "poisson")

loss_sparse_categorical_crossentropy(  
  y_true,  
  y_pred,  
  from_logits = FALSE,  
  axis = -1L,  
  ...,  
  reduction = "auto",  
  name = "sparse_categorical_crossentropy"
)

loss_squared_hinge(  
  y_true,  
  y_pred,  
  ...,  
  reduction = "auto",  
  name = "squared_hinge"
)
```
Arguments

- **y_true**: Ground truth values. shape = [batch_size, d1, .. dN].
- **y_pred**: The predicted values. shape = [batch_size, d1, .. dN]. (Tensor of the same shape as y_true)
- **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**: The axis along which to compute crossentropy (the features axis). Axis is 1-based (e.g, first axis is axis=1). Defaults to -1 (the last axis).
- **...**: Additional arguments passed on to the Python callable (for forward and backwards compatibility).
- **reduction**: Only applicable if y_true and y_pred are missing. Type of keras$losses$Reduction to apply to loss. Default value is AUTO. AUTO indicates that the reduction option will be determined by the usage context. For almost all cases this defaults to SUM_OVER_BATCH_SIZE. When used with tf$distribute$Strategy, outside of built-in training loops such as compile and fit, using AUTO or SUM_OVER_BATCH_SIZE will raise an error. Please see this custom training tutorial for more details.
- **name**: Only applicable if y_true and y_pred are missing. Optional name for the Loss instance.
- **delta**: A float, the point where the Huber loss function changes from a quadratic to linear.

Details

Loss functions for model training. These are typically supplied in the loss parameter of the compile.keras.engine.training.Model() function.

Value

If called with y_true and y_pred, then the corresponding loss is evaluated and the result returned (as a tensor). Alternatively, if y_true and y_pred are missing, then a callable is returned that will compute the loss function and, by default, reduce the loss to a scalar tensor; see the reduction parameter for details. (The callable is a typically a class instance that inherits from keras$losses$Loss).

binary_crossentropy

Computes the binary crossentropy loss.

label_smoothing details: 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.
categorical_crossentropy

Computes the categorical crossentropy loss.

When using the categorical_crossentropy loss, your targets should be in categorical format (e.g. if you have 10 classes, the target for each sample should be a 10-dimensional vector that is all-zeros except for a 1 at the index corresponding to the class of the sample). In order to convert integer targets into categorical targets, you can use the Keras utility function `to_categorical()`: categorical_labels <- to_categorical(int_labels, num_classes = NULL)

huber

Computes Huber loss value. For each value x in error = y_true - y_pred:

loss = 0.5 * x^2 if |x| <= d
loss = d * |x| - 0.5 * d^2 if |x| > d

where d is delta. See: https://en.wikipedia.org/wiki/Huber_loss

log_cosh

Logarithm of the hyperbolic cosine of the prediction error.

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. However, it may return NaNs if the intermediate value cosh(y_pred - y_true) is too large to be represented in the chosen precision.

See Also

compile.keras.engine.training.Model().loss_binary_crossentropy()

---

loss_cosine_proximity  (Deprecated) loss_cosine_proximity

Description

loss_cosine_proximity is deprecated and will be removed in a future version. It has been renamed to loss_cosine_similarity().

Usage

loss_cosine_proximity(...)  

Arguments

... passed on to loss_cosine_similarity()
### make_sampling_table

Generates a word rank-based probabilistic sampling table.

**Description**
Generates a word rank-based probabilistic sampling table.

**Usage**
```r
make_sampling_table(size, sampling_factor = 1e-05)
```

**Arguments**
- `size`: Int, number of possible words to sample.
- `sampling_factor`: The sampling factor in the word2vec formula.

**Details**
Used for generating the sampling_table argument for `skipgrams()`. sampling_table[[i]] is the probability of sampling the word i-th most common word in a dataset (more common words should be sampled less frequently, for balance).

The sampling probabilities are generated according to the sampling distribution used in word2vec:

\[ p(\text{word}) = \min(1, \sqrt{\frac{\text{word}_\text{frequency}}{\text{sampling}_\text{factor}}} / \frac{\text{word}_\text{frequency}}{\text{sampling}_\text{factor}}) \]

We assume that the word frequencies follow Zipf’s law (s=1) to derive a numerical approximation of frequency(rank):

\[ \text{frequency(rank)} \sim \frac{1}{\text{rank} \times (\log(\text{rank}) + \gamma) + 1/2 -1/(12*\text{rank})} \]

where \( \gamma \) is the Euler-Mascheroni constant.

**Value**
An array of length `size` where the ith entry is the probability that a word of rank i should be sampled.

**Note**
The word2vec formula is:

\[ p(\text{word}) = \min(1, \sqrt{\frac{\text{word}\_frequency}{\text{sampling}\_factor}} / (\text{word}\_frequency / \text{sampling}\_factor)) \]

**See Also**
Other text preprocessing: `pad_sequences()`, `skipgrams()`, `text_hashing_trick()`, `text_one_hot()`, `text_to_word_sequence()`
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_.*.

**Arguments**
- name: (Optional) string name of the metric instance.
- dtype: (Optional) data type of the metric result.

**Value**
A (subclassed) Metric instance that can be passed directly to compile(metrics = ), or used as a standalone object. See ?Metric for example usage.

**Usage with compile**

```r
model %>% compile(
  optimizer = 'sgd',
  loss = 'mse',
  metrics = list(metric_SOME_METRIC(), metric_SOME_OTHER_METRIC())
)
```

**Standalone usage**

```r
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()), '
')
  m$reset_state()
}
```

**Custom Metric (subclass)**

To be implemented by subclasses:

- **initialize()**: All state variables should be created in this method by calling self$add_weight() like:
  ```r
  self$var <- self$add_weight(...)  
  ```
- **update_state()**: Has all updates to the state variables like:
self$var$assign_add(...)  
  • result(): Computes and returns a value for the metric from the state variables.

Example custom metric subclass:

```
metric_binary_true_positives(keras$metrics$Metric) %py_class% {
  initialize <- function(name = 'binary_true_positives', ...) {
    super$initialize(name = name, ...)
    self$true_positives <- self$add_weight(name = 'tp', initializer = 'zeros')
  }

  update_state <- function(y_true, y_pred, sample_weight = NULL) {
    y_true <- k_cast(y_true, "bool")
    y_pred <- k_cast(y_pred, "bool")
    values <- y_true & y_pred
    values <- k_cast(values, self$dtype)
    if (!is.null(sample_weight)) {
      sample_weight <- k_cast(sample_weight, self$dtype)
      sample_weight <- tf$broadcast_to(sample_weight, values$shape)
      values <- values * sample_weight
    }
    self$true_positives$assign_add(tf$reduce_sum(values))
  }

  result <- function()
    self$true_positives
  }

model %>% compile(..., metrics = list(metric_binary_true_positives()))
```

---

**metric_accuracy**  
*Calculates how often predictions equal labels*

**Description**

Calculates how often predictions equal labels

**Usage**

```
metric_accuracy(..., name = NULL, dtype = NULL)
```

**Arguments**

- ...  
  Passed on to the underlying metric. Used for forwards and backwards compatibility.
- name  
  (Optional) string name of the metric instance.
- dtype  
  (Optional) data type of the metric result.
Details

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.

Value

A (subclassed) Metric instance that can be passed directly to compile(metrics = ), or used as a standalone object. See ?Metric for example usage.

See Also

Other metrics: custom_metric(), metric_auc(), metric_binary_accuracy(), metric_binary_crossentropy(), metric_categorical_accuracy(), metric_categorical_crossentropy(), metric_categorical_hinge(), metric_cosine_similarity(), metric_false_negatives(), metric_false_positives(), metric_hinge(), metric_kullback_leibler_divergence(), metric_logcosh_error(), metric_mean_absolute_error(), metric_mean_absolute_percentage_error(), metric_mean_iou(), metric_mean_relative_error(), metric_mean_squared_error(), metric_mean_squared_logarithmic_error(), metric_mean_tensor(), metric_mean_wrapper(), metric_mean(), metric_poisson(), metric_precision_at_recall(), metric_precision(), metric_recall_at_precision(), 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_auc | Approximates the AUC (Area under the curve) of the ROC or PR curves |

Description

Approximates the AUC (Area under the curve) of the ROC or PR curves

Usage

```r
metric_auc(
  ..., 
  num_thresholds = 200L, 
  curve = "ROC", 
  summation_method = "interpolation", 
  thresholds = NULL, 
  multi_label = FALSE, 
  num_labels = NULL, 
  label_weights = NULL, 
  from_logits = FALSE, 
  name = NULL, 
  dtype = NULL
)
```
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.
metric_binary_accuracy

Calculates how often predictions match binary labels

Description

Calculates how often predictions match binary labels
**Usage**

```python
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.
- `...`: Passed on to the underlying metric. Used for forwards and backwards compatibility.
- `name`: (Optional) string name of the metric instance.
- `dtype`: (Optional) data type of the metric result.

**Details**

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.

**Value**

If `y_true` and `y_pred` are missing, a (subclassed) Metric instance is returned. The Metric object can be passed directly to `compile(metrics = )` or used as a standalone object. See `?Metric` for example usage.

Alternatively, if called with `y_true` and `y_pred` arguments, then the computed case-wise values for the mini-batch are returned directly.

**See Also**

Other metrics: `custom_metric()`, `metric_accuracy()`, `metric_auc()`, `metric_binary_crossentropy()`, `metric_categorical_accuracy()`, `metric_categorical_crossentropy()`, `metric_categorical_hinge()`, `metric_cosine_similarity()`, `metric_false_negatives()`, `metric_false_positives()`, `metric_hinge()`, `metric_kullback_leibler_divergence()`, `metric_logcosh_error()`, `metric_mean_absolute_error()`, `metric_mean_absolute_percentage_error()`, `metric_mean_iou()`, `metric_mean_relative_error()`, `metric_mean_squared_error()`, `metric_mean_squared_logarithmic_error()`, `metric_mean_tensor()`, `metric_mean_wrapper()`, `metric_mean()`, `metric_poisson()`, `metric_precision_at_recall()`, `metric_precision()`, `metric_recall_at_precision()`, `metric_recall()`, `metric_root_mean_squared_error()`, `metric_sensitivity_at_specificity()`, `metric_sparse_categorical_accuracy()`, `metric_sparse_categorical_crossentropy()`
**metric_binary_crossentropy**

Computes the crossentropy metric between the labels and predictions

**Description**

Computes the crossentropy metric between the labels and predictions

**Usage**

```r
metric_binary_crossentropy(
  y_true, y_pred,
  from_logits = FALSE,
  label_smoothing = 0,
  axis = -1L,
  ..., 
  name = "binary_crossentropy",
  dtype = NULL
)
```

**Arguments**

- `y_true` Tensor of 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) (1-based) Defaults to -1. The dimension along which the metric is computed.
- `...` Passed on to the underlying metric. Used for forwards and backwards compatibility.
- `name` (Optional) string name of the metric instance.
- `dtype` (Optional) data type of the metric result.

**Details**

This is the crossentropy metric class to be used when there are only two label classes (0 and 1).
metric_categorical_accuracy

Calculates how often predictions match one-hot labels

Description

Calculates how often predictions match one-hot labels

Usage

```r
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.
- `...`: Passed on to the underlying metric. Used for forwards and backwards compatibility.
- `name`: (Optional) string name of the metric instance.
- `dtype`: (Optional) data type of the metric result.
metric_categorical_crossentropy

Computes the crossentropy metric between the labels and predictions

Description

Computes the crossentropy metric between the labels and predictions

Usage

```r
metric_categorical_crossentropy(
  y_true,
  y_pred,
  from_logits = FALSE,
```

Details

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 `tf.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.

Value

If `y_true` and `y_pred` are missing, a (subclassed) Metric instance is returned. The Metric object can be passed directly to `compile(metrics = )` or used as a standalone object. See `?Metric` for example usage.

Alternatively, if called with `y_true` and `y_pred` arguments, then the computed case-wise values for the mini-batch are returned directly.

See Also

Other metrics: `custom_metric()`, `metric_accuracy()`, `metric_auc()`, `metric_binary_accuracy()`, `metric_binary_crossentropy()`, `metric_categorical_crossentropy()`, `metric_categorical_hinge()`, `metric_cosine_similarity()`, `metric_false_negatives()`, `metric_false_positives()`, `metric_hinge()`, `metric_kullback_leibler_divergence()`, `metric_logcosh_error()`, `metric_mean_absolute_error()`, `metric_mean_absolute_percentage_error()`, `metric_mean_iou()`, `metric_mean_relative_error()`, `metric_mean_squared_error()`, `metric_mean_squared_logarithmic_error()`, `metric_mean_tensor()`, `metric_mean_wrapper()`, `metric_mean()`, `metric_poisson()`, `metric_precision_at_recall()`, `metric_precision()`, `metric_recall_at_precision()`, `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()`
Label smoothing = 0,
axis = -1L,
...
name = "categorical_crossentropy",
dtype = NULL
)

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>y_true</td>
<td>Tensor of true targets.</td>
</tr>
<tr>
<td>y_pred</td>
<td>Tensor of predicted targets.</td>
</tr>
<tr>
<td>from_logits</td>
<td>(Optional) Whether output is expected to be a logits tensor. By default, we consider that output encodes a probability distribution.</td>
</tr>
<tr>
<td>label_smoothing</td>
<td>(Optional) Float in [0, 1]. When &gt; 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</td>
</tr>
<tr>
<td>axis</td>
<td>(Optional) (1-based) Defaults to -1. The dimension along which the metric is computed.</td>
</tr>
<tr>
<td>...</td>
<td>Passed on to the underlying metric. Used for forwards and backwards compatibility.</td>
</tr>
<tr>
<td>name</td>
<td>(Optional) string name of the metric instance.</td>
</tr>
<tr>
<td>dtype</td>
<td>(Optional) data type of the metric result.</td>
</tr>
</tbody>
</table>

Details

This is the crossentropy metric class to be used when there are multiple label classes (2 or more). Here we assume that labels are given as a one_hot representation. e.g., When labels values are c(2,0,1):

```r
y_true = rbind(c(0, 0, 1),
               c(1, 0, 0),
               c(0, 1, 0))
```

Value

If y_true and y_pred are missing, a (subclassed) Metric instance is returned. The Metric object can be passed directly to compile(metrics = ) or used as a standalone object. See ?Metric for example usage. Alternatively, if called with y_true and y_pred arguments, then the computed case-wise values for the mini-batch are returned directly.

See Also

Other metrics: custom_metric(), metric_accuracy(), metric_auc(), metric_binary_accuracy(), metric_binary_crossentropy(), metric_categorical_accuracy(), metric_categorical_hinge(), metric_cosine_similarity(), metric_false_negatives(), metric_false_positives(), metric_hinge(),...
metric_categorical_hinge

Computes the categorical hinge metric between \( y_{true} \) and \( y_{pred} \)

**Description**

Computes the categorical hinge metric between \( y_{true} \) and \( y_{pred} \)

**Usage**

```r
metric_categorical_hinge(..., name = NULL, dtype = NULL)
```

**Arguments**

- `...`: Passed on to the underlying metric. Used for forwards and backwards compatibility.
- `name` (Optional): string name of the metric instance.
- `dtype` (Optional): data type of the metric result.

**Value**

A (subclassed) Metric instance that can be passed directly to `compile(metrics = )`, or used as a standalone object. See `?Metric` for example usage.

**See Also**

Other metrics: `custom_metric()`, `metric_accuracy()`, `metric_auc()`, `metric_binary_accuracy()`, `metric_binary_crossentropy()`, `metric_categorical_accuracy()`, `metric_categorical_crossentropy()`, `metric_cosine_similarity()`, `metric_false_negatives()`, `metric_false_positives()`, `metric_hinge()`, `metric_kullback_leibler_divergence()`, `metric_logcosh_error()`, `metric_mean_absolute_error()`, `metric_mean_absolute_percentage_error()`, `metric_mean_iou()`, `metric_mean_relative_error()`, `metric_mean_squared_error()`, `metric_mean_squared_logarithmic_error()`, `metric_mean_tensor()`, `metric_mean_wrapper()`, `metric_mean()`, `metric_poisson()`, `metric_precision_at_recall()`, `metric_precision()`, `metric_recall_at_precision()`, `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_cosine_proximity**

*(Deprecated) metric_cosine_proximity*

**Description**

`metric_cosine_proximity()` is deprecated and will be removed in a future version. Please update your code to use `metric_cosine_similarity()` if possible. If you need the actual function and not a Metric object, (e.g., because you are using the intermediate computed values in a custom training loop before reduction), please use `loss_cosine_similarity()` or `tensorflow::tf$compat$v1$keras$metrics$cosine_proximity()`.

**Usage**

```r
metric_cosine_proximity(y_true, y_pred)
```

**Arguments**

- `y_true` Tensor of true targets.
- `y_pred` Tensor of predicted targets.

---

**metric_cosine_similarity**

*Computes the cosine similarity between the labels and predictions*

**Description**

Computes the cosine similarity between the labels and predictions

**Usage**

```r
metric_cosine_similarity(
  ..., 
  axis = -1L, 
  name = "cosine_similarity", 
  dtype = NULL
)
```

**Arguments**

- `...` Passed on to the underlying metric. Used for forwards and backwards compatibility.
- `axis` (Optional) (1-based) Defaults to -1. The dimension along which the metric is computed.
- `name` (Optional) string name of the metric instance.
- `dtype` (Optional) data type of the metric result.
Details

cosine similarity = (a . b) / ||a|| ||b||

See: Cosine Similarity.

This metric keeps the average cosine similarity between predictions and labels over a stream of data.

Value

A (subclassed) Metric instance that can be passed directly to compile(metrics = ), or used as a standalone object. See ?Metric for example usage.

Note

If you want to compute the cosine_similarity for each case in a mini-batch you can use loss_cosine_similarity().

See Also

Other metrics: custom_metric(), metric_accuracy(), metric_auc(), metric_binary_accuracy(), metric_binary_crossentropy(), metric_categorical_accuracy(), metric_categorical_crossentropy(), metric_categorical_hinge(), metric_false_negatives(), metric_false_positives(), metric_hinge(), metric_kullback_leibler_divergence(), metric_logcosh_error(), metric_mean_absolute_error(), metric_mean_absolute_percentage_error(), metric_mean_iou(), metric_mean_relative_error(), metric_mean_squared_error(), metric_mean_squared_logarithmic_error(), metric_mean_tensor(), metric_mean_wrapper(), metric_mean(), metric_poisson(), metric_precision_at_recall(), metric_precision(), metric_recall_at_precision(), 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_false_positives**

**Calculates the number of false positives**

**Description**

Calculates the number of false positives

**Usage**

```r
metric_false_positives(..., thresholds = NULL, name = NULL, dtype = NULL)
```
**Arguments**

Passed on to the underlying metric. Used for forwards and backwards compatibility.

thresholds (Optional) Defaults to 0.5. A float value or a 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). 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.

**Details**

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.

**Value**

A (subclassed) Metric instance that can be passed directly to `compile(metrics = )`, or used as a standalone object. See ?Metric for example usage.

**See Also**

Other metrics: `custom_metric()`, `metric_accuracy()`, `metric_auc()`, `metric_binary_accuracy()`, `metric_binary_crossentropy()`, `metric_categorical_accuracy()`, `metric_categorical_crossentropy()`, `metric_categorical_hinge()`, `metric_kullback_leibler_divergence()`, `metric_logcosh_error()`, `metric_mean_absolute_error()`, `metric_mean_absolute_percentage_error()`, `metric_mean_iou()`, `metric_mean_relative_error()`, `metric_mean_squared_error()`, `metric_mean_squared_logarithmic_error()`, `metric_mean_tensor()`, `metric_mean_wrapper()`, `metric_mean()`, `metric_poisson()`, `metric_precision_at_recall()`, `metric_precision()`, `metric_recall_at_precision()`, `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_hinge**

Computes the hinge metric between y_true and y_pred

**Description**

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` Tensor of true targets.
- `y_pred` Tensor of predicted targets.
- `...` Passed on to the underlying metric. Used for forwards and backwards compatibility.
- `name` (Optional) string name of the metric instance.
- `dtype` (Optional) data type of the metric result.

**Details**

\[
\text{loss} = \text{tf\_reduce\_mean}(\text{tf\_maximum}(1 - y_{\text{true}} \times y_{\text{pred}}, 0L), \text{axis}\_=-1L)
\]

**Value**

If `y_true` and `y_pred` are missing, a (subclassed) Metric instance is returned. The Metric object can be passed directly to `compile(metrics = )` or used as a standalone object. See `?Metric` for example usage.

Alternatively, if called with `y_true` and `y_pred` arguments, then the computed case-wise values for the mini-batch are returned directly.

**See Also**

Other metrics: `custom_metric()`, `metric_accuracy()`, `metric_auc()`, `metric_binary_accuracy()`, `metric_binary_crossentropy()`, `metric_categorical_accuracy()`, `metric_categorical_crossentropy()`, `metric_categorical_hinge()`, `metric_cosine_similarity()`, `metric_false_negatives()`, `metric_false_positives()`, `metric_kullback_leibler_divergence()`, `metric_logcosh_error()`, `metric_mean_absolute_error()`, `metric_mean_absolute_percentage_error()`, `metric_mean_iou()`, `metric_mean_relative_error()`, `metric_mean_squared_error()`, `metric_mean_squared_logarithmic_error()`, `metric_mean_tensor()`, `metric_mean_wrapper()`, `metric_mean()`, `metric_poisson()`, `metric_precision_at_recall()`, `metric_precision()`, `metric_recall_at_precision()`, `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()`

---

**Description**

Computes Kullback-Leibler divergence
Usage

```python
metric_kullback_leibler_divergence(
    y_true,
    y_pred,
    ...,  
    name = "kullback_leibler_divergence",
    dtype = NULL
)
```

Arguments

- `y_true`: Tensor of true targets.
- `y_pred`: Tensor of predicted targets.
- `...`: Passed on to the underlying metric. Used for forwards and backwards compatibility.
- `name`: (Optional) string name of the metric instance.
- `dtype`: (Optional) data type of the metric result.

Details

```latex
\text{metric} = \text{y_true} \times \log(\text{y_true} / \text{y_pred})
```

See: https://en.wikipedia.org/wiki/Kullback%E2%80%93Leibler_divergence

Value

If `y_true` and `y_pred` are missing, a (subclassed) Metric instance is returned. The Metric object can be passed directly to `compile(metrics = )` or used as a standalone object. See `?Metric` for example usage.

Alternatively, if called with `y_true` and `y_pred` arguments, then the computed case-wise values for the mini-batch are returned directly.

See Also

Other metrics: `custom_metric()`, `metric_accuracy()`, `metric_auc()`, `metric_binary_accuracy()`, `metric_binary_crossentropy()`, `metric_categorical_accuracy()`, `metric_categorical_crossentropy()`, `metric_categorical_hinge()`, `metric_cosine_similarity()`, `metric_false_negatives()`, `metric_false_positives()`, `metric_hinge()`, `metric_logcosh_error()`, `metric_mean_absolute_error()`, `metric_mean_absolute_percentage_error()`, `metric_mean_iou()`, `metric_mean_relative_error()`, `metric_mean_squared_error()`, `metric_mean_squared_logarithmic_error()`, `metric_mean_tensor()`, `metric_mean_wrapper()`, `metric_mean()`, `metric_poisson()`, `metric_precision_at_recall()`, `metric_precision()`, `metric_recall_at_precision()`, `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_logcosh_error**

*Computes the logarithm of the hyperbolic cosine of the prediction error*

**Description**

\[
\text{logcosh} = \log\left(\frac{\exp(x) + \exp(-x)}{2}\right), \text{where } x \text{ is the error } (y_{\text{pred}} - y_{\text{true}})
\]

**Usage**

\[
\text{metric_logcosh_error(..., name = "logcosh", dtype = NULL)}
\]

**Arguments**

- ...: Passed on to the underlying metric. Used for forwards and backwards compatibility.
- name: (Optional) string name of the metric instance.
- dtype: (Optional) data type of the metric result.

**Value**

A (subclassed) Metric instance that can be passed directly to `compile(metrics = )`, or used as a standalone object. See `?Metric` for example usage.

**See Also**

Other metrics: `custom_metric()`, `metric_accuracy()`, `metric_auc()`, `metric_binary_accuracy()`, `metric_binary_crossentropy()`, `metric_categorical_accuracy()`, `metric_categorical_crossentropy()`, `metric_categorical_hinge()`, `metric_cosine_similarity()`, `metric_false_negatives()`, `metric_false_positives()`, `metric_hinge()`, `metric_kullback_leibler_divergence()`, `metric_mean_absolute_error()`, `metric_mean_absolute_percentage_error()`, `metric_mean_iou()`, `metric_mean_relative_error()`, `metric_mean_squared_error()`, `metric_mean_squared_logarithmic_error()`, `metric_mean_tensor()`, `metric_mean_wrapper()`, `metric_mean()`, `metric_poisson()`, `metric_precision_at_recall()`, `metric_precision()`, `metric_recall_at_precision()`, `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_mean

Computes the (weighted) mean of the given values

Description
Computes the (weighted) mean of the given values

Usage
metric_mean(..., name = "mean", dtype = NULL)

Arguments
...
Passed on to the underlying metric. Used for forwards and backwards compatibility.
name (Optional) string name of the metric instance.
dtype (Optional) data type of the metric result.

Details
For example, if values is c(1,3,5,7) then the mean is 4. If the weights were specified as c(1,1,0,0) then the mean would be 2.
This metric creates two variables, total and count that are used to compute the average of values. This average is ultimately returned as mean which is 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.

Value
A (subclassed) Metric instance that can be passed directly to compile(metrics = ), or used as a standalone object. See ?Metric for example usage.

Note
Unlike most other metrics, this only takes a single tensor as input to update state.
Example usage with compile():
model$add_metric(metric_mean(name='mean_1')(outputs))
model %>% compile(optimizer='sgd', loss='mse')

Example standalone usage:
m <- metric_mean()
m$update_state(c(1, 3, 5, 7))
m$result()
metric_mean_absolute_error

Computes the mean absolute error between the labels and predictions

Description

Computes the mean absolute error between the labels and predictions

Usage

metric_mean_absolute_error(
    y_true,
    y_pred,
    ..., 
    name = "mean_absolute_error",
    dtype = NULL
)

Arguments

y_true
    Tensor of true targets.

y_pred
    Tensor of predicted targets.

... 
    Passed on to the underlying metric. Used for forwards and backwards compatibility.

name 
    (Optional) string name of the metric instance.

dtype 
    (Optional) data type of the metric result.
**Details**

\[
\text{loss} = \text{mean}(\text{abs}(y_{\text{true}} - y_{\text{pred}}), \text{axis}=-1)
\]

**Value**

If \(y_{\text{true}}\) and \(y_{\text{pred}}\) are missing, a (subclassed) Metric instance is returned. The Metric object can be passed directly to `compile(metrics = )` or used as a standalone object. See `?Metric` for example usage.

Alternatively, if called with \(y_{\text{true}}\) and \(y_{\text{pred}}\) arguments, then the computed case-wise values for the mini-batch are returned directly.

**See Also**

Other metrics: `custom_metric()`, `metric_accuracy()`, `metric_auc()`, `metric_binary_accuracy()`, `metric_binary_crossentropy()`, `metric_categorical_accuracy()`, `metric_categorical_crossentropy()`, `metric_categorical_hinge()`, `metric_categorical_loss()`, `metric_categorical_probability()`, `metric_categorical_precision()`, `metric_categorical_recall()`, `metric_categorical_roc_auc()`, `metric_categorical_roc_auc_multiclass()`, `metric_categorical_roc_auc_multilabel()`, `metric_categorical_sensitivity()`, `metric_categorical_specificity()`, `metric_categorical_f1_score()`, `metric_categorical_fbeta_score()`, `metric_categorical_jaccard_index()`, `metric_categorical_kappa()`, `metric_categorical_kappa_weighted()`, `metric_categorical_precision_at_recall()`, `metric_categorical_recall_at_precision()`, `metric_categorical_roc_auc()`, `metric_categorical_roc_auc_multiclass()`, `metric_categorical_roc_auc_multilabel()`, `metric_categorical_sensitivity()`, `metric_categorical_specificity()`, `metric_categorical_f1_score()`, `metric_categorical_fbeta_score()`, `metric_categorical_jaccard_index()`, `metric_categorical_kappa()`, `metric_categorical_kappa_weighted()`, `metric_categorical_precision_at_recall()`, `metric_categorical_recall_at_precision()`, `metric_categorical_roc_auc()`, `metric_categorical_roc_auc_multiclass()`, `metric_categorical_roc_auc_multilabel()`, `metric_categorical_sensitivity()`, `metric_categorical_specificity()`, `metric_categorical_f1_score()`, `metric_categorical_fbeta_score()`, `metric_categorical_jaccard_index()`, `metric_categorical_kappa()`, `metric_categorical_kappa_weighted()`, `metric_categorical_precision_at_recall()`, `metric_categorical_recall_at_precision()`, `metric_categorical_roc_auc()`, `metric_categorical_roc_auc_multiclass()`, `metric_categorical_roc_auc_multilabel()`, `metric_categorical_sensitivity()`, `metric_categorical_specificity()`, `metric_categorical_f1_score()`, `metric_categorical_fbeta_score()`, `metric_categorical_jaccard_index()`, `metric_categorical_kappa()`, `metric_categorical_kappa_weighted()`, `metric_categorical_precision_at_recall()`, `metric_categorical_recall_at_precision()`, `metric_categorical_roc_auc()`, `metric_categorical_roc_auc_multiclass()`, `metric_categorical_roc_auc_multilabel()`, `metric_categorical_sensitivity()`, `metric_categorical_specificity()`, `metric_categorical_f1_score()`, `metric_categorical_fbeta_score()`, `metric_categorical_jaccard_index()`, `metric_categorical_kappa()`, `metric_categorical_kappa_weighted()`, `metric_categorical_precision_at_recall()`, `metric_categorical_recall_at_precision()`, `metric_categorical_roc_auc()`, `metric_categorical_roc_auc_multiclass()`, `metric_categorical_roc_auc_multilabel()`

---

**Description**

Computes the mean absolute percentage error between \(y_{\text{true}}\) and \(y_{\text{pred}}\)

**Usage**

```r
metric_mean_absolute_percentage_error(
  y_true,
  y_pred,
  ..., 
  name = "mean_absolute_percentage_error",
  dtype = NULL
)
```
metric_mean_iou

Computes the mean Intersection-Over-Union metric

Description

Computes the mean Intersection-Over-Union metric

Usage

metric_mean_iou(..., num_classes, name = NULL, dtype = NULL)
Arguments

... Passed on to the underlying metric. Used for forwards and backwards compatibility.

num_classes The possible number of labels the prediction task can have. This value must be provided, since a confusion matrix of \( \text{dim} \left( \text{num_classes}, \text{num_classes} \right) \) will be allocated.

name (Optional) string name of the metric instance.

dtype (Optional) data type of the metric result.

Details

Mean Intersection-Over-Union is a common evaluation metric for semantic image segmentation, which first computes the IOU for each semantic class and then computes the average over classes. IOU is defined as follows:

\[
\text{IOU} = \frac{\text{true_positive}}{\text{true_positive} + \text{false_positive} + \text{false_negative}}
\]

The predictions are accumulated in a confusion matrix, weighted by \text{sample_weight} and the metric is then calculated from it.

If \text{sample_weight} is NULL, weights default to 1. Use \text{sample_weight} of 0 to mask values.

Value

A (subclassed) Metric instance that can be passed directly to \text{compile}(), or used as a standalone object. See \?Metric for example usage.

See Also

Other metrics: \text{custom_metric()}, \text{metric_accuracy()}, \text{metric_auc()}, \text{metric_binary_accuracy()}, \text{metric_binary_crossentropy()}, \text{metric_categorical_accuracy()}, \text{metric_categorical_crossentropy()}, \text{metric_categorical_hinge()}, \text{metric_cosine_similarity()}, \text{metric_false_negatives()}, \text{metric_false_positives()}, \text{metric_hinge()}, \text{metric_kullback_leibler_divergence()}, \text{metric_logcosh_error()}, \text{metric_mean_absolute_error()}, \text{metric_mean_absolute_percentage_error()}, \text{metric_mean_relative_error()}, \text{metric_mean_squared_error()}, \text{metric_mean_squared_logarithmic_error()}, \text{metric_mean_tensor()}, \text{metric_mean_wrapper()}, \text{metric_mean()}, \text{metric_poisson()}, \text{metric_precision_at_recall()}, \text{metric_precision()}, \text{metric_recall_at_precision()}, \text{metric_recall()}, \text{metric_root_mean_squared_error()}, \text{metric_sensitivity_at_specificity()}, \text{metric_sparse_categorical_accuracy()}, \text{metric_sparse_categorical_crossentropy()}, \text{metric_sparse_top_k_categorical_accuracy()}, \text{metric_specificity_at_sensitivity()}, \text{metric_squared_hinge()}, \text{metric_sum()}, \text{metric_top_k_categorical_accuracy()}, \text{metric_true_negatives()}, \text{metric_true_positives()}
**Description**

Computes the mean relative error by normalizing with the given values

**Usage**

```r
metric_mean_relative_error(..., normalizer, name = NULL, dtype = NULL)
```

**Arguments**

- `...` Passed on to the underlying metric. Used for forwards and backwards compatibility.
- `normalizer` The normalizer values with same shape as predictions.
- `name` (Optional) string name of the metric instance.
- `dtype` (Optional) data type of the metric result.

**Details**

This metric creates two local variables, `total` and `count` that are used to compute the mean relative error. This is weighted by `sample_weight`, and it is ultimately returned as `mean_relative_error`: 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.

```r
metric = mean(|y_pred - y_true| / normalizer)
```

For example:

```r
m = metric_mean_relative_error(normalizer=c(1, 3, 2, 3))
m$update_state(c(1, 3, 2, 3), c(2, 4, 6, 8))
# result = mean(c(1, 1, 4, 5) / c(1, 3, 2, 3)) = mean(c(1/3, 2, 5/3))
# = 5/4 = 1.25
m$result()
```

**Value**

A (subclassed) Metric instance that can be passed directly to `compile(metrics = )`, or used as a standalone object. See `Metric` for example usage.
metric_mean_squared_error

Computes the mean squared error between labels and predictions

Description
Computes the mean squared error between labels and predictions.

Usage
```r
metric_mean_squared_error(
  y_true,
  y_pred,
  ...,
  name = "mean_absolute_percentage_error",
  dtype = NULL
)
```

Arguments
- `y_true`: Tensor of true targets.
- `y_pred`: Tensor of predicted targets.
- `...`: Passed on to the underlying metric. Used for forwards and backwards compatibility.
- `name`: (Optional) string name of the metric instance.
- `dtype`: (Optional) data type of the metric result.

Details
After computing the squared distance between the inputs, the mean value over the last dimension is returned.

```
loss = mean(square(y_true - y_pred), axis=-1)
```
**Value**

If `y_true` and `y_pred` are missing, a (subclassed) Metric instance is returned. The Metric object can be passed directly to `compile(metrics =)` or used as a standalone object. See ?Metric for example usage.

Alternatively, if called with `y_true` and `y_pred` arguments, then the computed case-wise values for the mini-batch are returned directly.

**See Also**

Other metrics: `custom_metric()`, `metric_accuracy()`, `metric_auc()`, `metric_binary_accuracy()`, `metric_binary_crossentropy()`, `metric_categorical_accuracy()`, `metric_categorical_crossentropy()`, `metric_categorical_hinge()`, `metric_categorical_hinge()`, `metric_cosine_similarity()`, `metric_false_negatives()`, `metric_false_positives()`, `metric_hinge()`, `metric_kullback_leibler_divergence()`, `metric_logcosh_error()`, `metric_mean_absolute_error()`, `metric_mean_absolute_percentage_error()`, `metric_mean_iou()`, `metric_mean_relative_error()`, `metric_mean_squared_logarithmic_error()`, `metric_mean_tensor()`, `metric_mean_wrapper()`, `metric_mean()`, `metric_poisson()`, `metric_precision_at_recall()`, `metric_precision()`, `metric_recall_at_precision()`, `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()`

---

**Description**

Computes the mean squared logarithmic error

**Usage**

```r
metric_mean_squared_logarithmic_error(
  y_true, 
  y_pred, 
  ..., 
  name = "mean_squared_logarithmic_error", 
  dtype = NULL
)
```

**Arguments**

- `y_true`  
  Tensor of true targets.
- `y_pred`  
  Tensor of predicted targets.
- `...`  
  Passed on to the underlying metric. Used for forwards and backwards compatibility.
- `name`  
  (Optional) string name of the metric instance.
- `dtype`  
  (Optional) data type of the metric result.
**metric_mean_tensor**

**Details**

\[
\text{loss} = \text{mean}(\text{square}(\log(y_{\text{true}} + 1) - \log(y_{\text{pred}} + 1)), \text{axis}=-1)
\]

**Value**

If \(y_{\text{true}}\) and \(y_{\text{pred}}\) are missing, a (subclassed) Metric instance is returned. The Metric object can be passed directly to \text{compile(metrics = )} or used as a standalone object. See \text{?Metric} for example usage.

Alternatively, if called with \(y_{\text{true}}\) and \(y_{\text{pred}}\) arguments, then the computed case-wise values for the mini-batch are returned directly.

**See Also**

Other metrics: \text{custom_metric()}, \text{metric_accuracy()}, \text{metric_auc()}, \text{metric_binary_accuracy()}, \text{metric_binary_crossentropy()}, \text{metric_categorical_accuracy()}, \text{metric_categorical_crossentropy()}, \text{metric_categorical_hinge()}, \text{metric_cosine_similarity()}, \text{metric_false_negatives()}, \text{metric_false_positives()}, \text{metric_hinge()}, \text{metric_kullback_leibler_divergence()}, \text{metric_logcosh_error()}, \text{metric_mean_absolute_error()}, \text{metric_mean_absolute_percentage_error()}, \text{metric_mean_iou()}, \text{metric_mean_relative_error()}, \text{metric_mean_squared_error()}, \text{metric_mean_tensor()}, \text{metric_mean_wrapper()}, \text{metric_mean()}, \text{metric_poisson()}, \text{metric_precision_at_recall()}, \text{metric_precision()}, \text{metric_recall_at_precision()}, \text{metric_recall()}, \text{metric_root_mean_squared_error()}, \text{metric_sensitivity_at_specificity()}, \text{metric_sparse_categorical_accuracy()}, \text{metric_sparse_categorical_crossentropy()}, \text{metric_sparse_top_k_categorical_accuracy()}, \text{metric_sparse_top_k_categorical_crossentropy()}, \text{metric_squared_hinge()}, \text{metric_sum()}, \text{metric_top_k_categorical_accuracy()}, \text{metric_true_negatives()}, \text{metric_true_positives()}

---

**metric_mean_tensor**  Computes the element-wise (weighted) mean of the given tensors

---

**Description**

Computes the element-wise (weighted) mean of the given tensors

**Usage**

\[
\text{metric\_mean\_tensor}(\ldots, \text{shape} = \text{NULL}, \text{name} = \text{NULL}, \text{dtype} = \text{NULL})
\]

**Arguments**

- \(\ldots\): Passed on to the underlying metric. Used for forwards and backwards compatibility.
- \text{shape}: (Optional) A list of integers, a list of integers, or a 1-D Tensor of type int32. If not specified, the shape is inferred from the values at the first call of \text{update\_state}.
- \text{name}: (Optional) string name of the metric instance.
- \text{dtype}: (Optional) data type of the metric result.
**metric_mean_wrapper**

Wraps a stateless metric function with the Mean metric

---

**Description**

Wraps a stateless metric function with the Mean metric

**Usage**

```
metric_mean_wrapper(..., fn, name = NULL, dtype = NULL)
```

**Arguments**

- `...` named 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.
metric_poisson

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. `MeanMetricWrapper$result()` will return the average metric value across all samples seen so far.

For example:

```r
accuracy <- function(y_true, y_pred)
  k_cast(y_true == y_pred, 'float32')

accuracy_metric <- metric_mean_wrapper(fn = accuracy)
model %>% compile(..., metrics=accuracy_metric)
```

**Value**

A (subclassed) `Metric` instance that can be passed directly to `compile(metrics = )`, or used as a standalone object. See `?Metric` for example usage.

**See Also**

Other metrics: `custom_metric()`, `metric_accuracy()`, `metric_auc()`, `metric_binary_accuracy()`, `metric_binary_crossentropy()`, `metric_categorical_accuracy()`, `metric_categorical_crossentropy()`, `metric_categorical_hinge()`, `metric_cosine_similarity()`, `metric_false_negatives()`, `metric_false_positives()`, `metric_hinge()`, `metric_kullback_leibler_divergence()`, `metric_logcosh_error()`, `metric_mean_absolute_error()`, `metric_mean_absolute_percentage_error()`, `metric_mean_iou()`, `metric_mean_relative_error()`, `metric_mean_squared_error()`, `metric_mean_squared_logarithmic_error()`, `metric_mean_tensor()`, `metric_mean()`, `metric_poisson()`, `metric_precision_at_recall()`, `metric_precision()`, `metric_recall_at_precision()`, `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()`

---

**Description**

\[ \text{metric} = y_{\text{pred}} - y_{\text{true}} \times \log(y_{\text{pred}}) \]

**Usage**

`metric_poisson(y_{\text{true}}, y_{\text{pred}}, ..., \text{name} = "poisson", \text{dtype} = \text{NULL})`
**metric_precision**

**Arguments**

- **y_true**: Tensor of true targets.
- **y_pred**: Tensor of predicted targets.
- **...**: Passed on to the underlying metric. Used for forwards and backwards compatibility.
- **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 (subclassed) Metric instance is returned. The Metric object can be passed directly to `compile(metrics = )` or used as a standalone object. See `?Metric` for example usage.

Alternatively, if called with `y_true` and `y_pred` arguments, then the computed case-wise values for the mini-batch are returned directly.

**See Also**

Other metrics: `custom_metric()`, `metric_accuracy()`, `metric_auc()`, `metric_binary_accuracy()`, `metric_binary_crossentropy()`, `metric_categorical_accuracy()`, `metric_categorical_crossentropy()`, `metric_categorical_hinge()`, `metric_categorical_hinge()`, `metric_cosine_similarity()`, `metric_false_negatives()`, `metric_false_positives()`, `metric_hinge()`, `metric_kullback_leibler_divergence()`, `metric_logcosh_error()`, `metric_mean_absolute_error()`, `metric_mean_absolute_percentage_error()`, `metric_mean_iou()`, `metric_mean_relative_error()`, `metric_mean_squared_error()`, `metric_mean_squared_logarithmic_error()`, `metric_mean_tensor()`, `metric_mean_wrapper()`, `metric_mean()`, `metric_precision_at_recall()`, `metric_precision()`, `metric_recall_at_precision()`, `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_precision | Computes the precision of the predictions with respect to the labels |

**Description**

Computes the precision of the predictions with respect to the labels

**Usage**

```r
metric_precision(
    ...,
    thresholds = NULL,
    top_k = NULL,
    class_id = NULL,
    name = NULL,
    dtype = NULL
)
```
Arguments

Passed on to the underlying metric. Used for forwards and backwards compatibility.

thresholds (Optional) A float value or a 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). 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) string name of the metric instance.

Details

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.

Value

A (subclassed) Metric instance that can be passed directly to compile(metrics = ), or used as a standalone object. See ?Metric for example usage.

See Also

Other metrics: custom_metric(), metric_accuracy(), metric_auc(), metric_binary_accuracy(), metric_binary_crossentropy(), metric_categorical_accuracy(), metric_categorical_crossentropy(), metric_categorical_hinge(), metric_cosine_similarity(), metric_false_negatives(), metric_false_positives(), metric_hinge(), metric_kullback_leibler_divergence(), metric_logcosh_error(), metric_mean_absolute_error(), metric_mean_absolute_percentage_error(), metric_mean_iou(), metric_mean_relative_error(), metric_mean_squared_error(), metric_mean_squared_logarithmic_error(), metric_mean_tensor(), metric_mean_wrapper(), metric_mean(), metric_poisson(), metric_precision_at_recall(), metric_recall_at_precision(), 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_precision_at_recall

Computes best precision where recall is >= specified value

Description
Computes best precision where recall is >= specified value

Usage

```r
metric_precision_at_recall(
    ..., recall, num_thresholds = 200L, class_id = NULL, name = NULL, dtype = NULL
)
```

Arguments

- `...`: Passed on to the underlying metric. Used for forwards and backwards compatibility.
- `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.

Details
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.

Value

A (subclassed) Metric instance that can be passed directly to `compile(metrics = )`, or used as a standalone object. See `Metric` for example usage.
See Also

Other metrics: `custom_metric()`, `metric_accuracy()`, `metric_auc()`, `metric_binary_accuracy()`, `metric_binary_crossentropy()`, `metric_categorical_accuracy()`, `metric_categorical_crossentropy()`, `metric_categorical_hinge()`, `metric_cosine_similarity()`, `metric_false_negatives()`, `metric_false_positives()`, `metric_hinge()`, `metric_kullback_leibler_divergence()`, `metric_logcosh_error()`, `metric_mean_absolute_error()`, `metric_mean_absolute_percentage_error()`, `metric_mean_iou()`, `metric_mean_relative_error()`, `metric_mean_squared_error()`, `metric_mean_squared_logarithmic_error()`, `metric_mean_tensor()`, `metric_mean_wrapper()`, `metric_mean()`, `metric_poisson()`, `metric_precision()`, `metric_recall_at_precision()`, `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_recall**

Computes the recall of the predictions with respect to the labels

**Description**

Computes the recall of the predictions with respect to the labels

**Usage**

```r
metric_recall(
  ...,
  thresholds = NULL,
  top_k = NULL,
  class_id = NULL,
  name = NULL,
  dtype = NULL
)
```

**Arguments**

- `...` (Optional) Passed on to the underlying metric. Used for forwards and backwards compatibility.
- `thresholds` (Optional) A float value or a 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`). 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.
**Details**

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.

**Value**

A (subclassed) `Metric` instance that can be passed directly to `compile(metrics = )`, or used as a standalone object. See `?Metric` for example usage.

**See Also**

Other metrics: `custom_metric()`, `metric_accuracy()`, `metric_auc()`, `metric_binary_accuracy()`, `metric_binary_crossentropy()`, `metric_categorical_accuracy()`, `metric_categorical_crossentropy()`, `metric_categorical_hinge()`, `metric_categorical_crossentropy()`, `metric_categorical_hinge()`, `metric_cosine_similarity()`, `metric_false_negatives()`, `metric_false_positives()`, `metric_hinge()`, `metric_kullback_leibler_divergence()`, `metric_logcosh_error()`, `metric_mean_absolute_error()`, `metric_mean_absolute_percentage_error()`, `metric_mean_iou()`, `metric_mean_relative_error()`, `metric_mean_squared_error()`, `metric_mean_squared_logarithmic_error()`, `metric_mean_tensor()`, `metric_mean_wrapper()`, `metric_mean()`, `metric_poisson()`, `metric_precision_at_recall()`, `metric_precision()`, `metric_recall_at_precision()`, `metric_root_mean_squared_error()`, `metric_sensitivity_at_specificity()`, `metric_sparse_categorical_accuracy()`, `metric_sparse_categorical_crossentropy()`, `metric_sensitivity_at_specificity()`, `metric_squared_hinge()`, `metric_sum()`, `metric_top_k_categorical_accuracy()`, `metric_true_negatives()`, `metric_true_positives()`
Arguments

Passed on to the underlying metric. Used for forwards and backwards compatibility.

- **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 $[0, \text{num_classes})$, where $\text{num_classes}$ is the last dimension of predictions.
- **name**: (Optional) string name of the metric instance.
- **dtype**: (Optional) data type of the metric result.

Details

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.

Value

A (subclassed) `Metric` instance that can be passed directly to `compile(metrics = )`, or used as a standalone object. See ?`Metric` for example usage.

See Also

Other metrics: `custom_metric()`, `metric_accuracy()`, `metric_auc()`, `metric_binary_accuracy()`, `metric_binary_crossentropy()`, `metric_categorical_accuracy()`, `metric_categorical_crossentropy()`, `metric_categorical_hinge()`, `metric_categorical_dice()`, `metric_categorical_hinge()`, `metric_categorical_sigmoid()`, `metric_categorical_sigmoid()`, `metric_categorical_dice()`, `metric_categorical_hinge()`, `metric_categorical_sigmoid()`, `metric_categorical_dice()`, `metric_categorical_hinge()`, `metric_categorical_sigmoid()`, `metric_categorical_dice()`, `metric_categorical_hinge()`, `metric_categorical_sigmoid()`, `metric_categorical_dice()`, `metric_categorical_hinge()`, `metric_categorical_sigmoid()`, `metric_categorical_dice()`, `metric_categorical_hinge()`, `metric_categorical_sigmoid()`, `metric_categorical_dice()`, `metric_categorical_hinge()`, `metric_categorical_sigmoid()`, `metric_categorical_dice()`, `metric_categorical_hinge()`, `metric_categorical_sigmoid()`, `metric_categorical_dice()`, `metric_categorical_hinge()`, `metric_categorical_sigmoid()`, `metric_categorical_dice()`, `metric_categorical_hinge()`, `metric_categorical_sigmoid()`, `metric_categorical_dice()`, `metric_categorical_hinge()`, `metric_categorical_sigmoid()`, `metric_categorical_dice()`, `metric_categorical_hinge()`, `metric_categorical_sigmoid()`, `metric_categorical_dice()`, `metric_categorical_hinge()`, `metric_categorical_sigmoid()`, `metric_categorical_dice()`, `metric_categorical_hinge()`, `metric_categorical_sigmoid()`, `metric_categorical_dice()`, `metric_categorical_hinge()`, `metric_categorical_sigmoid()`, `metric_categorical_dice()`, `metric_categorical_hinge()`, `metric_categorical_sigmoid()`, `metric_categorical_dice()`, `metric_categorical_hinge()`, `metric_categorical_sigmoid()`, `metric_categorical_dice()`, `metric_categorical_hinge()`, `metric_categorical_sigmoid()`, `metric_categorical_dice()`, `metric_categorical_hinge()`, `metric_categorical_sigmoid()`, `metric_categorical_dice()`, `metric_categorical_hinge()`, `metric_categorical_sigmoid()`, `metric_categorical_dice()`, `metric_categorical_hinge()`, `metric_categorical_sigmoid()`, `metric_categorical_dice()`, `metric_categorical_hinge()`, `metric_categorical_sigmoid()`, `metric_categorical_dice()`, `metric_categorical_hinge()`, `metric_categorical_sigmoid()`, `metric_categorical_dice()`, `metric_categorical_hinge()`, `metric_categorical_sigmoid()`, `metric_categorical_dice()`, `metric_categorical_hinge()`, `metric_categorical_sigmoid()`, `metric_categorical_dice()`, `metric_categorical_hinge()`, `metric_categorical_sigmoid()`, `metric_categorical_dice()`, `metric_categorical_hinge()`, `metric_categorical_sigmoid()`, `metric_categorical_dice()`, `metric_categorical_hinge()`, `metric_categorical_sigmoid()`, `metric_categorical_dice()`, `metric_categorical_hinge()`, `metric_categorical_sigmoid()`, `metric_categorical_dice()`, `metric_categorical_hinge()`, `metric_categorical_sigmoid()`, `metric_categorical_dice()`, `metric_categorical_hinge()`, `metric_categorical_sigmoid()`, `metric_categorical_dice()`, `metric_categorical_hinge()`, `metric_categorical_sigmoid()`, `metric_categorical_dice()`, `metric_categorical_hinge()`, `metric_categorical_sigmoid()`, `metric_categorical_dice()`, `metric_categorical_hinge()`, `metric_categorical_sigmoid()`, `metric_categorical_dice()`, `metric_categorical_hinge()`, `metric_categorical_sigmoid()`, `metric_categorical_dice()`, `metric_categorical_hinge()`, `metric_categorical_sigmoid()`, `metric_categorical_dice()`, `metric_categorical_hinge()`, `metric_categorical_sigmoid()`, `metric_categorical_dice()`, `metric_categorical_hinge()`, `metric_categorical_sigmoid()`, `metric_categorical_dice()`, `metric_categorical_hinge()`, `metric_categorical_sigmoid()`, `metric_categorical_dice()`, `metric_categorical_hinge()`, `metric_categorical_sigmoid()`, `metric_categorical_dice()`, `metric_categorical_hinge()`, `metric_categorical_sigmoid()`, `metric_categorical_dice()`, `metric_categorical_hinge()`, `metric_categorical_sigmoid()`, `metric_categorical_dice()`, `metric_categorical_hinge()`, `metric_categorical_sigmoid()`, `metric_categorical_dice()`, `metric_categorical_hinge()`, `metric_categorical_sigmoid()`, `metric_categorical_dice()`, `metric_categorical_hinge()`, `metric_categorical_sigmoid()`, `metric_categorical_dice()`, `metric_categorical_hinge()`, `metric_categorical_sigmoid()`, `metric_categorical_dice()`, `metric_categorical_hinge()`, `metric_categorical_sigmoid()`, `metric_categorical_dice()`, `metric_categorical_hinge()`, `metric_categorical_sigmoid()`, `metric_categorical_dice()`, `metric_categorical_hinge()`, `metric_categorical_sigmoid()`, `metric_categorical_dice()`, `metric_categorical_hinge()`, `metric_categorical_sigmoid()`, `metric_categorical_dice()`, `metric_categorical_hinge()`, `metric_categorical_sigmoid()`, `metric_categorical_dice()`, `metric_categorical_hinge()`, `metric_categorical_sigmoid()`, `metric_categorical_dice()`, `metric_categorical_hinge()`, `metric_categorical_sigmoid()`, `metric_categorical_dice()`, `metric_categorical_hinge()`, `metric_categorical_sigmoid()`, `metric_categorical_dice()`, `metric_categorical_hinge()`, `metric_categorical_sigmoid()`, `metric_categorical_dice()`, `metric_categorical_hinge()`, `metric_categorical_sigmoid()`, `metric_categorical_dice()`, `metric_categorical_hinge()`, `metric_categorical_sigmoid()`, `metric_categorical_dice()`, `metric_categorical_hinge()`, `metric_categorical_sigmoid()`, `metric_categorical_dice()`, `metric_categorical_hinge()`, `metric_categorical_sigmoid()`, `metric_categorical_dice()`, `metric_categorical_hinge()`, `metric_categorical_sigmoid()`, `metric_categorical_dice()`, `metric_categorical_hinge()`, `metric_categorical_sigmoid()`, `metric_categorical_dice()`, `metric_categorical_hinge()`, `metric_categorical_sigmoid()`, `metric_categorical_dice()`, `metric_categorical_hinge()`, `metric_categorical_sigmoid()`, `metric_categorical_dice()`, `metric_categorical_hinge()`, `metric_categorical_sigmoid()`, `metric_categorical_dice()`, `metric_categorical_hinge()`, `metric_categorical_sigmoid()`, `metric_categorical_dice()`, `metric_categorical_hinge()`, `metric_categorical_sigmoid()`, `metric_categorical_dice()`, `metric_categorical_hinge()`, `metric_categorical_sigmoid()`, `metric_categorical_dice()`, `metric_categorical_hinge()`, `metric_categorical_sigmoid()`, `metric_categorical_dice()`, `metric_categorical_hinge()`, `metric_categorical_sigmoid()`, `metric_categorical_dice()`, `metric_categorical_hinge()`, `metric_categorical_sigmoid()`, `metric_categorical_dice()`, `metric_categorical_hinge()`, `metric_categorical_sigmoid()`, `metric_categorical_dice()`, `metric_categorical_hinge()`, `metric_categorical_sigmoid()`, `metric_categorical_dice()`, `metric_categorical_hinge()`, `metric_categorical_sigmoid()`, `metric_categorical_dice()`, `metric_categorical_hinge()`, `metric_categorical_sigmoid()`, `metric_categorical_dice()`, `metric_categorical_hinge()`, `metric_categorical_sigmoid()`, `metric_categorical_dice()`, `metric_categorical_hinge()`, `metric_categorical_sigmoid()`, `metric_categorical_dice()`, `metric_categorical_hinge()`, `metric_categorical_sigmoid()`, `metric_categorical_dice()`, `metric_categorical_hinge()`, `metric_categorical_sigmoid()`, `metric_categorical_dice()`, `metric_categorical_hinge()`, `metric_categorical_sigmoid()`, `metric_categorical_dice()`, `metric_categorical_hinge()`, `metric_categorical_sigmoid()`, `metric_categorical_dice()`, `metric_categorical_hinge()`, `metric_categorical_sigmoid()`, `metric_categorical_dice()`, `metric_categorical_hinge()`, `metric_categorical_sigmoid()`, `metric_categorical_dice()`, `metric_categorical_hinge()`, `metric_categorical_sigmoid()`, `metric_categorical_dice()`, `metric_categorical_hinge()`, `metric_categorical_sigmoid()`, `metric_categorical_dice()`, `metric_categorical_hinge()`, `metric_categorical_sigmoid()`, `metric_categorical_dice()`, `metric_categorical_hinge()`, `metric_categorical_sigmoid()`, `metric_categorical_dice()`, `metric_categorical_hinge()`, `metric_categorical_sigmoid()`, `metric_categorical_dice()`, `metric_categorical_hinge()`, `metric_categorical_sigmoid()`, `metric_categorical_dice()`, `metric_categorical_hinge()`, `metric_categorical_sigmoid()`, `metric_categorical_dice()`, `metric_categorical_hinge()`, `metric_categorical_sigmoid()`, `metric_categorical_dice()`, `metric_categorical_hinge()`, `metric_categorical_sigmoid()`, `metric_categorical_dice()`, `metric_categorical_hinge()`, `metric_categorical_sigmoid()`, `metric_categorical_dice()`, `metric_categorical_hinge()`, `metric_categorical_sigmoid()`, `metric_categorical_dice()`, `metric_categorical_hinge()`, `metric_categorical_sigmoid()`, `metric_categorical_dice()`, `metric_categorical_hinge()`, `metric_categorical_sigmoid()`, `metric_categorical_dice()`, `metric_categorical_hinge()`, `metric_categorical_sigmoid()`, `metric_categorical_dice()`, `metric_categorical_hinge()`, `metric_categorical_sigmoid()`, `metric_categorical_dice()`, `metric_categorical_hinge()`, `metric_categorical_sigmoid()`, `metric_categorical_dice()`, `metric_categorical_hinge()`, `metric_categorical_sigmoid()`, `metric_categorical_dice()`, `metric_categorical_hinge()`, `metric_categorical_sigmoid()`, `metric_categorical_dice()`, `metric_categorical_hinge()`, `metric_categorical_sigmoid()`, `metric_categorical_dice()`, `metric_categorical_hinge()`, `metric_categorical_sigmoid()`, `metric_categorical_dice()`, `metric_categorical_hinge()`, `metric_categorical_sigmoid()`, `metric_categorical_dice()`, `metrics_float`
metric_root_mean_squared_error

Description

Computes root mean squared error metric between \( y_{true} \) and \( y_{pred} \)

Usage

```r
metric_root_mean_squared_error(..., name = NULL, dtype = NULL)
```

Arguments

- `...` Passed on to the underlying metric. Used for forwards and backwards compatibility.
- `name` (Optional) string name of the metric instance.
- `dtype` (Optional) data type of the metric result.

Value

A (subclassed) Metric instance that can be passed directly to `compile(metrics = )`, or used as a standalone object. See `?Metric` for example usage.

See Also

Other metrics: `custom_metric()`, `metric_accuracy()`, `metric_auc()`, `metric_binary_accuracy()`, `metric_binary_crossentropy()`, `metric_categorical_accuracy()`, `metric_categorical_crossentropy()`, `metric_categorical_hinge()`, `metric_categorical_crossentropy()`, `metric_categorical_hinge()`, `metric_cosine_similarity()`, `metric_false_negatives()`, `metric_false_positives()`, `metric_hinge()`, `metric_kullback_leibler_divergence()`, `metric_logcosh_error()`, `metric_mean_absolute_error()`, `metric_mean_absolute_percentage_error()`, `metric_mean_iou()`, `metric_mean_relative_error()`, `metric_mean_squared_error()`, `metric_mean_squared_logarithmic_error()`, `metric_mean_tensor()`, `metric_mean_wrapper()`, `metric_mean()`, `metric_poisson()`, `metric_precision_at_recall()`, `metric_precision()`, `metric_recall_at_precision()`, `metric_recall()`, `metric_sensitivity_at_specificity()`, `metric_sparse_categorical_accuracy()`, `metric_sparse_categorical_crossentropy()`, `metric_sparse_top_k_categorical_accuracy()`, `metric_sparse_top_k_categorical_crossentropy()`, `metric_squared_hinge()`, `metric_sum()`, `metric_top_k_categorical_accuracy()`, `metric_true_negatives()`, `metric_true_positives()`
Description

The sensitivity at a given specificity.

Usage

```r
metric_sensitivity_at_specificity(
  ..., 
  specificity, 
  num_thresholds = 200L, 
  class_id = NULL, 
  name = NULL, 
  dtype = NULL
)
```

Arguments

- `...`: Passed on to the underlying metric. Used for forwards and backwards compatibility.
- `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.

Details

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.
**metric_sparse_categorical_accuracy**

Calculates how often predictions match integer labels

**Description**

Calculates how often predictions match integer labels

**Usage**

```r
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.
- `...`: Passed on to the underlying metric. Used for forwards and backwards compatibility.
- `name`: (Optional) string name of the metric instance.
- `dtype`: (Optional) data type of the metric result.
metric_sparse_categorical_crossentropy

Computes the crossentropy metric between the labels and predictions

Description
Computes the crossentropy metric between the labels and predictions

Usage

```r
metric_sparse_categorical_crossentropy(
  y_true,
  y_pred,
  from_logits = FALSE,
  axis = -1L,
)```

Details

```r
acc = k_dot(sample_weight, y_true == k_argmax(y_pred, axis=2))
```

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.

Value

If `y_true` and `y_pred` are missing, a (subclassed) `Metric` instance is returned. The `Metric` object can be passed directly to `compile(metrics = )` or used as a standalone object. See `?Metric` for example usage. Alternatively, if called with `y_true` and `y_pred` arguments, then the computed case-wise values for the mini-batch are returned directly.

See Also

Other metrics: `custom_metric()`, `metric_accuracy()`, `metric_auc()`, `metric_binary_accuracy()`, `metric_binary_crossentropy()`, `metric_categorical_crossentropy()`, `metric_categorical_hinge()`, `metric_categorical_accuracy()`, `metric_categorical_hinge()`, `metric_cosine_similarity()`, `metric_false_negatives()`, `metric_false_positives()`, `metric_hinge()`, `metric_kullback_leibler_divergence()`, `metric_logcosh_error()`, `metric_mean_absolute_error()`, `metric_mean_absolute_percentage_error()`, `metric_mean_iou()`, `metric_mean_relative_error()`, `metric_mean_squared_error()`, `metric_mean_squared_logarithmic_error()`, `metric_mean_tensor()`, `metric_mean_wrapper()`, `metric_mean()`, `metric_poisson()`, `metric_precision_at_recall()`, `metric_precision()`, `metric_recall_at_precision()`, `metric_recall()`, `metric_root_mean_squared_error()`, `metric_sensitivity_at_specificity()`, `metric_sparse_categorical_crossentropy()`, `metric_sparse_top_k_categorical_crossentropy()`, `metric_specificity_at_sensitivity()`, `metric_squared_hinge()`, `metric_sum()`, `metric_top_k_categorical_accuracy()`, `metric_true_negatives()`, `metric_true_positives()`
...,
name = "sparse_categorical_crossentropy",
dtype = NULL
)

Arguments

y_true Tensor of 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.
axis (Optional) (1-based) Defaults to -1. The dimension along which the metric is computed.
... Passed on to the underlying metric. Used for forwards and backwards compatibility.
name (Optional) string name of the metric instance.
dtype (Optional) data type of the metric result.

Details

Use this crossentropy metric 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 metric. 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 # 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].

Value

If y_true and y_pred are missing, a (subclassed) Metric instance is returned. The Metric object can be passed directly to compile(metrics = ) or used as a standalone object. See ?Metric for example usage.

Alternatively, if called with y_true and y_pred arguments, then the computed case-wise values for the mini-batch are returned directly.

See Also

Other metrics: custom_metric(), metric_accuracy(), metric_auc(), metric_binary_accuracy(), metric_binary_crossentropy(), metric_categorical_accuracy(), metric_categorical_crossentropy(), metric_categorical_hinge(), metric_cosine_similarity(), metric_false_negatives(), metric_false_positives(), metric_hinge(), metric_kullback_leibler_divergence(), metric_logcosh_error(), metric_mean_absolute_error(), metric_mean_absolute_percentage_error(), metric_mean_iou(), metric_mean_relative_error(), metric_mean_squared_error(), metric_mean_squared_logarithmic_error(), metric_mean_tensor(), metric_mean_wrapper(), metric_mean(), metric_poisson(), metric_precision_at_recall(), metric_precision(), metric_recall_at_precision(), metric_recall(), metric_root_mean_squared_error(), metric_sensitivity_at_specificity(), metric_sparse_categorical_accuracy(), metric_sparse_top_k_categorical_accuracy()
**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**

```r
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.
- `...` Passed on to the underlying metric. Used for forwards and backwards compatibility.
- `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 (subclassed) Metric instance is returned. The Metric object can be passed directly to `compile(metrics = )` or used as a standalone object. See `?Metric` for example usage.

Alternatively, if called with `y_true` and `y_pred` arguments, then the computed case-wise values for the mini-batch are returned directly.
metric_specificity_at_sensitivity

Computes best specificity where sensitivity is >= specified value

Description
Computes best specificity where sensitivity is >= specified value

Usage

```
metric_specificity_at_sensitivity(
    ..., 
    sensitivity, 
    num_thresholds = 200L, 
    class_id = NULL, 
    name = NULL, 
    dtype = NULL 
)
```

Arguments

```
... Passed on to the underlying metric. Used for forwards and backwards compatibility.

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 dimen-
```
```
Details

Sensitivity measures the proportion of actual positives that are correctly identified as such \((\text{tp} / (\text{tp} + \text{fn}))\). Specificity measures the proportion of actual negatives that are correctly identified as such \((\text{tn} / (\text{tn} + \text{fp}))\).

This metric creates four local variables, \text{true_positives}, \text{true_negatives}, \text{false_positives} and \text{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 \text{sample_weight} is NULL, weights default to 1. Use \text{sample_weight} of 0 to mask values.

If \text{class_id} is specified, we calculate precision by considering only the entries in the batch for which \text{class_id} is above the threshold predictions, and computing the fraction of them for which \text{class_id} is indeed a correct label.

For additional information about specificity and sensitivity, see the following.

Value

A (subclassed) Metric instance that can be passed directly to compile(metrics = ), or used as a standalone object. See ?Metric for example usage.

See Also

Other metrics: custom_metric(), metric_accuracy(), metric_auc(), metric_binary_accuracy(), metric_binary_crossentropy(), metric_categorical_accuracy(), metric_categorical_crossentropy(), metric_categorical_hinge(), metric_cosine_similarity(), metric_false_negatives(), metric_false_positives(), metric_hinge(), metric_kullback_leibler_divergence(), metric_logcosh_error(), metric_mean_absolute_error(), metric_mean_absolute_percentage_error(), metric_mean_iou(), metric_mean_relative_error(), metric_mean_squared_error(), metric_mean_squared_logarithmic_error(), metric_mean_tensor(), metric_mean_wrapper(), metric_mean(), metric_poisson(), metric_precision_at_recall(), metric_precision(), metric_recall_at_precision(), 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_squared_hinge(), metric_sum(), metric_top_k_categorical_accuracy(), metric_true_negatives(), metric_true_positives()

---

**metric_squared_hinge**

*Computes the squared hinge metric*

Description

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)
**metric_sum**

**Description**
Computes the (weighted) sum of the given values

**Usage**
```R
metric_sum(..., name = NULL, dtype = NULL)
```

**Arguments**
- ...: Passed on to the underlying metric. Used for forwards and backwards compatibility.
- 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 (subclassed) Metric instance is returned. The Metric object can be passed directly to `compile(metrics = )` or used as a standalone object. See `?Metric` for example usage.

Alternatively, if called with `y_true` and `y_pred` arguments, then the computed case-wise values for the mini-batch are returned directly.

**See Also**
Other metrics: `custom_metric()`, `metric_accuracy()`, `metric_auc()`, `metric_binary_accuracy()`, `metric_binary_crossentropy()`, `metric_categorical_accuracy()`, `metric_categorical_crossentropy()`, `metric_categorical_hinge()`, `metric_categorical_slope()`, `metric_categorical_slope_error()`, `metric_categorical_slope_logarithmic_error()`, `metric_categorical_sensitivity_at_specificity()`, `metric_categorical_top_k_categorical_accuracy()`, `metric_categorical_specificity_at_sensitivity()`, `metric_categorical_sum()`, `metric_categorical_top_k_categorical_accuracy()`, `metric_categorical_true_negatives()`, `metric_categorical_true_positives()`, `metric_cosine_similarity()`, `metric_crossentropy()`, `metric_false_negatives()`, `metric_false_positives()`, `metric_hinge()`, `metric_iou()`, `metric_kullback_leibler_divergence()`, `metric_logcosh_error()`, `metric_mean_absolute_error()`, `metric_mean_absolute_percentage_error()`, `metric_mean_iou()`, `metric_mean_relative_error()`, `metric_mean_squared_error()`, `metric_mean_squared_logarithmic_error()`, `metric_mean_tensor()`, `metric_mean()`, `metric_poisson()`, `metric_precision_at_recall()`, `metric_precision()`, `metric_recall_at_precision()`, `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_sum()`, `metric_top_k_categorical_accuracy()`, `metric_true_negatives()`, `metric_true_positives()`
**Details**

For example, if values is `c(1, 3, 5, 7)` then the sum is 16. If the weights were specified as `c(1, 1, 0, 0)` then the sum would be 4.

This metric creates one variable, total, that is used to compute the sum of values. This is ultimately returned as sum.

If `sample_weight` is NULL, weights default to 1. Use `sample_weight` of 0 to mask values.

**Value**

A (subclassed) Metric instance that can be passed directly to `compile(metrics = )`, or used as a standalone object. See `?Metric` for example usage.

**See Also**

Other metrics: `custom_metric()`, `metric_accuracy()`, `metric_auc()`, `metric_binary_accuracy()`, `metric_binary_crossentropy()`, `metric_categorical_accuracy()`, `metric_categorical_crossentropy()`, `metric_categorical_hinge()`, `metric_cosine_similarity()`, `metric_false_negatives()`, `metric_false_positives()`, `metric_hinge()`, `metric_kullback_leibler_divergence()`, `metric_logcosh_error()`, `metric_mean_absolute_error()`, `metric_mean_absolute_percentage_error()`, `metric_mean_iou()`, `metric_mean_relative_error()`, `metric_mean_squared_error()`, `metric_mean_squared_logarithmic_error()`, `metric_mean_tensor()`, `metric_mean_wrapper()`, `metric_mean()`, `metric_poisson()`, `metric_precision_at_recall()`, `metric_precision()`, `metric_recall_at_precision()`, `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_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**

```r
metric_top_k_categorical_accuracy(
  y_true,
  y_pred,
  k = 5L,
  ..., 
  name = "top_k_categorical_accuracy",
  dtype = NULL
)
```
Argument:

- **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.
- **...**: Passed on to the underlying metric. Used for forwards and backwards compatibility.
- **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 (subclassed) Metric instance is returned. The Metric object can be passed directly to `compile(metrics =)` or used as a standalone object. See `?Metric` for example usage.

Alternatively, if called with `y_true` and `y_pred` arguments, then the computed case-wise values for the mini-batch are returned directly.

See Also:

Other metrics: custom_metric(), metric_accuracy(), metric_auc(), metric_binary_accuracy(), metric_binary_crossentropy(), metric_categorical_accuracy(), metric_categorical_crossentropy(), metric_categorical_hinge(), metric_cosine_similarity(), metric_false_negatives(), metric_false_positives(), metric_hinge(), metric_kullback_leibler_divergence(), metric_logcosh_error(), metric_mean_absolute_error(), metric_mean_absolute_percentage_error(), metric_mean_iou(), metric_mean_relative_error(), metric_mean_squared_error(), metric_mean_squared_logarithmic_error(), metric_mean_tensor(), metric_mean_wrapper(), metric_mean(), metric_poisson(), metric_precision_at_recall(), metric_recall_at_precision(), metric_recall(), metric_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:

Calculates the number of true negatives

Usage:

```
metric_true_negatives(..., thresholds = NULL, name = NULL, dtype = NULL)
```
metric_true_positives

**Arguments**

... Passed on to the underlying metric. Used for forwards and backwards compatibility.

```
thresholds  (Optional) Defaults to 0.5. A float value or a 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). 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.
```

**Details**

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.

**Value**

A (subclassed) `Metric` instance that can be passed directly to `compile(metrics = )`, or used as a standalone object. See `?Metric` for example usage.

**See Also**

Other metrics: `custom_metric()`, `metric_accuracy()`, `metric_auc()`, `metric_binary_accuracy()`, `metric_binary_crossentropy()`, `metric_categorical_accuracy()`, `metric_categorical_crossentropy()`, `metric_categorical_hinge()`, `metric_cosine_similarity()`, `metric_false_negatives()`, `metric_false_positives()`, `metric_hinge()`, `metric_kullback_leibler_divergence()`, `metric_logcosh_error()`, `metric_mean_absolute_error()`, `metric_mean_absolute_percentage_error()`, `metric_mean_iou()`, `metric_mean_relative_error()`, `metric_mean_squared_error()`, `metric_mean_squared_logarithmic_error()`, `metric_mean_tensor()`, `metric_mean_wrapper()`, `metric_mean()`, `metric_poisson()`, `metric_precision_at_recall()`, `metric_recall_at_precision()`, `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_positives()`

---

### Description

Calculates the number of true positives

### Usage

```R
metric_true_positives(..., thresholds = NULL, name = NULL, dtype = NULL)
```
model_from_saved_model

Load a Keras model from the Saved Model format

Description

Load a Keras model from the Saved Model format

Usage

model_from_saved_model(saved_model_path, custom_objects = NULL)
Arguments

saved_model_path
   a string specifying the path to the SavedModel directory.

custom_objects
   Optional dictionary mapping string names to custom classes or functions (e.g. custom loss functions).

Value

a Keras model.

Note

This functionality is experimental and only works with TensorFlow version >= "2.0".

See Also

Other saved_model: model_to_saved_model()

model_to_json
   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

model_to_json(object)

custom_objects = NULL)

Arguments

object
   Model object to save

json
   JSON with model configuration

custom_objects
   Optional named list mapping names to custom classes or functions to be considered during deserialization.

See Also

Other model persistence: get_weights(), model_to_yaml(), save_model_hdf5(), save_model_tf(), save_model_weights_hdf5(), serialize_model()
Export to Saved Model format

Description
Export to Saved Model format

Usage
model_to_saved_model(
    model,
    saved_model_path,
    custom_objects = NULL,
    as_text = FALSE,
    input_signature = NULL,
    serving_only = FALSE
)

Arguments
model  A Keras model to be saved. If the model is subclassed, the flag serving_only must be set to TRUE.
saved_model_path a string specifying the path to the SavedModel directory.
custom_objects Optional dictionary mapping string names to custom classes or functions (e.g. custom loss functions).
as_text  bool, FALSE by default. Whether to write the SavedModel proto in text format. Currently unavailable in serving-only mode.
input_signature A possibly nested sequence of tf.TensorSpec objects, used to specify the expected model inputs. See tf.function for more details.
serving_only  bool, FALSE by default. When this is true, only the prediction graph is saved.

Value
Invisibly returns the saved_model_path.

Note
This functionality is experimental and only works with TensorFlow version >= "2.0".

See Also
Other saved_model: model_from_saved_model()
model_to_yaml  

Model configuration as YAML

Description
Save and re-load models configurations as YAML. Note that the representation does not include the weights, only the architecture.

Usage
model_to_yaml(object)

model_from_yaml(yaml, custom_objects = NULL)

Arguments
- object  Model object to save
- yaml  YAML with model configuration
- custom_objects  Optional named list mapping names to custom classes or functions to be considered during deserialization.

See Also
Other model persistence: get_weights(), model_to_json(), save_model_hdf5(), save_model_tf(), save_model_weights_hdf5(), serialize_model()

multi_gpu_model  
Replicates a model on different GPUs.

Description
Replicates a model on different GPUs.

Usage
multi_gpu_model(model, gpus = NULL, cpu_merge = TRUE, cpu_relocation = FALSE)

Arguments
- model  A Keras model instance. To avoid OOM errors, this model could have been built on CPU, for instance (see usage example below).
- gpus  NULL to use all available GPUs (default). Integer >= 2 or list of integers, number of GPUs or list of GPU IDs on which to create model replicas.
- cpu_merge  A boolean value to identify whether to force merging model weights under the scope of the CPU or not.
**multi_gpu_model**

A boolean value to identify whether to create the model’s weights under the scope of the CPU. If the model is not defined under any preceding device scope, you can still rescue it by activating this option.

**Details**

Specifically, this function implements single-machine multi-GPU data parallelism. It works in the following way:

- Divide the model’s input(s) into multiple sub-batches.
- Apply a model copy on each sub-batch. Every model copy is executed on a dedicated GPU.
- Concatenate the results (on CPU) into one big batch.

E.g. if your `batch_size` is 64 and you use `gpus=2`, then we will divide the input into 2 sub-batches of 32 samples, process each sub-batch on one GPU, then return the full batch of 64 processed samples.

This induces quasi-linear speedup on up to 8 GPUs.

This function is only available with the TensorFlow backend for the time being.

**Value**

A Keras model object which can be used just like the initial `model` argument, but which distributes its workload on multiple GPUs.

**Model Saving**

To save the multi-gpu model, use `save_model_hdf5()` or `save_model_weights_hdf5()` with the template model (the argument you passed to `multi_gpu_model`), rather than the model returned by `multi_gpu_model`.

**Note**

This function is deprecated and has been removed from tensorflow on 2020-04-01. To distribute your training across all available GPUs, you can use `tensorflow::tf$distribute$MirroredStrategy()` by creating your model like this:

```r
strategy <- tensorflow::tf$distribute$MirroredStrategy()
with(strategy$scope(), {
  model <- application_xception(
    weights = NULL,
    input_shape = c(height, width, 3),
    classes = num_classes
  )
})
```

**See Also**

Other model functions: `compile.keras.engine.training.Model()`, `evaluate.keras.engine.training.Model()`, `evaluate_generator()`, `fit.keras.engine.training.Model()`, `fit_generator()`, `get_config()`, `get_layer()`, `keras_model_sequential()`, `keras_model()`, `pop_layer()`, `predict.keras.engine.training.Model()`
predict_generator(), predict_on_batch(), predict_proba(), summary.keras.engine.training.Model(), train_on_batch()

Examples

## Not run:

library(keras)
library(tensorflow)

num_samples <- 1000
height <- 224
width <- 224
num_classes <- 1000

# Instantiate the base model (or "template" model).
# We recommend doing this with under a CPU device scope,
# so that the model's weights are hosted on CPU memory.
# Otherwise they may end up hosted on a GPU, which would
# complicate weight sharing.
with(tf$device("/cpu:0"), {
  model <- application_xception(
    weights = NULL,
    input_shape = c(height, width, 3),
    classes = num_classes
  )
})

# Replicates the model on 8 GPUs.
# This assumes that your machine has 8 available GPUs.
parallel_model <- multi_gpu_model(model, gpus = 8)
parallel_model %>% compile(
  loss = "categorical_crossentropy",
  optimizer = "rmsprop"
)

# Generate dummy data.
x <- array(runif(num_samples * height * width*3),
  dim = c(num_samples, height, width, 3))
y <- array(runif(num_samples * num_classes),
  dim = c(num_samples, num_classes))

# This `fit` call will be distributed on 8 GPUs.
# Since the batch size is 256, each GPU will process 32 samples.
parallel_model %>% fit(x, y, epochs = 20, batch_size = 256)

# Save model via the template model (which shares the same weights):
model %>% save_model_hdf5("my_model.h5")

## End(Not run)
normalize

Normalize a matrix or nd-array

Description

Normalize a matrix or nd-array

Usage

normalize(x, axis = -1, order = 2)

Arguments

x Matrix or array to normalize
axis Axis along which to normalize. Axis indexes are 1-based (pass -1 to select the last axis).
order Normalization order (e.g. 2 for L2 norm)

Value

A normalized copy of the array.

optimizer_adadelta

Adadelta optimizer.

Description

Adadelta optimizer as described in ADADELTA: An Adaptive Learning Rate Method.

Usage

optimizer_adadelta(
  learning_rate = 1,
  rho = 0.95,
  epsilon = NULL,
  decay = 0,
  clipnorm = NULL,
  clipvalue = NULL,
  ...
)
### Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>learning_rate</td>
<td>float &gt;= 0. Learning rate.</td>
</tr>
<tr>
<td>rho</td>
<td>float &gt;= 0. Decay factor.</td>
</tr>
<tr>
<td>epsilon</td>
<td>float &gt;= 0. Fuzz factor. If NULL, defaults to <code>k_epsilon()</code>.</td>
</tr>
<tr>
<td>decay</td>
<td>float &gt;= 0. Learning rate decay over each update.</td>
</tr>
<tr>
<td>clipnorm</td>
<td>Gradients will be clipped when their L2 norm exceeds this value.</td>
</tr>
<tr>
<td>clipvalue</td>
<td>Gradients will be clipped when their absolute value exceeds this value.</td>
</tr>
<tr>
<td>...</td>
<td>Unused, present only for backwards compatibility</td>
</tr>
</tbody>
</table>

### Note

It is recommended to leave the parameters of this optimizer at their default values.

### See Also

Other optimizers: `optimizer_adagrad()`, `optimizer_adamax()`, `optimizer_adam()`, `optimizer_nadam()`, `optimizer_rmsprop()`, `optimizer_sgd()`

---

### optimizer_adagrad

*Adagrad optimizer.*

### Description

Adagrad optimizer as described in *Adaptive Subgradient Methods for Online Learning and Stochastic Optimization*.

### Usage

```r
optimizer_adagrad(
    learning_rate = 0.01,
    epsilon = NULL,
    decay = 0,
    clipnorm = NULL,
    clipvalue = NULL,
    ...
)
```

### Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>learning_rate</td>
<td>float &gt;= 0. Learning rate.</td>
</tr>
<tr>
<td>epsilon</td>
<td>float &gt;= 0. Fuzz factor. If NULL, defaults to <code>k_epsilon()</code>.</td>
</tr>
<tr>
<td>decay</td>
<td>float &gt;= 0. Learning rate decay over each update.</td>
</tr>
<tr>
<td>clipnorm</td>
<td>Gradients will be clipped when their L2 norm exceeds this value.</td>
</tr>
<tr>
<td>clipvalue</td>
<td>Gradients will be clipped when their absolute value exceeds this value.</td>
</tr>
<tr>
<td>...</td>
<td>Unused, present only for backwards compatibility</td>
</tr>
</tbody>
</table>
optimizer_adam

**Note**

It is recommended to leave the parameters of this optimizer at their default values.

**See Also**

Other optimizers: `optimizer_adadelta()`, `optimizer_adamax()`, `optimizer_adam()`, `optimizer_nadam()`, `optimizer_rmsprop()`, `optimizer_sgd()`

---

**optimizer_adam**  
*Adam optimizer*

**Description**

Adam optimizer as described in *Adam - A Method for Stochastic Optimization*.

**Usage**

```r
optimizer_adam(
  learning_rate = 0.001,
  beta_1 = 0.9,
  beta_2 = 0.999,
  epsilon = NULL,
  decay = 0,
  amsgrad = FALSE,
  clipnorm = NULL,
  clipvalue = NULL,
  ...
)
```

**Arguments**

- `learning_rate`  
  float >= 0. Learning rate.

- `beta_1`  
  The exponential decay rate for the 1st moment estimates. float, 0 < beta < 1. Generally close to 1.

- `beta_2`  
  The exponential decay rate for the 2nd moment estimates. float, 0 < beta < 1. Generally close to 1.

- `epsilon`  
  float >= 0. Fuzz factor. If NULL, defaults to `k_epsilon()`.

- `decay`  
  float >= 0. Learning rate decay over each update.

- `amsgrad`  
  Whether to apply the AMSGrad variant of this algorithm from the paper "On the Convergence of Adam and Beyond".

- `clipnorm`  
  Gradients will be clipped when their L2 norm exceeds this value.

- `clipvalue`  
  Gradients will be clipped when their absolute value exceeds this value.

- `...`  
  Unused, present only for backwards compatibility.
References

- Adam - A Method for Stochastic Optimization
- On the Convergence of Adam and Beyond

Note

Default parameters follow those provided in the original paper.

See Also

Other optimizers: optimizer_adadelta(), optimizer_adagrad(), optimizer_adamax(), optimizer_nadam(), optimizer_rmsprop(), optimizer_sgd()

optimizer_adamax

Adamax optimizer

Description

Adamax optimizer from Section 7 of the Adam paper. It is a variant of Adam based on the infinity norm.

Usage

optimizer_adamax(
  learning_rate = 0.002,
  beta_1 = 0.9,
  beta_2 = 0.999,
  epsilon = NULL,
  decay = 0,
  clipnorm = NULL,
  clipvalue = NULL,
  ...
)

Arguments

learning_rate float >= 0. Learning rate.
beta_1 The exponential decay rate for the 1st moment estimates. float, 0 < beta < 1. Generally close to 1.
beta_2 The exponential decay rate for the 2nd moment estimates. float, 0 < beta < 1. Generally close to 1.
epsilon float >= 0. Fuzz factor. If NULL, defaults to k_epsilon().
decay float >= 0. Learning rate decay over each update.
clipnorm Gradients will be clipped when their L2 norm exceeds this value.
clipvalue Gradients will be clipped when their absolute value exceeds this value.
... Unused, present only for backwards compatibility
See Also

Other optimizers: optimizer_adadelta(), optimizer_adagrad(), optimizer_adam(), optimizer_nadam(), optimizer_rmsprop(), optimizer_sgd()

optimizer_nadam  Nesterov Adam optimizer

Description

Much like Adam is essentially RMSprop with momentum, Nadam is Adam RMSprop with Nesterov momentum.

Usage

```r
optimizer_nadam(
  learning_rate = 0.002,
  beta_1 = 0.9,
  beta_2 = 0.999,
  epsilon = NULL,
  schedule_decay = 0.004,
  clipnorm = NULL,
  clipvalue = NULL,
  ...
)
```

Arguments

- **learning_rate** (float >= 0.): Learning rate.
- **beta_1** (float, 0 < beta < 1.): The exponential decay rate for the 1st moment estimates. Generally close to 1.
- **beta_2** (float, 0 < beta < 1.): The exponential decay rate for the 2nd moment estimates. Generally close to 1.
- **epsilon** (float >= 0.): Fuzz factor. If NULL, defaults to k_epsilon().
- **schedule_decay** (Schedule decay).
- **clipnorm** (Gradients will be clipped when their L2 norm exceeds this value.)
- **clipvalue** (Gradients will be clipped when their absolute value exceeds this value.)
- **...** (Unused, present only for backwards compatibility)

Details

Default parameters follow those provided in the paper. It is recommended to leave the parameters of this optimizer at their default values.
optimizer_rmsprop

See Also

On the importance of initialization and momentum in deep learning.

Other optimizers: optimizer_adadelta(), optimizer_adagrad(), optimizer_adamax(), optimizer_adam(), optimizer_rmsprop(), optimizer_sgd()

optimizer_rmsprop   RMSProp optimizer

Description

RMSProp optimizer

Usage

optimizer_rmsprop(
    learning_rate = 0.001,
    rho = 0.9,
    epsilon = NULL,
    decay = 0,
    clipnorm = NULL,
    clipvalue = NULL,
    ...
)

Arguments

learning_rate   float >= 0. Learning rate.
rho             float >= 0. Decay factor.
epsilon         float >= 0. Fuzz factor. If NULL, defaults to k_epsilon().
decay           float >= 0. Learning rate decay over each update.
clipnorm        Gradients will be clipped when their L2 norm exceeds this value.
clipvalue       Gradients will be clipped when their absolute value exceeds this value.
...              Unused, present only for backwards compatibility

Note

It is recommended to leave the parameters of this optimizer at their default values (except the learning rate, which can be freely tuned).
This optimizer is usually a good choice for recurrent neural networks.

See Also

Other optimizers: optimizer_adadelta(), optimizer_adagrad(), optimizer_adamax(), optimizer_adam(), optimizer_nadam(), optimizer_sgd()
Stochastic gradient descent optimizer with support for momentum, learning rate decay, and Nesterov momentum.

Usage

```r
optimizer_sgd(
  learning_rate = 0.01,
  momentum = 0,
  decay = 0,
  nesterov = FALSE,
  clipnorm = NULL,
  clipvalue = NULL,
  ...
)
```

Arguments

- `learning_rate` float >= 0. Learning rate.
- `momentum` float >= 0. Parameter that accelerates SGD in the relevant direction and dampens oscillations.
- `decay` float >= 0. Learning rate decay over each update.
- `nesterov` boolean. Whether to apply Nesterov momentum.
- `clipnorm` Gradients will be clipped when their L2 norm exceeds this value.
- `clipvalue` Gradients will be clipped when their absolute value exceeds this value.
- `...` Unused, present only for backwards compatibility

Value

Optimizer for use with `compile.keras.engine.training.Model`.

See Also

Other optimizers: `optimizer_adadelta()`, `optimizer_adagrad()`, `optimizer_adamax()`, `optimizer_adam()`, `optimizer_nadam()`, `optimizer_rmsprop()`
Description

Pads sequences to the same length

Usage

```python
pad_sequences(
    sequences,
    maxlen = NULL,
    dtype = "int32",
    padding = "pre",
    truncating = "pre",
    value = 0
)
```

Arguments

- `sequences`: List of lists where each element is a sequence
- `maxlen`: int, maximum length of all sequences
- `dtype`: type of the output sequences
- `padding`: 'pre' or 'post', pad either before or after each sequence.
- `truncating`: 'pre' or 'post', remove values from sequences larger than maxlen either in the beginning or in the end of the sequence
- `value`: float, padding value

Details

This function transforms a list of `num_samples` sequences (lists of integers) into a matrix of shape `(num_samples, num_timesteps)`. `num_timesteps` is either the `maxlen` argument if provided, or the length of the longest sequence otherwise.

Sequences that are shorter than `num_timesteps` are padded with `value` at the end.

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 is the default.

Value

Matrix with dimensions `(number_of_sequences, maxlen)`
plot.keras_training_history

Plot training history

Description

Plots metrics recorded during training.

Usage

```r
## 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.engine.training.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.

See Also

Other text preprocessing: `make_sampling_table()`, `skipgrams()`, `text_hashing_trick()`, `text_one_hot()`, `text_to_word_sequence()`
**pop_layer**

Remove the last layer in a model

**Description**

Remove the last layer in a model

**Usage**

```r
pop_layer(object)
```

**Arguments**

- `object` Keras model object

**See Also**


---

**predict.keras.engine.training.Model**

Generate predictions from a Keras model

**Description**

Generates output predictions for the input samples, processing the samples in a batched way.

**Usage**

```r
## S3 method for class 'keras.engine.training.Model'
predict(
  object,
  x,
  batch_size = NULL,
  verbose = 0,
  steps = NULL,
  callbacks = NULL,
  ...
)
```
predict_generator

Arguments

- **object**: Keras model
- **x**: Input data (vector, matrix, or array). You can also pass a tfdataset or a generator returning a list with (inputs, targets) or (inputs, targets, sample_weights).
- **batch_size**: Integer. If unspecified, it will default to 32.
- **verbose**: Verbosity mode, 0 or 1.
- **steps**: Total number of steps (batches of samples) before declaring the evaluation round finished. Ignored with the default value of NULL.
- **callbacks**: List of callbacks to apply during prediction. Unused

Value

- vector, matrix, or array of predictions

See Also

Other model functions: `compile.keras.engine.training.Model()`, `evaluate.keras.engine.training.Model()`, `evaluate_generator()`, `fit.keras.engine.training.Model()`, `fit_generator()`, `get_config()`, `get_layer()`, `keras_model_sequential()`, `keras_model()`, `multi_gpu_model()`, `pop_layer()`, `predict_generator()`, `predict_on_batch()`, `predict_proba()`, `summary.keras.engine.training.Model()`, `train_on_batch()`

---

**predict_generator** generates predictions for the input samples from a data generator.

**Description**

The generator should return the same kind of data as accepted by `predict_on_batch()`.

**Usage**

```r
predict_generator(
  object,
  generator,
  steps,
  max_queue_size = 10,
  workers = 1,
  verbose = 0,
  callbacks = NULL
)
```
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 (vector, matrix, or array). You can also pass a `tfdataset` or a generator returning a list with (inputs, targets) or (inputs, targets, sample_weights).
**Value**
array of predictions.

**See Also**

---

**predict_proba**

*(Deprecated) Generates probability or class probability predictions for the input samples.*

---

**Description**
These functions were removed in Tensorflow version 2.6. See details for how to update your code:

**Usage**

```r
predict_proba(object, x, batch_size = NULL, verbose = 0, steps = NULL)
predict_classes(object, x, batch_size = NULL, verbose = 0, steps = NULL)
```

**Arguments**

- **object**: Keras model object
- **x**: Input data (vector, matrix, or array). You can also pass a `tfdataset` or a generator returning a list with (inputs, targets) or (inputs, targets, sample_weights).
- **batch_size**: Integer. If unspecified, it will default to 32.
- **verbose**: Verbosity mode, 0 or 1.
- **steps**: Total number of steps (batches of samples) before declaring the evaluation round finished. The default `NULL` is equal to the number of samples in your dataset divided by the batch size.

**Details**

How to update your code:

- `predict_proba()`: use `predict()` directly.
- `predict_classes()`:
  - If your model does multi-class classification: (e.g. if it uses a `softmax` last-layer activation).
    ```r
    model %>% predict(x) %>% k_argmax()
    ```
• if your model does binary classification (e.g. if it uses a sigmoid last-layer activation).

```r
model %>% predict(x) %>% `>`(0.5) %>% k_cast("int32")
```

The input samples are processed batch by batch.

See Also

Other model functions: `compile.keras.engine.training.Model()`, `evaluate.keras.engine.training.Model()`, `evaluate_generator()`, `fit.keras.engine.training.Model()`, `fit_generator()`, `get_config()`, `get_layer()`, `keras_model_sequential()`, `keras_model()`, `multi_gpu_model()`, `pop_layer()`, `predict.keras.engine.training.Model()`, `predict_generator()`, `predict_on_batch()`, `summary.keras.engine.training.Model()`, `train_on_batch()`

---

**regularizer_l1**  
*L1 and L2 regularization*

**Description**

L1 and L2 regularization

**Usage**

```r
regularizer_l1(l = 0.01)
regularizer_l2(l = 0.01)
regularizer_l1_l2(l1 = 0.01, l2 = 0.01)
```

**Arguments**

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Regularization factor.</td>
</tr>
<tr>
<td>l1</td>
<td>L1 regularization factor.</td>
</tr>
<tr>
<td>l2</td>
<td>L2 regularization factor.</td>
</tr>
</tbody>
</table>
reset_states  

Reset the states for a layer

Description

Reset the states for a layer

Usage

reset_states(object)

Arguments

object  
Model or layer object

See Also

Other layer methods: count_params(), get_config(), get_input_at(), get_weights()
Details

The following components of the model are saved:

- The model architecture, allowing to re-instantiate the model.
- The model weights.
- The state of the optimizer, allowing to resume training exactly where you left off. This allows you to save the entirety of the state of a model in a single file.

Saved models can be reinstated via `load_model_hdf5()`. The model returned by `load_model_hdf5()` is a compiled model ready to be used (unless the saved model was never compiled in the first place or `compile = FALSE` is specified).

As an alternative to providing the `custom_objects` argument, you can execute the definition and persistence of your model using the `with_custom_object_scope()` function.

Note

The `serialize_model()` function enables saving Keras models to R objects that can be persisted across R sessions.

See Also

Other model persistence: `get_weights()`, `model_to_json()`, `model_to_yaml()`, `save_model_tf()`, `save_model_weights_hdf5()`, `serialize_model()`

---

**save_model_tf**  
Save/Load models using SavedModel format

**Description**

Save/Load models using SavedModel format

**Usage**

```r
save_model_tf(
  object,
  filepath,
  overwrite = TRUE,
  include_optimizer = TRUE,
  signatures = NULL,
  options = NULL
)
```

```r
load_model_tf(filepath, custom_objects = NULL, compile = TRUE)
```
save_model_weights_hdf5

Arguments

- **object**
  - Model object to save

- **filepath**
  - File path

- **overwrite**
  - Overwrite existing file if necessary

- **include_optimizer**
  - If TRUE, save optimizer's state.

- **signatures**
  - Signatures to save with the SavedModel. Please see the signatures argument in tf$saved_model$save for details.

- **options**
  - Optional tf$saved_model$SaveOptions object that specifies options for saving to SavedModel

- **custom_objects**
  - Mapping class names (or function names) of custom (non-Keras) objects to class/functions (for example, custom metrics or custom loss functions). This mapping can be done with the dict() function of reticulate.

- **compile**
  - Whether to compile the model after loading.

See Also

Other model persistence: `get_weights()`, `model_to_json()`, `model_to_yaml()`, `save_model_hdf5()`, `save_model_weights_hdf5()`, `serialize_model()`

---

save_model_weights_hdf5

Save/Load model weights using HDF5 files

Description

Save/Load model weights using HDF5 files

Usage

```r
save_model_weights_hdf5(object, filepath, overwrite = TRUE)

load_model_weights_hdf5(
  object,
  filepath,
  by_name = FALSE,
  skip_mismatch = FALSE,
  reshape = FALSE
)
```
Arguments

- **object**: Model object to save/load
- **filepath**: Path to the file
- **overwrite**: Whether to silently overwrite any existing file at the target location
- **by_name**: Whether to load weights by name or by topological order.
- **skip_mismatch**: Logical, whether to skip loading of layers where there is a mismatch in the number of weights, or a mismatch in the shape of the weight (only valid when `by_name = FALSE`).
- **reshape**: Reshape weights to fit the layer when the correct number of values are present but the shape does not match.

Details

The weight file has:

- `layer_names` (attribute), a list of strings (ordered names of model layers).
- For every layer, a group named `layer.name`
- For every such layer group, a group attribute `weight_names`, a list of strings (ordered names of weights tensor of the layer).
- For every weight in the layer, a dataset storing the weight value, named after the weight tensor.

For `load_model_weights()`, if `by_name` is `FALSE` (default) weights are loaded based on the network’s topology, meaning 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.

If `by_name` is `TRUE`, weights are loaded into layers only if they share the same name. This is useful for fine-tuning or transfer-learning models where some of the layers have changed.

See Also

Other model persistence: `get_weights()`, `model_to_json()`, `model_to_yaml()`, `save_model_hdf5()`, `save_model_tf()`, `serialize_model()`
Usage

```r
save_model_weights_tf(object, filepath, overwrite = TRUE)

load_model_weights_tf(
  object,
  filepath,
  by_name = FALSE,
  skip_mismatch = FALSE,
  reshape = FALSE
)
```

Arguments

- **object**  Model object to save/load
- **filepath**  Path to the file
- **overwrite**  Whether to silently overwrite any existing file at the target location
- **by_name**  Whether to load weights by name or by topological order.
- **skip_mismatch**  Logical, whether to skip loading of layers where there is a mismatch in the number of weights, or a mismatch in the shape of the weight (only valid when `by_name = FALSE`).
- **reshape**  Reshape weights to fit the layer when the correct number of values are present but the shape does not match.

Details

When saving in TensorFlow format, all objects referenced by the network are saved in the same format as `tf.train.Checkpoint`, including any Layer instances or Optimizer instances assigned to `object` attributes. For networks constructed from inputs and outputs using `tf.keras.Model(inputs, outputs)`, Layer instances used by the network are tracked/saved automatically. For user-defined classes which inherit from `tf.keras.Model`, Layer instances must be assigned to `object` attributes, typically in the constructor.

See the documentation of `tf.train.Checkpoint` and `tf.keras.Model` for details.

---

**save_text_tokenizer**  
*Save a text tokenizer to an external file*

Description

Enables persistence of text tokenizers alongside saved models.

Usage

```r
save_text_tokenizer(object, filename)

load_text_tokenizer(filename)
```
sequences_to_matrix

Convert a list of sequences into a matrix.

Description

Convert a list of sequences into a matrix.

Usage

sequences_to_matrix(
  tokenizer,
  sequences,
  mode = c("binary", "count", "tfidf", "freq")
)
Arguments

tokenizer  
sequences  
mode  

Value

A matrix

See Also

Other text tokenization: `fit_text_tokenizer()`, `save_text_tokenizer()`, `text_tokenizer()`, `texts_to_matrix()`, `texts_to_sequences_generator()`, `texts_to_sequences()`

Description

`sequential_model_input_layer`

Usage

```
sequential_model_input_layer(
  input_shape = NULL,
  batch_size = NULL,
  dtype = NULL,
  input_tensor = NULL,
  sparse = NULL,
  name = NULL,
  ragged = NULL,
  type_spec = NULL,
  ...,
  input_layer_name = NULL
)
```

Arguments

- `input_shape`  
- `batch_size`  
- `dtype`  
- `input_tensor`  
- `sparse`  
- `name`  
- `ragged`  
- `type_spec`  
- `input_layer_name`  

- `input_shape`  
  - an integer vector of dimensions (not including the batch axis), or a `tf$TensorShape` instance (also not including the batch axis).
- `batch_size`  
  - Optional input batch size (integer or NULL).
- `dtype`  
  - Optional datatype of the input. When not provided, the Keras default float type will be used.
- `input_tensor`  
  - Optional tensor to use as layer input. If set, the layer will use the `tf$TypeSpec` of this tensor rather than creating a new placeholder tensor.
serialize_model

sparse Boolean, whether the placeholder created is meant to be sparse. Default to FALSE.
ragged Boolean, whether the placeholder created is meant to be ragged. In this case, values of 'NULL' in the 'shape' argument represent ragged dimensions. For more information about RaggedTensors, see this guide. Default to FALSE.
type_spec A tf$TypeSpec object to create Input from. This tf$TypeSpec represents the entire batch. When provided, all other args except name must be NULL.
... additional arguments passed on to keras$layers$InputLayer.
input_layer_name name
Optional name of the input layer (string).

serialize_model | Serialize a model to an R object

Description
Model objects are external references to Keras objects which cannot be saved and restored across R sessions. The serialize_model() and unserialize_model() functions provide facilities to convert Keras models to R objects for persistence within R data files.

Usage
serialize_model(model, include_optimizer = TRUE)
unserialize_model(model, custom_objects = NULL, compile = TRUE)

Arguments
model Keras model or R "raw" object containing serialized Keras model.
include_optimizer If TRUE, save optimizer's state.
custom_objects Mapping class names (or function names) of custom (non-Keras) objects to class/functions (for example, custom metrics or custom loss functions). This mapping can be done with the dict() function of reticulate.
compile Whether to compile the model after loading.

Value
serialize_model() returns an R "raw" object containing an hdf5 version of the Keras model.
unserialize_model() returns a Keras model.

Note
The save_model_hdf5() function enables saving Keras models to external hdf5 files.
set_vocabulary

Sets vocabulary (and optionally document frequency) data for the layer

Description

This method sets the vocabulary and DF data for this layer directly, instead of analyzing a dataset through adapt(). It should be used whenever the vocab (and optionally document frequency) information is already known. If vocabulary data is already present in the layer, this method will either replace it, if prepend is set to FALSE, or append to it (if ‘append’ is set to TRUE)

Usage

set_vocabulary(
  object,
  vocab,
  idf_weights = NULL,
  df_data = NULL,
  oov_df_value = FALSE,
  append = NULL
)

Arguments

- **object**: a text vectorization layer
- **vocab**: An array of string tokens.
- **idf_weights**: An array of document frequency data with equal length to vocab. Only necessary if the layer output_mode is TFIDF.
- **df_data**: An array of document frequency data. Only necessary if the layer output_mode is “tfidf”.
- **oov_df_value**: The document frequency of the OOV token. Only necessary if output_mode is "tfidf". OOV data is optional when appending additional data in "tfidf" mode; if an OOV value is supplied it will overwrite the existing OOV value.
- **append**: Whether to overwrite or append any existing vocabulary data. (deprecated since TensorFlow >= 2.3)

See Also

- get_vocabulary()
skipgrams  

Generates skipgram word pairs.

Description

Generates skipgram word pairs.

Usage

```r
skipgrams(
  sequence,
  vocabulary_size,
  window_size = 4,
  negative_samples = 1,
  shuffle = TRUE,
  categorical = FALSE,
  sampling_table = NULL,
  seed = NULL
)
```

Arguments

- `sequence` A word sequence (sentence), encoded as a list of word indices (integers). If using a `sampling_table`, word indices are expected to match the rank of the words in a reference dataset (e.g. 10 would encode the 10-th most frequently occurring token). Note that index 0 is expected to be a non-word and will be skipped.
- `vocabulary_size` Int, maximum possible word index + 1
- `window_size` Int, size of sampling windows (technically half-window). The window of a word \( w_i \) will be \([i-\text{window}\_\text{size}, i+\text{window}\_\text{size}+1]\)
- `negative_samples` float \( \geq 0 \). 0 for no negative (i.e. random) samples. 1 for same number as positive samples.
- `shuffle` whether to shuffle the word couples before returning them.
- `categorical` bool. if FALSE, labels will be integers (eg. [0, 1, 1 .. ]). if TRUE labels will be categorical eg. [[1,0],[0,1],[0,1] .. ]
- `sampling_table` 1D array of size `vocabulary_size` where the entry \( i \) encodes the probability to sample a word of rank \( i \).
- `seed` Random seed

Details

This function transforms a list of word indexes (lists of integers) into lists of words of the form:

- (word, word in the same window), with label 1 (positive samples).
• (word, random word from the vocabulary), with label 0 (negative samples).

Read more about Skipgram in this gnomic paper by Mikolov et al.: Efficient Estimation of Word Representations in Vector Space

Value

List of couples, labels where:

• couples is a list of 2-element integer vectors: \([\text{word\_index}, \text{other\_word\_index}]\).
• labels is an integer vector of 0 and 1, where 1 indicates that \(\text{other\_word\_index}\) was found in the same window as \(\text{word\_index}\), and 0 indicates that \(\text{other\_word\_index}\) was random.
• if categorical is set to TRUE, the labels are categorical, ie. 1 becomes \([0,1]\), and 0 becomes \([1,0]\).

See Also

Other text preprocessing: make_sampling_table(), pad_sequences(), text_hashing_trick(), text_one_hot(), text_to_word_sequence()
**texts_to_matrix**

*Convert a list of texts to a matrix.*

**Description**

Convert a list of texts to a matrix.

**Usage**

```r
texts_to_matrix(tokenizer, texts, mode = c("binary", "count", "tfidf", "freq"))
```

**Arguments**

- `tokenizer` : Tokenizer
- `texts` : Vector/list of texts (strings).
- `mode` : one of "binary", "count", "tfidf", "freq".

**Value**

A matrix

**See Also**

Other text tokenization: `fit_text_tokenizer()`, `save_text_tokenizer()`, `sequences_to_matrix()`, `text_tokenizer()`, `texts_to_sequences_generator()`, `texts_to_sequences()`

---

**texts_to_sequences**

*Transform each text in texts in a sequence of integers.*

**Description**

Only top "num_words" most frequent words will be taken into account. Only words known by the tokenizer will be taken into account.

**Usage**

```r
texts_to_sequences(tokenizer, texts)
```

**Arguments**

- `tokenizer` : Tokenizer
- `texts` : Vector/list of texts (strings).

**See Also**

Other text tokenization: `fit_text_tokenizer()`, `save_text_tokenizer()`, `sequences_to_matrix()`, `text_tokenizer()`, `texts_to_sequences_generator()`, `texts_to_sequences()`
texts_to_sequences_generator

Transforms each text in texts in a sequence of integers.

Description

Only top "num_words" most frequent words will be taken into account. Only words known by the tokenizer will be taken into account.

Usage

    texts_to_sequences_generator(tokenizer, texts)

Arguments

- tokenizer: Tokenizer
- texts: Vector/list of texts (strings).

Value

Generator which yields individual sequences

See Also

Other text tokenization: fit_text_tokenizer(), save_text_tokenizer(), sequences_to_matrix(), text_tokenizer(), texts_to_matrix(), texts_to_sequences()

text_hashing_trick

Converts a text to a sequence of indexes in a fixed-size hashing space.

Description

Converts a text to a sequence of indexes in a fixed-size hashing space.

Usage

    text_hashing_trick(
        text,
        n,
        hash_function = NULL,
        filters = "!"#$%&()*+,-./:;<=>?@[\]^_\`{|}~\t\n",
        lower = TRUE,
        split = " ",
    )
Arguments

- **text**: Input text (string).
- **n**: Dimension of the hashing space.
- **hash_function**: If NULL uses Python hash function, can be 'md5' or any function that takes in input a string and returns an int. Note that hash is not a stable hashing function, so it is not consistent across different runs, while 'md5' is a stable hashing function.
- **filters**: Sequence of characters to filter out such as punctuation. Default includes basic punctuation, tabs, and newlines.
- **lower**: Whether to convert the input to lowercase.
- **split**: Sentence split marker (string).

Details

Two or more words may be assigned to the same index, due to possible collisions by the hashing function.

Value

A list of integer word indices (unicity non-guaranteed).

See Also

Other text preprocessing: `make_sampling_table()`, `pad_sequences()`, `skipgrams()`, `text_one_hot()`, `text_to_word_sequence()`

Description

One-hot encode a text into a list of word indexes in a vocabulary of size n.

Usage

```R
text_one_hot(
  input_text,
  n,
  filters = "!"#$%&()*+,-./:;<=>?@[\]^`{|}~\t\n",
  lower = TRUE,
  split = " ",
  text = NULL
)
```
Arguments

- **input_text**: Input text (string).
- **n**: Size of vocabulary (integer).
- **filters**: Sequence of characters to filter out such as punctuation. Default includes basic punctuation, tabs, and newlines.
- **lower**: Whether to convert the input to lowercase.
- **split**: Sentence split marker (string).

Value

List of integers in [1, n]. Each integer encodes a word (unicity non-guaranteed).

See Also

Other text preprocessing: `make_sampling_table()`, `pad_sequences()`, `skipgrams()`, `text_hashing_trick()`, `text_to_word_sequence()`

---

### text_tokenizer

**Text tokenization utility**

**Description**

Vectorize a text corpus, by turning each text into either a sequence of integers (each integer being the index of a token in a dictionary) or into a vector where the coefficient for each token could be binary, based on word count, based on tf-idf...

**Usage**

```r
text_tokenizer(
  num_words = NULL,
  filters = "!"#$%&()*+,-./:;<=>?@[^\]^\_\~/\~\t\n",
  lower = TRUE,
  split = " ",
  char_level = FALSE,
  oov_token = NULL
)
```

**Arguments**

- **num_words**
  - the maximum number of words to keep, based on word frequency. Only the most common `num_words` words will be kept.
- **filters**
  - a string where each element is a character that will be filtered from the texts. The default is all punctuation, plus tabs and line breaks, minus the ’ character.
- **lower**
  - boolean. Whether to convert the texts to lowercase.
text_to_word_sequence

Convert text to a sequence of words (or tokens).

Description

Convert text to a sequence of words (or tokens).

Usage

```r
text_to_word_sequence(
  text,
  filters = "!"#$%&()+,-./:;<=?>@\]^_\{\}~\t\n",
  lower = TRUE,
  split = " "
)
```
timeseries_generator

Argument

text      Input text (string).
filters   Sequence of characters to filter out such as punctuation. Default includes basic punctuation, tabs, and newlines.
lower     Whether to convert the input to lowercase.
split     Sentence split marker (string).

Value

Words (or tokens)

See Also

Other text preprocessing: make_sampling_table(), pad_sequences(), skipgrams(), text_hashing_trick(), text_one_hot()

---

timeseries_generator  Utility function for generating batches of temporal data.

Description

Utility function for generating batches of temporal data.

Usage

timeseries_generator(
  data,
  targets,
  length,
  sampling_rate = 1,
  stride = 1,
  start_index = 0,
  end_index = NULL,
  shuffle = FALSE,
  reverse = FALSE,
  batch_size = 128
)

Arguments

data      Object containing consecutive data points (timesteps). The data should be 2D, and axis 1 is expected to be the time dimension.
targets   Targets corresponding to timesteps in data. It should have same length as data.
length     Length of the output sequences (in number of timesteps).
sampling_rate  Period between successive individual timesteps within sequences. For rate $r$, timesteps data[i], data[i-$r$], ..., data[i-length] are used for creating a sample sequence.

stride  Period between successive output sequences. For stride $s$, consecutive output samples would be centered around data[i], data[i+$s$], data[i+2*$s$], etc.

start_index, end_index  Data points earlier than start_index or later than end_index will not be used in the output sequences. This is useful to reserve part of the data for test or validation.

shuffle  Whether to shuffle output samples, or instead draw them in chronological order.

reverse  Boolean: if true, timesteps in each output sample will be in reverse chronological order.

batch_size  Number of timeseries samples in each batch (except maybe the last one).

**Value**

An object that can be passed to generator based training functions (e.g. `fit_generator()`).

---

**time_distributed**  
*Apply a layer to every temporal slice of an input.*

**Description**

The input should be at least 3D, and the dimension of index one will be considered to be the temporal dimension.

**Usage**

```python
    time_distributed(
        object,
        layer,
        input_shape = NULL,
        batch_input_shape = NULL,
        batch_size = NULL,
        dtype = NULL,
        name = NULL,
        trainable = NULL,
        weights = NULL
    )
```

**Arguments**

- **object**  Model or layer object
- **layer**  A layer instance.
- **input_shape**  Dimensionality of the input (integer) not including the samples axis. This argument is required when using this layer as the first layer in a model.
to_categorical

batch_input_shape
Shapes, including the batch size. For instance, batch_input_shape=c(10, 32) indicates that the expected input will be batches of 10 32-dimensional vectors.
batch_input_shape=list(NULL, 32) indicates batches of an arbitrary number of 32-dimensional vectors.

batch_size
Fixed batch size for layer

dtype
The data type expected by the input, as a string (float32, float64, int32...)

name
An 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.

trainable
Whether the layer weights will be updated during training.

weights
Initial weights for layer.

Details
Consider a batch of 32 samples, where each sample is a sequence of 10 vectors of 16 dimensions. The batch input shape of the layer is then (32, 10, 16), and the input_shape, not including the samples dimension, is (10, 16). You can then use time_distributed to apply a layer_dense to each of the 10 timesteps, independently.

See Also
Other layer wrappers: bidirectional()

---

to_categorical

Converts a class vector (integers) to binary class matrix.

Description
Converts a class vector (integers) to binary class matrix.

Usage
to_categorical(y, num_classes = NULL, dtype = "float32")

Arguments

- **y**: Class vector to be converted into a matrix (integers from 0 to num_classes).
- **num_classes**: Total number of classes.
- **dtype**: The data type expected by the input, as a string

Details
E.g. for use with loss_categorical_crossentropy().

Value
A binary matrix representation of the input.
Description

Single gradient update or model evaluation over one batch of samples.

Usage

\[
\text{train\_on\_batch}(\text{object}, \text{x}, \text{y}, \text{class\_weight} = \text{NULL}, \text{sample\_weight} = \text{NULL})
\]

\[
\text{test\_on\_batch}(\text{object}, \text{x}, \text{y}, \text{sample\_weight} = \text{NULL})
\]

Arguments

- **object**: Keras model object
- **x**: input data, as an array or list of arrays (if the model has multiple inputs).
- **y**: labels, as an array.
- **class\_weight**: named list mapping classes to a weight value, used for scaling the loss function (during training only).
- **sample\_weight**: sample weights, as an array.

Value

Scalar training or test loss (if the model has no metrics) or list of scalars (if the model computes other metrics). The property `model$metrics_names` will give you the display labels for the scalar outputs.

See Also


**use_implementation**  
*Select a Keras implementation and backend*

**Description**
Select a Keras implementation and backend

**Usage**
```
use_implementation(implementation = c("keras", "tensorflow"))
```
```
use_backend(backend = c("tensorflow", "cntk", "theano", "plaidml"))
```

**Arguments**
- **implementation**  One of "keras" or "tensorflow" (defaults to "keras").
- **backend**  One of "tensorflow", "cntk", or "theano" (defaults to "tensorflow")

**Details**
Keras has multiple implementations (the original keras implementation and the implementation native to TensorFlow) and supports multiple backends ("tensorflow", "cntk", "theano", and "plaidml"). These functions allow switching between the various implementations and backends.

The functions should be called after `library(keras)` and before calling other functions within the package (see below for an example).

The default implementation and backend should be suitable for most use cases. The "tensorflow" implementation is useful when using Keras in conjunction with TensorFlow Estimators (the **tfestimators** R package).

**Examples**
```
## Not run:
# use the tensorflow implementation
library(keras)
use_implementation("tensorflow")

# use the cntk backend
library(keras)
use_backend("theano")
```
```
## End(Not run)
```
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.

Examples

```r
## Not run:
# 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
  save_model_hdf5("my_model.h5")
}
```

# loading the model within the custom object scope doesn't
# require explicitly providing the custom_object
load_model_hdf5("my_model.h5")
}

## End(Not run)

---

**%py_class%**  
 *Make a python class constructor*

**Description**  
Make a python class constructor

**Usage**  
`spec %py_class% body`

**Arguments**  
- `spec`  
  a bare symbol `MyClassName`, or a call `MyClassName(SuperClass)`
- `body`  
  an expression that can be evaluated to construct the class methods.

**Value**  
The python class constructor, invisibly. Note, the same constructor is also assigned in the parent frame.

**Examples**  
```r
## Not run:
MyClass %py_class% {
    initialize <- function(x) {
        print("Hi from MyClass$initialize()!")
        self$x <- x
    }
    my_method <- function() {
        self$x
    }
}
my_class_instance <- MyClass(42)
my_class_instance$my_method()

MyClass2(MyClass) %py_class% {
    "This will be a __doc__ string for MyClass2"
}```
initialize <- function(...) {
    "This will be the __doc__ string for the MyClass2.__init__() method"
    print("Hi from MyClass2$initialize()!")
    super$initialize(...)  
}  

my_class_instance2 <- MyClass2(42)  
my_class_instance2$my_method()  
reticulate::py_help(MyClass2)  # see the __doc__ strings and more!  

## End(Not run)
Index

* activation layers
  layer_activation, 200
  layer_activation_elu, 201
  layer_activation_leaky_relu, 203
  layer_activation_parametric_relu, 204
  layer_activation_relu, 205
  layer_activation_selu, 206
  layer_activation_softmax, 207
  layer_activation_thresholded_relu, 208

* attention layers
  layer_attention, 213

* callbacks
  callback_csv_logger, 29
  callback_early_stopping, 30
  callback_lambda, 31
  callback_learning_rate_scheduler, 32
  callback_model_checkpoint, 33
  callback_progbar_logger, 34
  callback_reduce_lr_on_plateau, 35
  callback_remote_monitor, 36
  callback_tensorboard, 37
  callback_terminate_on_naan, 38

* convolutional layers
  layer_conv_1d, 223
  layer_conv_1d_transpose, 225
  layer_conv_2d, 228
  layer_conv_2d_transpose, 230
  layer_conv_3d, 233
  layer_conv_3d_transpose, 236
  layer_conv_1stm_2d, 238
  layer_cropping_1d, 241
  layer_cropping_2d, 242
  layer_cropping_3d, 244
  layer_depthwise_conv_2d, 252
  layer_separable_conv_1d, 297
  layer_separable_conv_2d, 300
  layer_upsampling_1d, 312
  layer_upsampling_2d, 313
  layer_upsampling_3d, 315
  layer_zero_padding_1d, 316
  layer_zero_padding_2d, 317
  layer_zero_padding_3d, 319

* core layers
  layer_activation, 200
  layer_activity_regularization, 209
  layer_attention, 213
  layer_dense, 249
  layer_dense_features, 251
  layer_dropout, 255
  layer_flatten, 258
  layer_input, 272
  layer_lambda, 273
  layer_masking, 284
  layer_permute, 294
  layer_repeat_vector, 295
  layer_reshape, 296

* datasets
  dataset_boston_housing, 46
  dataset_cifar10, 47
  dataset_cifar100, 48
  dataset_fashion_mnist, 48
  dataset_imdb, 49
  dataset_mnist, 51
  dataset_reuters, 51
  keras, 92

* dropout layers
  layer_dropout, 255
  layer_spatial_dropout_1d, 306
  layer_spatial_dropout_2d, 307
  layer_spatial_dropout_3d, 308

* image preprocessing
  fit_image_data_generator, 60
  flow_images_from_data, 61
  flow_images_from_dataframe, 63
  flow_images_from_directory, 65
image_load, 78
image_to_array, 79

* initializers
  - initializer_constant, 81
  - initializer_glorot_normal, 81
  - initializer_glorot_uniform, 82
  - initializer_he_normal, 82
  - initializer_he_uniform, 83
  - initializer_identity, 84
  - initializer_lecun_normal, 84
  - initializer_lecun_uniform, 85
  - initializer_ones, 85
  - initializer_orthogonal, 86
  - initializer_random_normal, 87
  - initializer_random_uniform, 87
  - initializer_truncated_normal, 88
  - initializer_variance_scaling, 88
  - initializer_zeros, 89

* layer methods
  - count_params, 43
  - get_config, 69
  - get_input_at, 71
  - get_weights, 73
  - reset_states, 393

* layer wrappers
  - bidirectional, 28
  - time_distributed, 410

* locally connected layers
  - layer_locally_connected_1d, 275
  - layer_locally_connected_2d, 278

* merge layers
  - layer_add, 211
  - layer_average, 214
  - layer_concatenate, 222
  - layer_dot, 254
  - layer_maximum, 285
  - layer_minimum, 290
  - layer_multiply, 291
  - layer_subtract, 310

* metrics
  - custom_metric, 45
  - metric_accuracy, 327
  - metric_auc, 328
  - metric_binary_accuracy, 330
  - metric_binary_crossentropy, 332
  - metric_categorical_accuracy, 333
  - metric_categorical_crossentropy, 334
  - metric_categorical_hinge, 336
  - metric_cosine_similarity, 337
  - metric_false_negatives, 338
  - metric_false_positives, 339
  - metric_hinge, 340
  - metric_kullback_leibler_divergence, 341
  - metric_logcosh_error, 343
  - metric_mean, 344
  - metric_mean_absolute_error, 345
  - metric_mean_absolute_percentage_error, 346
  - metric_mean_iou, 347
  - metric_mean_relative_error, 349
  - metric_mean_squared_error, 350
  - metric_mean_squared_logarithmic_error, 351
  - metric_mean_tensor, 352
  - metric_mean_wrapper, 353
  - metric_poisson, 354
  - metric_precision, 355
  - metric_precision_at_recall, 357
  - metric_recall, 358
  - metric_recall_at_precision, 359
  - metric_root_mean_squared_error, 361
  - metric_sensitivity_at_specificity, 362
  - metric_sparse_categorical_accuracy, 363
  - metric_sparse_categorical_crossentropy, 364
  - metric_sparse_top_k_categorical_accuracy, 366
  - metric_specificity_at_sensitivity, 367
  - metric_squared_hinge, 368
  - metric_sum, 369
  - metric_top_k_categorical_accuracy, 370
  - metric_true_negatives, 371
  - metric_true_positives, 372

* model functions
  - compile.keras.engine.training.Model, 39
  - evaluate.keras.engine.training.Model, 53
  - evaluate_generator, 54
fit.keras.engine.training.Model, 56
fit_generator, 58
get_config, 69
get_layer, 72
keras_model, 97
keras_model_sequential, 99
multi_gpu_model, 376
pop_layer, 388
predict.keras.engine.training.Model, 388
predict_generator, 389
predict_on_batch, 390
predict_proba, 391
summary.keras.engine.training.Model, 403
train_on_batch, 412
* model persistence
  get_weights, 73
  model_to_json, 374
  model_to_yaml, 376
  save_model_hdf5, 393
  save_model_tf, 394
  save_model_weights_hdf5, 395
  serialize_model, 400
* noise layers
  layer_alpha_dropout, 212
  layer_gaussian_dropout, 259
  layer_gaussian_noise, 260
* optimizers
  optimizer_adadelta, 379
  optimizer_adagrad, 380
  optimizer_adam, 381
  optimizer_adamax, 382
  optimizer_nadam, 383
  optimizer_rmsprop, 384
  optimizer_sgd, 385
* pooling layers
  layer_average_pooling_1d, 215
  layer_average_pooling_2d, 216
  layer_average_pooling_3d, 218
  layer_global_average_pooling_1d, 261
  layer_global_average_pooling_2d, 262
  layer_global_average_pooling_3d, 263
  layer_global_max_pooling_1d, 264
  layer_global_max_pooling_2d, 265
  layer_global_max_pooling_3d, 267
  layer_max_pooling_1d, 286
  layer_max_pooling_2d, 287
  layer_max_pooling_3d, 288
* recurrent layers
  layer_cudnn_gru, 245
  layer_cudnn_lstm, 247
  layer_gru, 268
  layer_lstm, 280
  layer_simple_rnn, 303
* saved_model
  model_from_saved_model, 373
  model_to_saved_model, 375
* text preprocessing
  make_sampling_table, 325
  pad_sequences, 386
  skipgrams, 402
  text_hashing_trick, 405
  text_one_hot, 406
  text_to_word_sequence, 408
* text tokenization
  fit_text_tokenizer, 61
  save_text_tokenizer, 397
  sequences_to_matrix, 398
  text_tokenizer, 407
  texts_to_matrix, 404
  texts_to_sequences, 404
  texts_to_sequences_generator, 405
%py_class%, 415
_Package (keras-package), 10
activationelu(activation_relu), 11
activationexponential
(activation_relu), 11
activationelu(activation_relu), 11
activationhardsigmoid
(activation_relu), 11
activationlinear(activation_relu), 11
activationselu(activation_relu), 11
activationsigmoid(activation_relu), 11
activationsoftmax(activation_relu), 11
activationsoftplus(activation_relu), 11
activationsoftsign(activation_relu), 11
activationswish(activation_relu), 11
activationsinh(activation_relu), 11
adapt, 13
adapt(), 401
application_densenet, 14
application_densenet121
  (application_densenet), 14
application_densenet169
  (application_densenet), 14
application_densenet201
  (application_densenet), 14
application_inception_resnet_v2, 15
application_inception_v3, 17
application_mobilenet, 18
application_mobilenet_v2, 20
application_nasnet, 21
application_nasnetlarge
  (application_nasnet), 21
application_nasnetmobile
  (application_nasnet), 21
application_resnet50, 23
application_vgg, 25
application_vgg16 (application_vgg), 25
application_vgg19 (application_vgg), 25
application_xception, 26
backend, 28
backend(), 42
bidirectional, 28, 411
binary_crossentropy, (loss-functions), 320
callback_csv_logger, 29, 31–36, 38
callback_early_stopping, 30, 30, 32–36, 38
callback_lambda, 30, 31, 33–36, 38
callback_learning_rate_scheduler, 30–32, 32, 34–36, 38
callback_model_checkpoint, 30–33, 33–36, 38
callback_progbar_logger, 30–34, 34, 35, 36, 38
callback_reduce_lr_on_plateau, 30–34, 35, 36, 38
callback_remote_monitor, 30–35, 36, 38
callback_tensorboard, 30–36, 37, 38
callback_terminate_on_naan, 30–36, 38, 38
clone_model, 39
compile(), 53, 57, 59
compile.keras.engine.training.Model, 39, 54, 55, 58, 60, 69, 72, 97, 100, 377, 385, 388–392, 403, 412
compile.keras.engine.training.Model(), 323, 324
count_params, 43, 69, 71, 73, 393
dataset_boston_housing, 46, 47–49, 51, 52
dataset_cifar10, 47, 47, 48, 49, 51, 52
dataset_cifar100, 47, 48, 49, 51, 52
dataset_fashion_mnist, 47, 48, 49, 51, 52
dataset_imdb, 47–49, 49, 51, 52
dataset_imdb(), 51, 52
dataset_imdb_word_index(dataset_imdb), 49
dataset_mnist, 47–49, 51, 51
dataset_reuters, 47–49, 51
dataset_reuters_word_index(dataset_reuters), 51
denseret_preprocess_input
  (application_densenet), 14
evaluate.keras.engine.training.Model, 41, 53, 55, 58, 60, 69, 72, 97, 100, 377, 388–392, 403, 412
evaluate_generator, 41, 54, 54, 58, 60, 69, 72, 97, 100, 377, 388–392, 403, 412
evaluate_generator(). 66
export_savedmodel.keras.engine.training.Model, 55
fit.keras.engine.training.Model, 41, 54, 55, 56, 60, 69, 72, 97, 100, 377, 388–392, 403, 412
fit_generator, 41, 54, 55, 58, 58, 69, 72, 97, 100, 377, 388–392, 403, 412
fit_generator(). 410
INDEX

fit_image_data_generator, 60, 62, 65, 67, 79, 80
fit_text_tokenizer, 61, 398, 399, 404, 405, 408
fit_text_tokenizer(), 398
flow_images_from_data, 60, 61, 65, 67, 79, 80
flow_images_from_dataframe, 60, 62, 63, 67, 79, 80
flow_images_from_directory, 60, 62, 65, 67, 79, 80
flow_images_from_directory(), 58
freeze_weights, 67
from_config (get_config), 69
generator_next, 68
generator_next(), 70
generator_next(), 69
get_config, 41, 43, 54, 55, 58, 60, 69, 71–73, 97, 100, 377, 388–393, 403, 412
get_file, 70
get_input_at, 43, 69, 71, 73, 393
get_input_mask_at(get_input_at), 71
generator_next(), 71
generator_next(), 71
generator_next(), 71
generator_next(), 71
generator_next(), 71
generator_next(), 71
generator_next(), 71
get_layer, 41, 54, 55, 58, 60, 69, 72, 97, 100, 377, 388–392, 403, 412
get_output_at(get_input_at), 71
get_output_mask_at(get_input_at), 71
get_output_shape_at(get_input_at), 71
get_vocabulary, 72
get_vocabulary(), 401
get_weights, 43, 69, 71, 73, 374, 376, 393–396, 401

hdf5_matrix, 73

image_array_resize (image_to_array), 79
image_array_save (image_to_array), 79
image_data_generator, 76
image_data_generator(), 60, 62, 64, 66, 68
image_dataset_from_directory, 75
image_load, 60, 62, 65, 67, 78, 80
image_to_array, 60, 62, 65, 67, 79, 79
imagenet_decode_predictions, 74
imagenet_preprocess_input, 74
implementation, 80
inception_resnet_v2_preprocess_input
  (application_inception_resnet_v2), 15
inception_v3_preprocess_input
  (application_inception_v3), 17
initialize_constant, 81, 82–89
initialize_glorot_normal, 81, 82–89
initialize_glorot_uniform, 81, 82, 83–89
initialize_he_normal, 81, 82, 83–89
initialize_he_uniform, 81–83, 84–89
initialize_identity, 81–83, 84, 85–89
initialize_l2_normal, 81–84, 85–89, 207
initialize_l2_normal, 81–85, 85, 86–89
initialize_orthogonal, 81–86, 86, 87–89
initialize_random_normal, 81–86, 87, 88, 89
initialize_random_normal(), 88
initialize_random_uniform, 81–87, 88, 89
initialize_truncated_normal, 81–88, 88, 89
initialize_variance_scaling, 81–88, 88, 89
initialize_zeros, 81–89, 89
install_keras, 90
is_keras_available, 91
k_abs, 100
k_all, 101
k_any, 102
k_arange, 102
k_argmax, 103
k_argmin, 104
k_backend, 104
k_batch_dot, 105
k_batch_flatten, 106
k_batch_get_value, 106
k_batch_get_value(), 108
k_batch_normalization, 107
k_batch_set_value, 108
k_batch_set_value(), 107
k_bias_add, 108
k_binary_crossentropy, 109
k_cast, 110
k_cast_to_floatx, 110
k_categorical_crossentropy, 111
k_clear_session, 112
k_clip, 112
k_concatenate, 113
k_constant, 113
layer_conv_2d(), 278
layer_conv_lstm_2d, 225, 228, 230, 233, 235, 238, 238, 242, 243, 245, 254, 299, 302, 313, 315–318, 320
layer_cropping_1d, 225, 228, 230, 233, 235, 238, 241, 244, 243, 245, 254, 299, 302, 313, 315–318, 320
layer_cropping_3d, 225, 228, 230, 233, 235, 238, 241–243, 244, 254, 299, 302, 313, 315–318, 320
layer_cudnn_gru, 245, 249, 271, 284, 306
layer_cudnn_lstm, 247, 247, 271, 284, 306
layer_dense, 201, 210, 214, 249, 252, 256, 258, 272, 274, 285, 295–297
layer_dense_features, 201, 210, 214, 251, 251, 256, 258, 272, 274, 285, 295–297
layer_dot, 211, 215, 222, 254, 285, 290, 291, 310
layer_embedding, 256
layer_flatten, 201, 210, 214, 251, 252, 256, 258, 272, 274, 285, 295–297
layer_gaussian_dropout, 213, 259, 261
layer_gaussian_noise, 213, 260, 260
layer_global_average_pooling_1d, 216, 217, 219, 261, 263–266, 268, 287–289
layer_global_average_pooling_2d, 216, 217, 219, 262, 262, 264–266, 268, 287–289
layer_global_average_pooling_3d, 216, 217, 219, 262, 263, 263, 265, 266, 268, 287–289
layer_global_max_pooling_1d, 216, 217, 219, 262–264, 264, 266, 268, 287–289
layer_global_max_pooling_2d, 216, 217, 219, 262–265, 265, 268, 287–289
layer_global_max_pooling_3d, 216, 217, 219, 262–266, 267, 287–289
layer_gru, 247, 249, 268, 284, 306
layer_input, 201, 210, 214, 251, 252, 256, 258, 272, 274, 285, 295–297
layer_lstm, 247, 249, 271, 280, 306
layer_lambda, 201, 210, 214, 251, 252, 256, 258, 272, 273, 285, 295–297
layer_layer_normalization, 274
layer_locally_connected_1d, 275, 280
layer_locally_connected_2d, 277, 278
layer_lstm, 247, 249, 271, 280, 306
layer_masking, 201, 210, 214, 251, 252, 256, 258, 272, 274, 285, 295–297
layer_max_pooling_1d, 216, 217, 219, 262–266, 268, 286, 288, 289
layer_max_pooling_2d, 216, 217, 219, 262–266, 268, 287, 289
layer_max_pooling_3d, 216, 217, 219, 262–266, 268, 288, 288
layer_maximum, 211, 215, 222, 255, 285, 290, 291, 310
layer_minimum, 211, 215, 222, 255, 285, 290, 291, 310
layer_multi_head_attention, 292
layer_multiply, 211, 215, 222, 255, 285, 290, 291, 310
layer_permute, 201, 210, 214, 251, 252, 256, 258, 272, 274, 285, 294, 296, 297
layer_repeat_vector, 201, 210, 214, 251, 252, 256, 258, 272, 274, 285, 295, 295, 297
layer_reshape, 201, 210, 214, 251, 252, 256, 258, 272, 274, 285, 295, 296, 296
layer_separable_conv_1d, 225, 228, 230, 233, 235, 238, 241–243, 245, 254, 297, 302, 313, 315–318, 320
layer_simple_rnn, 247, 249, 271, 284, 303
INDEX

layer_spatial_dropout_1d, 256, 306, 308, 309
layer_spatial_dropout_2d, 256, 307, 307, 309
layer_spatial_dropout_3d, 256, 307, 308, 308
layer_subtract, 211, 215, 222, 255, 285, 290, 291, 310
layer_text_vectorization, 311
load_model_hdf5 (save_model_hdf5), 293
load_model_hdf5(), 46, 398
load_model_tf (save_model_tf), 394
load_model_weights_hdf5 (save_model_weights_hdf5), 395
load_model_weights_tf (save_model_weights_tf), 396
load_text_tokenizer (save_text_tokenizer), 397
loss-functions, 320
loss_binary_crossentropy (loss-functions), 320
loss_binary_crossentropy(), 324
loss_categorical_crossentropy (loss-functions), 320
loss_categorical_crossentropy(), 411
loss_categorical_hinge (loss-functions), 320
loss_cosine_proximity, 324
loss_cosine_similarity (loss-functions), 320
loss_cosine_similarity(), 324
loss_hinge (loss-functions), 320
loss_huber (loss-functions), 320
loss_kl_divergence (loss-functions), 320
loss_kullback_leibler_divergence (loss-functions), 320
loss_logcosh (loss-functions), 320
loss_mean_absolute_error (loss-functions), 320
loss_mean_absolute_percentage_error (loss-functions), 320
loss_mean_squared_error (loss-functions), 320
loss_mean_squared_logarithmic_error (loss-functions), 320
loss_poisson (loss-functions), 320
loss_sparse_categorical_crossentropy (loss-functions), 320
loss_squared_hinge (loss-functions), 320
make_sampling_table, 325, 387, 403, 406, 407, 409
Metric, 326
metric_cosine_proximity, 324
metric_cosine_similarity (loss-functions), 320
metric_mean_wrapper(), 46
metric_sparse_categorical_accuracy, 46
INDEX

mobilenet_decode_predictions (application_mobilenet), 18
mobilenet_load_model_hdf5 (application_mobilenet), 18
mobilenet_preprocess_input (application_mobilenet), 18
mobilenet_v2_decode_predictions (application_mobilenet_v2), 20
mobilenet_v2_load_model_hdf5 (application_mobilenet_v2), 20
mobilenet_v2_preprocess_input (application_mobilenet_v2), 20
model_from_json (model_to_json), 374
model_from_saved_model, 373, 375
model_from_yaml (model_to_yaml), 376
model_to_json, 73, 374, 376, 394–396, 401
model_to_sAVED_MODEL, 374, 375
model_to_yaml, 73, 374, 376, 394–396, 401
multi_gpu_model, 41, 54, 55, 58, 60, 69, 72, 97, 100, 376, 388–392, 403, 412
nasnet_preprocess_input (application_nasnet), 21
normalize, 379
optimizer_adadelta, 379, 381–385
optimizer_adagrad, 380, 381, 382–385
optimizer_adam, 380, 381, 383–385
optimizer_adamax, 380–382, 382, 384, 385
optimizer_nadam, 380–383, 383, 384, 385
optimizer_rmsprop, 380–384, 384, 385
optimizer_sgd, 380–384, 385
pad_sequences, 325, 386, 403, 406, 407, 409
plot(), 387
plot.keras_training_history, 387
pop_layer, 41, 54, 55, 58, 60, 69, 72, 97, 100, 377, 388, 389–392, 403, 412
predict.keras.engine.training.Model, 41, 54, 55, 58, 60, 69, 72, 97, 100, 377, 388, 389, 390–392, 403, 412
predict_classes (predict_proba), 391
predict_generator, 41, 54, 55, 58, 60, 69, 72, 97, 100, 377, 388, 389, 390–392, 403, 412
predict_generator(), 66
predict_on_batch, 41, 54, 55, 58, 60, 69, 72, 97, 100, 378, 388–390, 390, 392, 403, 412
predict_proba, 41, 54, 55, 58, 60, 69, 72, 97, 100, 378, 388–391, 391, 403, 412
py_class (%py_class%), 415
py_to_r(), 28
R6Class, 93–96
regularizer_l1_l2, 392
regularizer_l1_l2 (regularizer_l1), 392
regularizer_l2 (regularizer_l1), 392
reset_states, 43, 69, 71, 73, 393
reticulate::conda_install(), 90
reticulate::py_install(), 91
reticulate::virtualenv_install(), 90
<table>
<thead>
<tr>
<th>Function/Method</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>save_model_hdf5</td>
<td>73, 374, 376, 393, 395, 396, 401</td>
</tr>
<tr>
<td>save_model_hdf5()</td>
<td>42, 94, 377, 400</td>
</tr>
<tr>
<td>save_model_tf</td>
<td>73, 374, 376, 394, 396, 401</td>
</tr>
<tr>
<td>save_model_weights_hdf5</td>
<td>73, 374, 376, 394, 395, 396, 401</td>
</tr>
<tr>
<td>save_model_weights_hdf5()</td>
<td>42, 94, 377</td>
</tr>
<tr>
<td>save_model_weights_tf</td>
<td>396</td>
</tr>
<tr>
<td>save_text_tokenizer</td>
<td>61, 397, 399, 404, 405, 408</td>
</tr>
<tr>
<td>sequences_to_matrix</td>
<td>61, 398, 398, 404, 405, 408</td>
</tr>
<tr>
<td>sequences_to_matrix()</td>
<td>61</td>
</tr>
<tr>
<td>sequential_model_input_layer</td>
<td>99, 399</td>
</tr>
<tr>
<td>serialize_model</td>
<td>73, 374, 376, 394–396, 400</td>
</tr>
<tr>
<td>serialize_model()</td>
<td>394</td>
</tr>
<tr>
<td>set_vocabulary</td>
<td>401</td>
</tr>
<tr>
<td>set_vocabulary()</td>
<td>72</td>
</tr>
<tr>
<td>set_weights()</td>
<td>73</td>
</tr>
<tr>
<td>skipgrams</td>
<td>325, 387, 402, 406, 407, 409</td>
</tr>
<tr>
<td>skipgrams()</td>
<td>325</td>
</tr>
<tr>
<td>summary.keras.engine.training.Model</td>
<td>41, 54, 55, 58, 60, 69, 72, 97, 100, 378, 388–392, 403, 412</td>
</tr>
<tr>
<td>tensorflow::install_tensorflow()</td>
<td>90, 91</td>
</tr>
<tr>
<td>test_on_batch()</td>
<td>412</td>
</tr>
<tr>
<td>text_hashing_trick</td>
<td>325, 387, 403, 405, 407, 409</td>
</tr>
<tr>
<td>text_one_hot()</td>
<td>325, 387, 403, 406, 406, 409</td>
</tr>
<tr>
<td>text_to_word_sequence</td>
<td>325, 387, 403, 406, 407, 408</td>
</tr>
<tr>
<td>text_tokenizer</td>
<td>61, 398, 399, 404, 405, 408</td>
</tr>
<tr>
<td>texts_to_matrix</td>
<td>61, 398, 399, 404, 405, 408</td>
</tr>
<tr>
<td>texts_to_matrix()</td>
<td>61</td>
</tr>
<tr>
<td>texts_to_sequences</td>
<td>61, 398, 399, 404, 405, 408</td>
</tr>
<tr>
<td>texts_to_sequences()</td>
<td>61</td>
</tr>
<tr>
<td>texts_to_sequences_generator</td>
<td>61, 398, 399, 404, 405, 408</td>
</tr>
<tr>
<td>time_distributed</td>
<td>29, 410</td>
</tr>
<tr>
<td>timeseries_generator</td>
<td>409</td>
</tr>
<tr>
<td>to_categorical</td>
<td>411</td>
</tr>
<tr>
<td>to_categorical()</td>
<td>324</td>
</tr>
</tbody>
</table>