Package ‘keras’

May 23, 2022

Type Package
Title R Interface to ‘Keras’
Version 2.9.0
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 (>= 3.4)
Imports generics (>= 0.0.1), reticulate (> 1.22), tensorflow (>= 2.8.0), tfruns (>= 1.0), magrittr, zeallot, glue, methods, R6, ellipsis, rlang
Suggests ggplot2, testthat (>= 2.1.0), knitr, rmarkdown, callr, tfdatasets, withr, png, jpeg
RoxygenNote 7.2.0
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
- activation_relu
- adapt
- application_densenet
- application_efficientnet
- application_inception_resnet_v2
- application_inception_v3
- application_mobilenet
- application_mobilenet_v2
- application_mobilenet_v3
- application_nasnet
- application_resnet
- application_vgg
- application_xception
- backend
- bidirectional
- 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
- callbackTerminate_on_naan
- clone_model
- compile.keras.engine.training.Model
- constraints
- count_params
- create_layer
- create_layer_wrapper
- custom_metric
- dataset_boston_housing
- dataset_cifar10
- dataset_cifar100
- dataset_fashion_mnist
- dataset_imdb
- dataset_mnist
- dataset_reuters
- evaluate.keras.engine.training.Model
- export_savedmodel.keras.engine.training.Model
<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>k_backend</td>
<td>108</td>
</tr>
<tr>
<td>k_batch_dot</td>
<td>108</td>
</tr>
<tr>
<td>k_batch_flatten</td>
<td>109</td>
</tr>
<tr>
<td>k_batch_get_value</td>
<td>110</td>
</tr>
<tr>
<td>k_batch_normalization</td>
<td>110</td>
</tr>
<tr>
<td>k_batch_set_value</td>
<td>111</td>
</tr>
<tr>
<td>k_bias_add</td>
<td>112</td>
</tr>
<tr>
<td>k_binary_crossentropy</td>
<td>112</td>
</tr>
<tr>
<td>k_cast</td>
<td>113</td>
</tr>
<tr>
<td>k_cast_to_floatx</td>
<td>114</td>
</tr>
<tr>
<td>k_categorical_crossentropy</td>
<td>114</td>
</tr>
<tr>
<td>k_clear_session</td>
<td>115</td>
</tr>
<tr>
<td>k_clip</td>
<td>115</td>
</tr>
<tr>
<td>k_concatenate</td>
<td>116</td>
</tr>
<tr>
<td>k_constant</td>
<td>117</td>
</tr>
<tr>
<td>k_conv1d</td>
<td>117</td>
</tr>
<tr>
<td>k_conv2d</td>
<td>118</td>
</tr>
<tr>
<td>k_conv2d_transpose</td>
<td>119</td>
</tr>
<tr>
<td>k_conv3d</td>
<td>120</td>
</tr>
<tr>
<td>k_conv3d_transpose</td>
<td>121</td>
</tr>
<tr>
<td>k_cos</td>
<td>122</td>
</tr>
<tr>
<td>k_count_params</td>
<td>122</td>
</tr>
<tr>
<td>k_ctc_batch_cost</td>
<td>123</td>
</tr>
<tr>
<td>k_ctc_decode</td>
<td>124</td>
</tr>
<tr>
<td>k_ctc_label_dense_to_sparse</td>
<td>125</td>
</tr>
<tr>
<td>k_cumprod</td>
<td>125</td>
</tr>
<tr>
<td>k_cumsum</td>
<td>126</td>
</tr>
<tr>
<td>k_depthwise_conv2d</td>
<td>127</td>
</tr>
<tr>
<td>k_dot</td>
<td>128</td>
</tr>
<tr>
<td>k_dropout</td>
<td>128</td>
</tr>
<tr>
<td>k_dtype</td>
<td>129</td>
</tr>
<tr>
<td>k_elu</td>
<td>130</td>
</tr>
<tr>
<td>k_epsilon</td>
<td>130</td>
</tr>
<tr>
<td>k_equal</td>
<td>131</td>
</tr>
<tr>
<td>k_eval</td>
<td>131</td>
</tr>
<tr>
<td>k_exp</td>
<td>132</td>
</tr>
<tr>
<td>k_expand_dims</td>
<td>133</td>
</tr>
<tr>
<td>k_eye</td>
<td>133</td>
</tr>
<tr>
<td>k_flatten</td>
<td>134</td>
</tr>
<tr>
<td>k_floatx</td>
<td>135</td>
</tr>
<tr>
<td>k_foldl</td>
<td>135</td>
</tr>
<tr>
<td>k_folding</td>
<td>136</td>
</tr>
<tr>
<td>k_function</td>
<td>137</td>
</tr>
<tr>
<td>k_gather</td>
<td>137</td>
</tr>
<tr>
<td>k_get_session</td>
<td>138</td>
</tr>
<tr>
<td>k_get_uid</td>
<td>139</td>
</tr>
<tr>
<td>k_get_value</td>
<td>139</td>
</tr>
<tr>
<td>k_get_variable_shape</td>
<td>140</td>
</tr>
<tr>
<td>topic</td>
<td>page</td>
</tr>
<tr>
<td>--------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>k_gradients</td>
<td>140</td>
</tr>
<tr>
<td>k_greater</td>
<td>141</td>
</tr>
<tr>
<td>k_greater_equal</td>
<td>142</td>
</tr>
<tr>
<td>k_hard_sigmoid</td>
<td>142</td>
</tr>
<tr>
<td>k_identity</td>
<td>143</td>
</tr>
<tr>
<td>k_image_data_format</td>
<td>143</td>
</tr>
<tr>
<td>k_int_shape</td>
<td>144</td>
</tr>
<tr>
<td>k_in_test_phase</td>
<td>145</td>
</tr>
<tr>
<td>k_in_top_k</td>
<td>145</td>
</tr>
<tr>
<td>k_in_train_phase</td>
<td>146</td>
</tr>
<tr>
<td>k_is_keras_tensor</td>
<td>147</td>
</tr>
<tr>
<td>k_is_placeholder</td>
<td>147</td>
</tr>
<tr>
<td>k_is_sparse</td>
<td>148</td>
</tr>
<tr>
<td>k_is_tensor</td>
<td>148</td>
</tr>
<tr>
<td>k_l2_normalize</td>
<td>149</td>
</tr>
<tr>
<td>k_learning_phase</td>
<td>150</td>
</tr>
<tr>
<td>k_less</td>
<td>150</td>
</tr>
<tr>
<td>k_less_equal</td>
<td>151</td>
</tr>
<tr>
<td>k_local_conv1d</td>
<td>151</td>
</tr>
<tr>
<td>k_local_conv2d</td>
<td>152</td>
</tr>
<tr>
<td>k_log</td>
<td>153</td>
</tr>
<tr>
<td>k_manual_variable_initialization</td>
<td>154</td>
</tr>
<tr>
<td>k_map_fn</td>
<td>154</td>
</tr>
<tr>
<td>k_max</td>
<td>155</td>
</tr>
<tr>
<td>k_maximum</td>
<td>156</td>
</tr>
<tr>
<td>k_mean</td>
<td>156</td>
</tr>
<tr>
<td>k_min</td>
<td>157</td>
</tr>
<tr>
<td>k_minimum</td>
<td>158</td>
</tr>
<tr>
<td>k_moving_average_update</td>
<td>158</td>
</tr>
<tr>
<td>k_ndim</td>
<td>159</td>
</tr>
<tr>
<td>k_normalize_batch_in_training</td>
<td>160</td>
</tr>
<tr>
<td>k_not_equal</td>
<td>160</td>
</tr>
<tr>
<td>k_ones</td>
<td>161</td>
</tr>
<tr>
<td>k_ones_like</td>
<td>162</td>
</tr>
<tr>
<td>k_one_hot</td>
<td>162</td>
</tr>
<tr>
<td>k_permute_dimensions</td>
<td>163</td>
</tr>
<tr>
<td>k_placeholder</td>
<td>164</td>
</tr>
<tr>
<td>k_pool2d</td>
<td>165</td>
</tr>
<tr>
<td>k_pool3d</td>
<td>166</td>
</tr>
<tr>
<td>k_pow</td>
<td>167</td>
</tr>
<tr>
<td>k_print_tensor</td>
<td>167</td>
</tr>
<tr>
<td>k_prod</td>
<td>168</td>
</tr>
<tr>
<td>k_random_binomial</td>
<td>169</td>
</tr>
<tr>
<td>k_random_normal</td>
<td>169</td>
</tr>
<tr>
<td>k_random_normal_variable</td>
<td>170</td>
</tr>
<tr>
<td>k_random_uniform</td>
<td>171</td>
</tr>
<tr>
<td>k_random_uniform_variable</td>
<td>172</td>
</tr>
<tr>
<td>k_relu</td>
<td>173</td>
</tr>
<tr>
<td>Function</td>
<td>Page</td>
</tr>
<tr>
<td>-----------------------------------------------</td>
<td>--------</td>
</tr>
<tr>
<td>k_repeat</td>
<td>173</td>
</tr>
<tr>
<td>k_repeat_elements</td>
<td>174</td>
</tr>
<tr>
<td>k_reset_uids</td>
<td>175</td>
</tr>
<tr>
<td>k_reshape</td>
<td>175</td>
</tr>
<tr>
<td>k_resize_images</td>
<td>176</td>
</tr>
<tr>
<td>k_resize_volumes</td>
<td>176</td>
</tr>
<tr>
<td>k_reverse</td>
<td>177</td>
</tr>
<tr>
<td>k_rnn</td>
<td>178</td>
</tr>
<tr>
<td>k_round</td>
<td>179</td>
</tr>
<tr>
<td>k_separable_conv2d</td>
<td>179</td>
</tr>
<tr>
<td>k_set_learning_phase</td>
<td>180</td>
</tr>
<tr>
<td>k_set_value</td>
<td>181</td>
</tr>
<tr>
<td>k_shape</td>
<td>181</td>
</tr>
<tr>
<td>k_sigmoid</td>
<td>182</td>
</tr>
<tr>
<td>k_sign</td>
<td>183</td>
</tr>
<tr>
<td>k_sin</td>
<td>183</td>
</tr>
<tr>
<td>k_softmax</td>
<td>184</td>
</tr>
<tr>
<td>k_softplus</td>
<td>185</td>
</tr>
<tr>
<td>k.softsign</td>
<td>185</td>
</tr>
<tr>
<td>k_sparse_categorical_crossentropy</td>
<td>186</td>
</tr>
<tr>
<td>kSpatial_2d_paddning</td>
<td>187</td>
</tr>
<tr>
<td>kSpatial_3d_paddning</td>
<td>187</td>
</tr>
<tr>
<td>k_sqrt</td>
<td>188</td>
</tr>
<tr>
<td>k_square</td>
<td>189</td>
</tr>
<tr>
<td>k.squeeze</td>
<td>189</td>
</tr>
<tr>
<td>k_stack</td>
<td>190</td>
</tr>
<tr>
<td>k_std</td>
<td>191</td>
</tr>
<tr>
<td>k_stop_gradient</td>
<td>191</td>
</tr>
<tr>
<td>k_sum</td>
<td>192</td>
</tr>
<tr>
<td>k_switch</td>
<td>193</td>
</tr>
<tr>
<td>k_tanh</td>
<td>193</td>
</tr>
<tr>
<td>k_temporal_padding</td>
<td>194</td>
</tr>
<tr>
<td>k_tile</td>
<td>195</td>
</tr>
<tr>
<td>k_to_dense</td>
<td>195</td>
</tr>
<tr>
<td>k.transpose</td>
<td>196</td>
</tr>
<tr>
<td>k_truncated_normal</td>
<td>196</td>
</tr>
<tr>
<td>k_unstack</td>
<td>197</td>
</tr>
<tr>
<td>k_update</td>
<td>198</td>
</tr>
<tr>
<td>k_update_add</td>
<td>198</td>
</tr>
<tr>
<td>k_update_sub</td>
<td>199</td>
</tr>
<tr>
<td>k_var</td>
<td>200</td>
</tr>
<tr>
<td>k_variable</td>
<td>200</td>
</tr>
<tr>
<td>k_zeros</td>
<td>201</td>
</tr>
<tr>
<td>k_zeros_like</td>
<td>202</td>
</tr>
<tr>
<td>layer_activation</td>
<td>202</td>
</tr>
<tr>
<td>layer_activation_elu</td>
<td>203</td>
</tr>
<tr>
<td>layer_activation_leaky_relu</td>
<td>205</td>
</tr>
<tr>
<td>layer_activation_parametric_relu</td>
<td>206</td>
</tr>
</tbody>
</table>
layer_activation_relu .......................................................... 207
layer_activation_selu .......................................................... 209
layer_activation_softmax ..................................................... 210
layer_activation_thresholded_relu ......................................... 211
layer_activity_regularization .............................................. 212
layer_add ........................................................................... 214
layer_additive_attention ..................................................... 214
layer_alpha_dropout ........................................................... 215
layer_attention ................................................................. 216
layer_average ................................................................. 217
layer_average_pooling_1d ..................................................... 218
layer_average_pooling_2d ..................................................... 219
layer_average_pooling_3d ..................................................... 221
layer_batch_normalization ................................................... 222
layer_category_encoding ..................................................... 225
layer_center_crop .............................................................. 226
layer_concatenate .............................................................. 227
layer_conv_1d ................................................................. 228
layer_conv_1d_transpose ..................................................... 231
layer_conv_2d ................................................................. 233
layer_conv_2d_transpose ..................................................... 236
layer_conv_3d ................................................................. 239
layer_conv_3d_transpose ..................................................... 242
layer_conv_lstm_1d ............................................................ 244
layer_conv_lstm_2d ............................................................ 247
layer_conv_lstm_3d ............................................................ 250
layer_cropping_1d ............................................................. 253
layer_cropping_2d ............................................................. 254
layer_cropping_3d ............................................................. 255
layer_dense ................................................................. 257
layer_dense_features .......................................................... 259
layer_depthwise_conv_1d ..................................................... 260
layer_depthwise_conv_2d ..................................................... 262
layer_discretization ........................................................... 265
layer_dot ................................................................. 266
layer_dropout ................................................................. 267
layer_embedding ............................................................... 268
layer_flatten ................................................................. 270
layer_gaussian_dropout ...................................................... 271
layer_gaussian_noise .......................................................... 272
layer_global_average_pooling_1d ........................................... 273
layer_global_average_pooling_2d .......................................... 275
layer_global_average_pooling_3d .......................................... 276
layer_global_max_pooling_1d ............................................... 277
layer_global_max_pooling_2d ............................................... 278
layer_global_max_pooling_3d ............................................... 279
layer_gru ................................................................. 280
layer_gru_cell ............................................................... 284
layer_hashing ................................................................. 286
layer_input ................................................................. 288
layer_integer_lookup ..................................................... 289
layer_lambda ............................................................... 291
layer_normalization ...................................................... 293
layer_locally_connected_1d ............................................. 294
layer_locally_connected_2d ............................................. 297
layer_lstm ................................................................. 299
layer_lstm_cell ........................................................... 303
layer_masking ............................................................. 305
layer_maximum ............................................................ 306
layer_max_pooling_1d .................................................... 307
layer_max_pooling_2d .................................................... 308
layer_max_pooling_3d .................................................... 310
layer_minimum ............................................................. 311
layer_multiply ............................................................. 312
layer_multi_head_attention ............................................. 312
layer_normalization ..................................................... 314
layer_permute ............................................................. 316
layer_random_brightness ............................................... 317
layer_random_contrast ................................................. 318
layer_random_crop ....................................................... 319
layer_random_flip ........................................................ 321
layer_random_height ..................................................... 322
layer_random_rotation .................................................. 323
layer_random_translation .............................................. 325
layer_random_width ..................................................... 326
layer_random_zoom ...................................................... 328
layer_repeat_vector ...................................................... 329
layer_rescaling ........................................................... 330
layer_reshape ............................................................. 331
layer_resizing ............................................................ 333
layer_rnn ................................................................. 334
layer_separable_conv_1d ................................................ 338
layer_separable_conv_2d ................................................ 341
layer_simple_rnn ........................................................ 344
layer_simple_rnn_cell ................................................... 347
layer Spatial_dropout_1d ................................................. 349
layer Spatial_dropout_2d ................................................. 350
layer Spatial_dropout_3d ................................................. 351
layer_stacked_rnn_cells ............................................... 352
layer_string_lookup ..................................................... 353
layer_subtract ........................................................... 355
layer_text_vectorization ............................................... 356
layer_unit_normalization ............................................... 359
layer_upsampling_1d ..................................................... 360
layer_upsampling_2d ..................................................... 361
layer_upsampling_3d ..................................................... 362
layer_zero_padding_1d .................................................. 364
layer_zero_padding_2d .................................................. 365
layer_zero_padding_3d .................................................. 366
learning_rate_schedule_cosine_decay .................................. 368
learning_rate_schedule_cosine_decay_restarts ......................... 369
learning_rate_schedule_exponential_decay ............................. 370
learning_rate_schedule_inverse_time_decay ........................... 372
learning_rate_schedule_piecewise_constant_decay ..................... 374
learning_rate_schedule_polynomial_decay ............................ 375
loss-functions .......................................................... 377
make_sampling_table ..................................................... 381
mark_active .................................................................. 382
Metric ....................................................................... 383
metric_accuracy ........................................................... 385
metric_auc .................................................................. 386
metric_binary_accuracy .................................................... 388
metric_binary_crossentropy ................................................. 390
metric_categorical_accuracy .............................................. 391
metric_categorical_crossentropy ........................................... 392
metric_categorical_hinge .................................................. 394
metric_cosine_similarity .................................................. 395
metric_false_negatives ..................................................... 396
metric_false_positives ...................................................... 397
metric_hinge ................................................................ 398
metric_kullback_leibler_divergence ...................................... 399
metric_logcosh_error ....................................................... 400
metric_mean ................................................................. 401
metric_mean_absolute_error ................................................. 402
metric_mean_absolute_percentage_error ................................. 403
metric_mean_iou ............................................................. 405
metric_mean_relative_error ............................................... 406
metric_mean_squared_error ............................................... 407
metric_mean_squared_logarithmic_error ................................. 408
metric_mean_tensor ........................................................ 409
metric_mean_wrapper ....................................................... 410
metric_poisson ............................................................... 411
metric_precision ............................................................ 412
metric_precision_at_recall ............................................... 414
metric_recall ................................................................. 415
metric_recall_at_precision .................................................. 416
metric_root_mean_squared_error ......................................... 418
metric_sensitivity_at_specificity .......................................... 419
metric_sparse_categorical_accuracy ..................................... 420
metric_sparse_categorical_crossentropy ................................. 421
metric_sparse_categorical_crossentropy .................................. 421
metric_sparse_top_k_categorical_accuracy .............................. 423
metric_specification_at_sensitivity ......................................... 424
metric_squared_hinge ...................................................... 425
metric_sum ................................................................. 426
topics documented:

- metric_top_k_categorical_accuracy ........................................... 427
- metric_true_negatives .............................................................. 428
- metric_true_positives .............................................................. 429
- model_from_saved_model ............................................................ 430
- model_to_json ........................................................................... 431
- model_to_yaml .......................................................................... 432
- new_learning_rate_schedule_class ............................................... 432
- normalize ................................................................................. 433
- optimizer_adadelta .................................................................... 434
- optimizer_adagrad ..................................................................... 435
- optimizer_adam ......................................................................... 436
- optimizer_adamax ...................................................................... 437
- optimizer_nadam ........................................................................ 438
- optimizer_rmsprop .................................................................... 439
- optimizer_sgd ........................................................................... 440
- pad_sequences ............................................................................ 441
- plot.keras.engine.training.Model .................................................. 442
- plot.keras_training_history .......................................................... 443
- pop_layer ..................................................................................... 444
- predict.keras.engine.training.Model .............................................. 445
- predict_on_batch ........................................................................ 446
- regularizer_11 ............................................................................. 447
- regularizer_orthogonal ............................................................... 447
- reset_states ............................................................................... 448
- save_model_hdf5 ........................................................................ 449
- save_model_tf ............................................................................ 450
- save_model_weights_hdf5 ............................................................. 451
- save_model_weights_tf ................................................................. 452
- save_text_tokenizer .................................................................... 453
- sequences_to_matrix ................................................................... 454
- sequential_model_input_layer ...................................................... 455
- serialize_model ......................................................................... 456
- skipgrams .................................................................................... 457
- summary.keras.engine.training.Model ......................................... 458
- texts_to_matrix .......................................................................... 459
- texts_to_sequences ...................................................................... 460
- texts_to_sequences_generator ....................................................... 460
- text_dataset_from_directory ....................................................... 461
- text_hashing_trick ....................................................................... 463
- text_one_hot ............................................................................... 464
- text_tokenizer ............................................................................ 465
- text_to_word_sequence ............................................................... 466
- timeseries_dataset_from_array .................................................... 467
- timeseries_generator .................................................................. 470
- time_distributed ......................................................................... 471
- to_categorical ............................................................................. 472
- train_on_batch ............................................................................. 473
- use_implementation ...................................................................... 474
keras-package

R interface to Keras

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, \( \text{sigmoid}(x) = 1 / (1 + \exp(-x)) \).
- `softmax(...)`: Softmax converts a vector of values to a probability distribution.
- `softplus(...)`: Softplus activation function, \( \text{softplus}(x) = \log(\exp(x) + 1) \).
- `softsign(...)`: Softsign activation function, \( \text{softsign}(x) = x / (\text{abs}(x) + 1) \).
- `tanh(...)`: Hyperbolic tangent activation function.
- `exponential(...)`: Exponential activation function.
- `gelu(...)`: Applies the Gaussian error linear unit (GELU) activation function.
- `swish(...)`: Swish activation function, \( \text{swish}(x) = x * \text{sigmoid}(x) \).

### 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_relu

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.

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

Usage

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

Arguments

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

Details

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

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

Keras model example with multiple adapts:
```r
layer <- layer_normalization(axis=NULL)
adapt(layer, c(0, 2))
model <- keras_model_sequential(layer)
predict(model, c(0, 1, 2)) # [1] -1 0 1

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

tf.data.Dataset example with multiple adapts:

layer <- layer_normalization(axis=NULL)
adapt(layer, c(0, 2))
input_ds <- tfdatasets::range_dataset(0, 3)
normalized_ds <- input_ds %>%
  tfdatasets::dataset_map(layer)
str(reticulate::iterate(normalized_ds))
# List of 3
# $ :tf.Tensor([-1.], shape=(1,), dtype=float32)
# $ :tf.Tensor([0.], shape=(1,), dtype=float32)
# $ :tf.Tensor([1.], shape=(1,), dtype=float32)
adapt(layer, c(-1, 1))
normalized_ds <- input_ds %>%
  tfdatasets::dataset_map(layer) # Re-map over the input dataset.
str(reticulate::iterate(normalized_ds$as_numpy_iterator()))
# List of 3
# $ : num [1(1d)] -1
# $ : num [1(1d)] 0
# $ : num [1(1d)] 1

See Also

* [https://www.tensorflow.org/guide/keras/preprocessing_layers#the_adapt_method](https://www.tensorflow.org/guide/keras/preprocessing_layers#the_adapt_method)
* [https://keras.io/guides/preprocessing_layers/#the-adapt-method](https://keras.io/guides/preprocessing_layers/#the-adapt-method)
```

**application_densenet**  
Instantiates the DenseNet architecture.

**Description**

Instantiates the DenseNet architecture.
Usage

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

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

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

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

```
densenet_preprocess_input(x, data_format = NULL)
```

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_efficientnet**

*Instantiates the EfficientNetB0 architecture*

---

**Description**

Instantiates the EfficientNetB0 architecture

**Usage**

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

```r
application_efficientnet_b1(
  include_top = TRUE,
  weights = "imagenet",
  ...
)
```
application_efficientnet

    input_tensor = NULL,
    input_shape = NULL,
    pooling = NULL,
    classes = 1000L,
    classifier_activation = "softmax",
    ...
)

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

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

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

application_efficientnet_b5(
    include_top = TRUE,
    weights = "imagenet",
    input_tensor = NULL,
    input_shape = NULL,
    pooling = NULL,
    classes = 1000L,
Arguments

include_top  Whether to include the fully-connected layer at the top of the network. Defaults to TRUE.

weights  One of NULL (random initialization), 'imagenet' (pre-training on ImageNet), or the path to the weights file to be loaded. Defaults to 'imagenet'.

input_tensor  Optional Keras tensor (i.e. output of layer_input()) to use as image input for the model.

input_shape  Optional shape list, only to be specified if include_top is FALSE. It should have exactly 3 inputs channels.

pooling  Optional pooling mode for feature extraction when include_top is FALSE. Defaults to NULL.

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

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

... For backwards and forwards compatibility

Details
Reference:

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

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

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

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

EfficientNet models expect their inputs to be float tensors of pixels with values in the [0-255] range.

Note
Each Keras Application typically expects a specific kind of input preprocessing. For EfficientNet, input preprocessing is included as part of the model (as a Rescaling layer), and thus a calling a preprocessing function is not necessary.

See Also

- https://www.tensorflow.org/api_docs/python/tf/keras/applications/efficientnet/EfficientNetB0
- https://keras.io/api/applications/

---

application_inception_resnet_v2

Inception-ResNet v2 model, with weights trained on ImageNet

Description

Inception-ResNet v2 model, with weights trained on ImageNet
**Usage**

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

inception_resnet_v2_preprocess_input(x)
```

**Arguments**

- `include_top` (optional): Whether to include the fully-connected layer at the top of the network. Defaults to `TRUE`.
- `weights` (optional): One of `NULL` (random initialization), 'imagenet' (pre-training on ImageNet), or the path to the weights file to be loaded. Defaults to 'imagenet'.
- `input_tensor` (optional): 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 `(299, 299, 3)`). It should have exactly 3 inputs channels, and width and height should be no smaller than 71. E.g. `(150, 150, 3)` would be one valid value.
- `pooling` (optional): Optional pooling mode for feature extraction when `include_top` is `FALSE`. Defaults to `NULL`:
  - `NULL` means that the output of the model will be the 4D tensor output of the last convolutional layer.
  - `'avg'` means that global average pooling will be applied to the output of the last convolutional layer, and thus the output of the model will be a 2D tensor.
  - `'max'` means that global max pooling will be applied.
- `classes` (optional): Optional number of classes to classify images into, only to be specified if `include_top` is `TRUE`, and if no `weights` argument is specified. Defaults to 1000 (number of ImageNet classes).
- `classifier_activation` (optional): A string or callable. The activation function to use on the "top" layer. Ignored unless `include_top` is `TRUE`. Set `classifier_activation = NULL` to return the logits of the "top" layer. Defaults to 'softmax'. When loading pretrained weights, `classifier_activation` can only be `NULL` or "softmax".
- `x`: `preprocess_input()` takes an array or floating point tensor, 3D or 4D with 3 color channels, with values in the range [0, 255].
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


```python
application_inception_v3

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

Description

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

Usage

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

inception_v3_preprocess_input(x)
```

Arguments

- `include_top` (Default: TRUE) Whether to include the fully-connected layer at the top of the network.
- `weights` (Default: 'imagenet') One of NULL (random initialization), 'imagenet' (pre-training on ImageNet), or the path to the weights file to be loaded.
- `input_tensor` (Default: NULL) 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 (299, 299, 3). It should have exactly 3 inputs channels, and width and height should be no smaller than 71. E.g. (150, 150, 3) would be one valid value.

pooling
Optional pooling mode for feature extraction when include_top is FALSE. Defaults to NULL.
- NULL means that the output of the model will be the 4D tensor output of the last convolutional layer.
- 'avg' means that global average pooling will be applied to the output of the last convolutional layer, and thus the output of the model will be a 2D tensor.
- 'max' means that global max pooling will be applied.

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

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

... For backwards and forwards compatibility

x
preprocess_input() takes an array or floating point tensor, 3D or 4D with 3 color channels, with values in the range [0, 255].

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

application_mobilenet  MobileNet model architecture.
Usage

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

```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.

classifier_activation

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

... For backwards and forwards compatibility

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.

---

**application_mobilenet_v2**

*MobileNetV2 model architecture*

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,
classifier_activation = "softmax",
...)

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.

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

**x**
input tensor, 4D

**preds**
Tensor encoding a batch of predictions.

**top**
integer, how many top-guesses to return.

**filepath**
File path
application_mobilenet_v3

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_mobilenet_v3

Instantiates the MobileNetV3Large architecture

Description

Instantiates the MobileNetV3Large architecture

Usage

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

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

Arguments

input_shape Optional shape vector, to be specified if you would like to use a model with an input image resolution that is not c(224, 224, 3). It should have exactly 3 inputs channels c(224, 224, 3). You can also omit this option if you would like to infer input_shape from an input_tensor. If you choose to include both input_tensor and input_shape then input_shape will be used if they match, if the shapes do not match then we will throw an error. E.g. c(160, 160, 3) would be one valid value.

alpha controls the width of the network. This is known as the depth multiplier in the MobileNetV3 paper, but the name is kept for consistency with MobileNetV1 in Keras.

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

minimalistic In addition to large and small models this module also contains so-called minimalistic models, these models have the same per-layer dimensions characteristic as MobileNetV3 however, they don’t utilize any of the advanced blocks (squeeze-and-excite units, hard-swish, and 5x5 convolutions). While these models are less efficient on CPU, they are much more performant on GPU/DSP.

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

weights String, one of NULL (random initialization), 'imagenet' (pre-training on ImageNet), or the path to the weights file to be loaded.

input_tensor Optional Keras tensor (i.e. output of layer_input()) to use as image input for the model.

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

pooling String, optional pooling mode for feature extraction when include_top is FALSE.

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

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

classifier_activation A string or callable. The activation function to use on the "top" layer. Ignored unless include_top = TRUE. Set classifier_activation = NULL to return the
application_mobilenet_v3

logits of the "top" layer. When loading pretrained weights, classifier_activation can only be NULL or "softmax".

include_preprocessing

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

Details

Reference:

• Searching for MobileNetV3 (ICCV 2019)

The following table describes the performance of MobileNets v3::

MACs stands for Multiply Adds

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

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

Value

A keras Model instance

Note

Each Keras application typically expects a specific kind of input preprocessing. For ModelNetV3, by default input preprocessing is included as a part of the model (as a Rescaling layer), and thus a preprocessing function is not necessary. In this use case, ModelNetV3 models expect their inputs to be float tensors of pixels with values in the [0~255] range. At the same time, preprocessing as a part of the model (i.e. Rescaling layer) can be disabled by setting include_preprocessing argument to FALSE. With preprocessing disabled ModelNetV3 models expect their inputs to be float tensors of pixels with values in the [-1, 1] range.

See Also

• https://www.tensorflow.org/api_docs/python/tf/keras/applications/MobileNetV3Large
• https://www.tensorflow.org/api_docs/python/tf/keras/applications/MobileNetV3Small
• https://keras.io/api/applications/
application_nasnet  Instatiates 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
)
```

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

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

```r
nasnet_preprocess_input(x)
```
**Arguments**

**input_shape**  Optional shape list, the input shape is by default (331, 331, 3) for NASNet-Large 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].

**application_resnet**  Instantiates the ResNet architecture

**Description**

Instantiates the ResNet architecture
Usage

```r
apPLICATION_RESNET

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

application_resnet101(
    include_top = TRUE,
    weights = "imagenet",
    input_tensor = NULL,
    input_shape = NULL,
    pooling = NULL,
    classes = 1000,
    ...
)

application_resnet152(
    include_top = TRUE,
    weights = "imagenet",
    input_tensor = NULL,
    input_shape = NULL,
    pooling = NULL,
    classes = 1000,
    ...
)

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

application_resnet101_v2(
    include_top = TRUE,
    weights = "imagenet",
    input_tensor = NULL,
    input_shape = NULL,
    pooling = NULL,
    ...
)
```
```r
application_resnet

classes = 1000,
classifier_activation = "softmax",
...
)

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

classification_preprocess_input(x)

classification_v2_preprocess_input(x)

Arguments

include_top Whether to include the fully-connected layer at the top of the network. Defaults to TRUE.
weights One of NULL (random initialization), 'imagenet' (pre-training on ImageNet), or the path to the weights file to be loaded. Defaults to 'imagenet'.
input_tensor Optional Keras tensor (i.e. output of 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 c(224, 224, 3) (with 'channels_last' data format) or c(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. c(200, 200, 3) would be one valid value.
pooling Optional pooling mode for feature extraction when include_top is FALSE. Defaults to NULL.

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

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

For backwards and forwards compatibility
classifier_activation

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

x

preprocess_input() takes an array or floating point tensor, 3D or 4D with 3 color channels, with values in the range [0, 255].

Details

Reference:

- Deep Residual Learning for Image Recognition (CVPR 2015)

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

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

Note: each Keras Application expects a specific kind of input preprocessing. For ResNet, call tf.keras.applications.resnet.preprocess_input on your inputs before passing them to the model. resnet.preprocess_input will convert the input images from RGB to BGR, then will zero-center each color channel with respect to the ImageNet dataset, without scaling.

See Also

- https://www.tensorflow.org/api_docs/python/tf/keras/applications/resnet50/ResNet50
- https://www.tensorflow.org/api_docs/python/tf/keras/applications/resnet/ResNet101
- https://www.tensorflow.org/api_docs/python/tf/keras/applications/resnet/ResNet152
- https://www.tensorflow.org/api_docs/python/tf/keras/applications/resnet_v2/ResNet50V2
- https://www.tensorflow.org/api_docs/python/tf/keras/applications/resnet_v2/ResNet101V2
- https://www.tensorflow.org/api_docs/python/tf/keras/applications/resnet_v2/ResNet152V2
- https://keras.io/api/applications/

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,
```
## Application_VGG

```r
# 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)
```

---

**Description**

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,
    classifier_activation = "softmax"
)
```

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

**Arguments**

- **include_top**
  - whether to include the 3 fully-connected layers at the top of the network.
- **weights**
  - One of `NULL` (random initialization), `imagenet` (pre-training on ImageNet), or the path to the weights file to be loaded. Defaults to `'imagenet'`.
- **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)) 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. Defaults to NULL.
  - NULL means that the output of the model will be the 4D tensor output of the last convolutional layer.
  - 'avg' means that global average pooling will be applied to the output of the last convolutional layer, and thus the output of the model will be a 2D tensor.
  - 'max' means that global max pooling will be applied.

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

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

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)
```
application_xception

## End(Not run)

---

application_xception  Instantiates the Xception architecture

### Description

Instantiates the Xception architecture

### Usage

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

```r
xception_preprocess_input(x)
```

### Arguments

- **include_top**: Whether to include the fully-connected layer at the top of the network. Defaults to `TRUE`.
- **weights**: One of `NULL` (random initialization), `'imagenet'` (pre-training on ImageNet), or the path to the weights file to be loaded. Defaults to `'imagenet'`.
- **input_tensor**: Optional Keras tensor (i.e. output of `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 `(299, 299, 3)`). It should have exactly 3 inputs channels, and width and height should be no smaller than 71. E.g. `(150, 150, 3)` would be one valid value.
- **pooling**: Optional pooling mode for feature extraction when `include_top` is `FALSE`. Defaults to `NULL`.
  - `NULL` means that the output of the model will be the 4D tensor output of the last convolutional layer.
  - `'avg'` means that global average pooling will be applied to the output of the last convolutional layer, and thus the output of the model will be a 2D tensor.
  - `'max'` means that global max pooling will be applied.
classes
   Optional number of classes to classify images into, only to be specified if include_top is TRUE, and if no weights argument is specified. Defaults to 1000 (number of ImageNet classes).

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

...  For backwards and forwards compatibility
x    preprocess_input() takes an array or floating point tensor, 3D or 4D with 3 color channels, with values in the range [0, 255].

Details
   For image classification use cases, see this page for detailed examples.
   For transfer learning use cases, make sure to read the guide to transfer learning & fine-tuning.
   The default input image size for this model is 299x299.

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

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

See Also
   • https://www.tensorflow.org/api_docs/python/tf/keras/applications/xception/Xception
   • https://keras.io/api/applications/

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/] for additional details on the available functions.

---

| bidirectional | Bidirectional wrapper for RNNs |

**Description**

Bidirectional wrapper for RNNs

**Usage**

```r
bidirectional(
  object,
  layer,
  merge_mode = "concat",
  weights = NULL,
  backward_layer = NULL,
  ...
)
```

**Arguments**

- **object** What to compose the new Layer instance with. Typically a Sequential model or a Tensor (e.g., as returned by `layer_input()`). The return value depends on `object`. If `object` is:
  - missing or NULL, the Layer instance is returned.
  - a Sequential model, the model with an additional layer is returned.
  - a Tensor, the output tensor from `layer_instance(object)` is returned.
- **layer** A RNN layer instance, such as `layer_lstm()` or `layer_gru()`. It could also be a `keras$layers$Layer` instance that meets the following criteria:
  1. Be a sequence-processing layer (accepts 3D+ inputs).
  2. Have a `go_backwards`, `return_sequences` and `return_state` attribute (with the same semantics as for the RNN class).
3. Have an input_spec attribute.
4. Implement serialization via get_config() and from_config(). Note that the recommended way to create new RNN layers is to write a custom RNN cell and use it with layer_rnn(), instead of subclassing keras$layers$Layer directly.
5. When returns_sequences = TRUE, the output of the masked timestep will be zero regardless of the layer's original zero_output_for_mask value.

merge_mode Mode by which outputs of the forward and backward RNNs will be combined. One of 'sum', 'mul', 'concat', 'ave', NULL. If NULL, the outputs will not be combined, they will be returned as a list. Default value is 'concat'.

weights Split and propagated to the initial_weights attribute on the forward and backward layer.

backward_layer Optional keras.layers.RNN, or keras.layers.Layer instance to be used to handle backwards input processing. If backward_layer is not provided, the layer instance passed as the layer argument will be used to generate the backward layer automatically. Note that the provided backward_layer should have properties matching those of the layer argument, in particular it should have the same values for stateful, return_states, return_sequences, etc. In addition, backward_layer and layer should have different go_backwards argument values. A ValueError will be raised if these requirements are not met.

... standard layer arguments.

See Also

- https://www.tensorflow.org/api_docs/python/tf/keras/layers/Bidirectional
- https://keras.io/api/layers/recurrent_layers/bidirectional/

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

callback_csv_logger

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,
See Also

Other callbacks: `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()`, `callback_terminate_on_naan()`

---

`callback_early_stopping`

*Stop training when a monitored quantity has stopped improving.*

Description

Stop training when a monitored quantity has stopped improving.

Usage

```r
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

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

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()

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

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.
callback_learning_rate_scheduler

Description

Learning rate scheduler.

Usage

callback_learning_rate_scheduler(schedule)

Arguments

schedule a function that takes an epoch index as input (integer, indexed from 0) and current learning rate and returns a new learning rate as output (float).
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 file-name.

Usage

```r
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.
save_freq  

'epoch' or integer. When using 'epoch', the callback saves the model after each epoch. When using integer, the callback saves the model at end of a batch at which this many samples have been seen since last saving. Note that if the saving isn't aligned to epochs, the monitored metric may potentially be less reliable (it could reflect as little as 1 batch, since the metrics get reset every epoch). Defaults to 'epoch'

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.

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_progbar_logger

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 seen 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_model_checkpoint(), 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

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
  - 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_progbar_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

```r
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

```python
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, clone_function = NULL)
```

**Arguments**

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

```r
function(layer) layer$\_\_class\_\_\$from_config(layer$\_\_get_config())
```

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

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,
```
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()
Usage

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())
}
```

```r
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**

```r
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**

```r
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

Note

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

---

**create_layer_wrapper** | Create a Keras Layer wrapper

Description

Create a Keras Layer wrapper

Usage

```r
create_layer_wrapper(Layer, modifiers = NULL, convert = TRUE)
```

Arguments

- **Layer**: 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 built-in 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 (%>%).

The R function will arguments taken from the `initialize` (or `__init__`) method of the Layer.

If `Layer` is an R6 object, this will delay initializing the python session, so it is safe to use in an R package.
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](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(),
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
```r
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)`.
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)`.
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()`
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

```r
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.
**dataset_mnist**

**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

Description

Dataset of 11,228 newswires from Reuters, labeled over 46 topics. As with dataset_imdb(), each wire is encoded as a sequence of word indexes (same conventions).

Usage

dataset_reuters(
    path = "reuters.npz",
    num_words = NULL,
    skip_top = 0L,
    maxlen = NULL,
    test_split = 0.2,
    seed = 113L,
    start_char = 1L,
    oov_char = 2L,
    index_from = 3L
)

dataset_reuters_word_index(path = "reuters_word_index.pkl")

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>path</td>
<td>Where to cache the data (relative to ~/.keras/dataset).</td>
</tr>
<tr>
<td>num_words</td>
<td>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.</td>
</tr>
<tr>
<td>skip_top</td>
<td>Skip the top N most frequently occurring words (which may not be informative).</td>
</tr>
<tr>
<td>maxlen</td>
<td>Truncate sequences after this length.</td>
</tr>
<tr>
<td>test_split</td>
<td>Fraction of the dataset to be used as test data.</td>
</tr>
<tr>
<td>seed</td>
<td>Random seed for sample shuffling.</td>
</tr>
<tr>
<td>start_char</td>
<td>The start of a sequence will be marked with this character. Set to 1 because 0 is usually the padding character.</td>
</tr>
<tr>
<td>oov_char</td>
<td>Words that were cut out because of the num_words or skip_top limit will be replaced with this character.</td>
</tr>
<tr>
<td>index_from</td>
<td>Index actual words with this index and higher.</td>
</tr>
</tbody>
</table>

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.
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 = "auto",
  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()`.

See Also

Other datasets: `dataset_boston_housing()`, `dataset_cifar100()`, `dataset_cifar10()`, `dataset_fashion_mnist()`, `dataset_imdb()`, `dataset_mnist()`
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()`, `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()`

---

**export_savedmodel.keras.engine.training.Model**

*Export a Saved Model*

**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 = "auto"),
  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.
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).

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).

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

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.

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

List of callbacks to be called during training.

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.

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.

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: 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.

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.

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()`.

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.

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.train_on_batch()`, `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**

Required for featurewise_center, featurewise_std_normalization and zca_whitening.

**Usage**

`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().
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

```r
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.
directory character, path to the directory to read images from. If NULL, data in x_col column should be absolute paths.

x_col character, column in dataframe that contains the filenames (or absolute paths if directory is NULL).

y_col string or list, column/s in dataframe that has the target data.

generator Image data generator to use for augmenting/normalizing image data.

target_size Either NULL (default to original size) or integer vector (img_height, img_width).

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 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.

class_mode one of "categorical", "binary", "sparse", "input", "other" or None. Default: "categorical". Mode for yielding the targets:
   • "binary": 1D array of binary labels,
   • "categorical": 2D array of one-hot encoded labels. Supports multi-label output.
   • "sparse": 1D array of integer labels,
   • "input": images identical to input images (mainly used to work with autoencoders),
   • "other": array of y_col data,
   • "multi_output": 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()).

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".

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 supported. By default, "nearest" is used.

drop_duplicates (deprecated in TF >= 2.3) Boolean, whether to drop duplicate rows based on filename. The default value is TRUE.
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 an array of corresponding labels. The generator loops indefinitely.

Note

This functions 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()
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". Determines 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 labels. If NULL, no labels are returned (the generator will only yield batches of image 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 supported. By default, "nearest" is used.

Details

Yields batches indefinitely, in an infinite loop.
freeze_weights

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()`

---

freeze_weights  
*Freeze and unfreeze weights*

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

**Usage**

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

**Arguments**

- **object**  
  Keras model or layer object

- **from**  
  Layer instance, layer name, or layer index within model

- **to**  
  Layer instance, layer name, or layer index within model

- **which**  
  layer names, integer positions, layers, logical vector (of length(object$layers)), or a function returning a logical vector.

**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:
conv_base <- application_vgg16(
  weights = "imagenet",
  include_top = FALSE,
  input_shape = c(150, 150, 3)
)
```
# freeze it's weights
freeze_weights(conv_base)

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")
)

conv_base

# 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")
)

conv_base

# freeze only the last 5 layers
freeze_weights(conv_base, from = -5)

conv_base

# equivalently, also freeze only the last 5 layers
unfreeze_weights(conv_base, to = -6)

conv_base

# Freeze only layers of a certain type, e.g, BatchNorm layers
batch_norm_layer_class_name <- class(layer_batch_normalization())[1]
is_batch_norm_layer <- function(x) inherits(x, batch_norm_layer_class_name)

model <- application_efficientnet_b0()
freeze_weights(model, which = is_batch_norm_layer)

model

# equivalent to:
for(layer in model$layers) {

if(is_batch_norm_layer(layer))
  layer$trainable <- FALSE
else
  layer$trainable <- TRUE
}

## End(Not run)

---

### generator_next

*Retrieve 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

```r
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).

---

### get_config

*Layer/Model configuration*

#### Description

A layer config is an object returned from `get_config()` that contains the configuration of a layer or model. The same layer or model can be reinstantiated later (without its trained weights) from this configuration using `from_config()`. The config does not include connectivity information, nor the class name (those are handled externally).

#### Usage

```r
get_config(object)
from_config(config)
```

#### Arguments

- `object`: Layer or model object
- `config`: Object with layer or model configuration
get_file

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() function (for model configuration only, not weights) or the save_model_tfn() function to save the model configuration and weights to the filesystem.

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_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()

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

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)
get_layer

Arguments

<table>
<thead>
<tr>
<th>object</th>
<th>Layer or model object</th>
</tr>
</thead>
<tbody>
<tr>
<td>node_index</td>
<td>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.</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>object</th>
<th>Keras model object</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>String, name of layer.</td>
</tr>
<tr>
<td>index</td>
<td>Integer, index of layer (1-based). Also valid are negative values, which count from the end of model.</td>
</tr>
</tbody>
</table>

Value

A layer instance.

See Also

**get_weights**

*Layer/Model weights as R arrays*

**Description**

Layer/Model weights as R arrays

**Usage**

```r
get_weights(object, trainable = NA)
```

```r
set_weights(object, weights)
```

**Arguments**

- `object`: Layer or model object
- `trainable`: if `NA` (the default), all weights are returned. If `TRUE`,
- `weights`: Weights as R array

**Note**

You can access the Layer/Model as `tf.Tensors` or `tf.Variables` at `object$weights`, `object$trainable_weights`, or `object$non_trainable_weights`

**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**

```r
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**

Decode 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.

Usage
image_dataset_from_directory(directory, labels = "inferred", label_mode = "int", class_names = NULL, color_mode = "rgb", batch_size = 32,
image_dataset_from_directory

image_size = c(256, 256),
shuffle = TRUE,
seed = NULL,
validation_split = NULL,
subset = NULL,
interpolation = "bilinear",
follow_links = FALSE,
crop_to_aspect_ratio = FALSE,
...
)

Arguments

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

labels Either "inferred" (labels are generated from the directory structure), 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 Valid values:
  • 'int': labels are encoded as integers (e.g. for sparse_categorical_crossentropy loss).
  • 'categorical': labels are encoded as a categorical vector (e.g. for categorical_crossentropy loss).
  • 'binary': labels (there can be only 2) are encoded as float32 scalars with values 0 or 1 (e.g. for binary_crossentropy).
  • NULL: (no labels).

class_names Only valid if "labels" is "inferred". This is the explicit list of class names (must match names of subdirectories). Used to control the order of the classes (otherwise alphanumerical order is used).

color_mode One of "grayscale", "rgb", "rgba". 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 alphanumeric order.

seed Optional random seed for shuffling and transformations.

validation_split Optional float between 0 and 1, fraction of data to reserve for validation.

subset 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.
crop_to_aspect_ratio
If TRUE, resize the images without aspect ratio distortion. When the original aspect ratio differs from the target aspect ratio, the output image will be cropped so as to return the largest possible window in the image (of size image_size) that matches the target aspect ratio. By default (crop_to_aspect_ratio=False), aspect ratio may not be preserved.

... Legacy arguments

Details
If your directory structure is:

main_directory/
  ...class_a/
      ....a_image_1.jpg
      ....a_image_2.jpg
  ...class_b/
      ....b_image_1.jpg
      ....b_image_2.jpg

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

Supported image formats: jpeg, png, bmp, gif. Animated gifs are truncated to the first frame.

Value
A tf.data.Dataset object. If label_mode is NULL, it yields float32 tensors of shape (batch_size, image_size[1], image_size[2], num_channels), encoding images (see below for rules regarding num_channels).

Otherwise, it yields pairs of (images, labels), where images has shape (batch_size, image_size[1], image_size[2], num_channels), and labels follows the format described below.

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

Rules regarding number of channels in the yielded images:
- if color_mode is grayscale, there’s 1 channel in the image tensors.
- if color_mode is rgb, there are 3 channel in the image tensors.
- if color_mode is rgba, there are 4 channel in the image tensors.

See Also
https://www.tensorflow.org/api_docs/python/tf/keras/utils/image_dataset_from_directory
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": aaaaaaaa|abcd|dddddddd
  • "reflect": abcddcba|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

Loads an image into PIL format.

Description

Loads an image into PIL format.

Usage

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

Arguments

path Path to image file
grayscale DEPRECATED use color_mode="grayscale"
color_mode One of "grayscale", "rgb", "rgba". Default: "rgb". The desired image format.
target_size Either NULL (default to original size) or integer vector (img_height, img_width).
interpolation Interpolation method used to resample the image if the target size is different from that of the loaded image. Supported methods are "nearest", "bilinear", and "bicubic". If PIL version 1.1.3 or newer is installed, "lanczos" is also supported. If PIL version 3.4.0 or newer is installed, "box" and "hamming" are also supported. By default, "nearest" is used.

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

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

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
width Width to resize to
path Path to save image to
file_format Optional file format override. If omitted, the format to use is determined from the filename extension. If a file object was used instead of a filename, this parameter should always be used.
scale Whether to rescale image values to be within 0,255

See Also
Other image preprocessing: fit_image_data_generator(), flow_images_from_dataframe(), flow_images_from_data(), flow_images_from_directory(), image_load()
**implementation**

<table>
<thead>
<tr>
<th>implementation</th>
<th>Keras implementation</th>
</tr>
</thead>
</table>

**Description**

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

**Usage**

```python
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**  
*Initializer that generates tensors initialized to a constant value.*

**Description**

Initializer that generates tensors initialized 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 stddev = \(\sqrt{2 / (\text{fan}_\text{in} + \text{fan}_\text{out})}\) 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**

```r
initializer_glorot_normal(seed = NULL)
```

**Arguments**

- `seed`: Integer used to seed the random generator.

**References**


**See Also**

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

---

**initializer_glorot_uniform**

Glorot uniform initializer, also called Xavier uniform initializer.

**Description**

It draws samples from a uniform distribution within \(-\text{limit}, \text{limit}\) where limit is \(\sqrt{6 / (\text{fan}_\text{in} + \text{fan}_\text{out})}\) 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**

```r
initializer_glorot_uniform(seed = NULL)
```

**Arguments**

- `seed`: Integer used to seed the random generator.
**initializer_he_normal**

He normal initializer.

### Description

It draws samples from a truncated normal distribution centered on 0 with $	ext{stddev} = \sqrt{2 / \text{fan\_in}}$ where \text{fan\_in} is the number of input units in the weight tensor.

### Usage

```r
initializer_he_normal(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_uniform()`, `initializer_identity()`, `initializer_lecun_normal()`, `initializer_lecun_uniform()`, `initializer_ones()`, `initializer_orthogonal()`, `initializer_random_normal()`, `initializer_random_uniform()`, `initializer_truncated_normal()`, `initializer_variance_scaling()`, `initializer_zeros()`
initializer_he_uniform

He uniform variance scaling initializer.

Description

It draws samples from a uniform distribution within `-limit, limit` where `limit` is `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

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 \( \text{stddev} <- \sqrt{\frac{1}{\text{fan\_in}}} \) where \( \text{fan\_in} \) is the number of input units in the weight tensor.

Usage

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 $\text{limit}$ is $\sqrt{3 / \text{fan}_{\text{in}}}$ where $\text{fan}_{\text{in}}$ is the number of input units in the weight tensor.

**Usage**

```r
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_lecun_uniform()`, `initializer_ones()`, `initializer_orthogonal()`, `initializer_random_normal()`, `initializer_random_uniform()`, `initializer_truncated_normal()`, `initializer_variance_scaling()`, `initializer_zeros()`

initializer_ones

**Description**

Initializer that generates tensors initialized to 1.

**Usage**

```r
initializer_ones()
```

**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_orthogonal

Initializer that generates a random orthogonal matrix.

**Description**

Initializer that generates a random orthogonal matrix.

**Usage**

```r
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_ones()`, `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**

```r
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.
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

initializer_truncated_normal(mean = 0, stddev = 0.05, seed = NULL)
**initializer_variance_scaling**

**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_normal()`, `initializer_random_uniform()`, `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(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**

```r
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 backward 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)`.
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 "2.9".

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

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()
is_keras_available

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 and conda_binary() for more details.

version TensorFlow version to install. Valid values include:

- "default" installs 2.9
- "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.

pip_ignore_installed Whether pip should ignore installed python packages and reinstall all already installed python packages. This defaults to TRUE, to ensure that TensorFlow dependencies like NumPy are compatible with the prebuilt TensorFlow binaries.

Details

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

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

## 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 0.

Value

the keras Python module
### keras_array

**Keras array object**

<table>
<thead>
<tr>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>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.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>keras_array(x, dtype = NULL)</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>x</strong> Object or list of objects to convert</td>
</tr>
<tr>
<td><strong>dtype</strong> 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.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>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 (&quot;C&quot; as opposed to &quot;Fortran&quot; ordering, which is the default for R arrays).</td>
</tr>
<tr>
<td>If the passed array is already a NumPy array with the desired dtype and &quot;C&quot; order then it is returned unmodified (no additional copies are made).</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>NumPy array with the specified dtype (or list of NumPy arrays if a list was passed for x).</td>
</tr>
</tbody>
</table>

### keras_model

**Keras Model**

<table>
<thead>
<tr>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A model is a directed acyclic graph of layers.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>keras_model(inputs, outputs = NULL, ...)</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>inputs</strong> Input layer</td>
</tr>
<tr>
<td><strong>outputs</strong> Output layer</td>
</tr>
<tr>
<td><strong>...</strong> Any additional arguments</td>
</tr>
</tbody>
</table>
keras_model_sequential

Keras Model composed of a linear stack of layers

Description

Keras Model composed of a linear stack of layers

Usage

keras_model_sequential(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 dimen-
   sions. 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.
in
   put_layer_name, 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
Other model functions: compile.keras.engine.training.Model(), evaluate.keras.engine.training.Model(),
evaluate_generator(), fit.keras.engine.training.Model(), fit_generator(), get_config(),
g

 et_layer(), 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()

Examples
## 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**

```r
k.abs(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_all**  
*Bitwise reduction (logical AND).*

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

**Usage**  
\[
k_{\text{all}}(x, \text{axis} = \text{NULL}, \text{keepdims} = \text{FALSE})
\]

**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](https://keras.rstudio.com/articles/backend.html#backend-functions).

---

**k_any**  
*Bitwise reduction (logical OR).*

**Description**  
Bitwise reduction (logical OR).

**Usage**  
\[
k_{\text{any}}(x, \text{axis} = \text{NULL}, \text{keepdims} = \text{FALSE})
\]

**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.
**k_arange**

**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**

```r
k_arange(start, stop = NULL, step = 1, dtype = "int32")
```

**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

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.

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.
k_batch_dot

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.

<table>
<thead>
<tr>
<th>k_backend</th>
<th>Active Keras backend</th>
</tr>
</thead>
</table>

Description

Active Keras backend

Usage

k_backend()

Value

The name of the backend Keras is currently using.

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.

<table>
<thead>
<tr>
<th>k_batch_dot</th>
<th>Batchwise dot product</th>
</tr>
</thead>
</table>

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)
**k_batch_flatten**

Arguments

- **x**: Keras tensor or variable with 2 or 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](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**

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

**Description**  
i.e. returns \( \text{output} \leftarrow \frac{(x - \text{mean})}{\sqrt{\text{var}} + \epsilon} \cdot \gamma + \beta \)

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

### 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).

---

**Description**

Sets the values of many tensor variables at once.

**Usage**

```
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**

```
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.

---

**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**

```
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.

---

**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

```python
k_cast(x, dtype)
```

Arguments

- **x**
  - Keras tensor (or variable).
- **dtype**
  - String, either ('float16', 'float32', or 'float64').

Value
Keras 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.
k_cast_to_floatx  

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

**Description**

Cast an array to the default Keras float type.

**Usage**

```r
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.
**k_clear_session**

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_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.

---

**k_clip**  
*Element-wise value clipping.*

---

**Description**
Element-wise value clipping.

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

**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.

<table>
<thead>
<tr>
<th>k_concatenate</th>
<th>Concatenates a list of tensors alongside the specified axis.</th>
</tr>
</thead>
</table>

Description
Concatenates a list of tensors alongside the specified axis.

Usage
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.
$k_{\text{constant}}$

$Create a constant tensor.$

**Description**

Creates a constant tensor.

**Usage**

$k_{\text{constant}}(value, dtype = \text{NULL}, shape = \text{NULL}, name = \text{NULL})$

**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.

**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](https://keras.rstudio.com/articles/backend.html#backend-functions).

---

$k_{\text{conv1d}}$

$1D convolution.$

**Description**

1D convolution.

**Usage**

$k_{\text{conv1d}}(x, kernel, strides = 1, padding = \text{"valid"}, data\_format = \text{NULL}, dilation\_rate = 1)$
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.

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`: Stride.
- `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.
**k_conv2d_transpose**

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

**Description**

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

**Usage**

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

**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.

---

**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.
**k_ctc_batch_cost**

**Runs CTC loss algorithm on each batch element.**

**Description**

Runs CTC loss algorithm on each batch element.

**Usage**

```python
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](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.
**k_ctc_label_dense_to_sparse**

*Converts CTC labels from dense to sparse.*

**Description**

Converts CTC labels from dense to sparse.

**Usage**

```python
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**

```python
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).
**k_cumsum**

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

**Description**

When attempting to multiply a nD tensor with a nD tensor, it reproduces the Theano behavior. (e.g. $(2, 3) \times (4, 3, 5) \rightarrow (2, 4, 5)$)

**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

**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
- `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).

---

<table>
<thead>
<tr>
<th>k_dtype</th>
<th>Returns the dtype of a Keras tensor or variable, as a string.</th>
</tr>
</thead>
</table>

**Description**

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

**Usage**

\[
k_{dtype}(x)\]

**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**

**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_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](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**

\[
k\_eval(x)
\]
**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**

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**

```r
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.

---

**k_eye**  

*Instantiate an identity matrix and returns it.*

---

**Description**

Instantiate an identity matrix and returns it.

**Usage**

```r
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](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.

---

### 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.
**k_function**

*Instantiates a Keras function*

**Description**

Instantiates a Keras function

**Usage**

```r
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**

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

**Arguments**

- **reference**: A tensor.
- **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

```python
k_get_session()
```

```python
k_set_session(session)
```

Arguments


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.
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_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.

---

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_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](https://keras.rstudio.com/articles/backend.html#backend-functions).

---

### k_greater

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

**Description**

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

**Usage**

```
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](https://keras.rstudio.com/articles/backend.html#backend-functions).
**k_greater_equal**

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

**Description**

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

**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.

---

**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 \(-2.5 \leq x \leq 2.5\), returns \(0.2 \times x + 0.5\).

**Usage**

\[k_{\text{hard\_sigmoid}}(x)\]

**Arguments**

- \(x\)  
  A tensor or variable.

**Value**

A tensor.
k_identity

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()

k_set_image_data_format(data_format)
```
### k_int_shape

**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](https://keras.rstudio.com/articles/backend.html#backend-functions).

<table>
<thead>
<tr>
<th>k_int_shape</th>
<th>Returns the shape of tensor or variable as a list of int or NULL entries.</th>
</tr>
</thead>
</table>

**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](https://keras.rstudio.com/articles/backend.html#backend-functions).
\textbf{k\_in\_test\_phase} \hspace{10em} Selects \( x \) in test phase, and \( \text{alt} \) otherwise.

\textbf{Description}

Note that \( \text{alt} \) should have the \textit{same shape as} \( x \).

\textbf{Usage}

\begin{verbatim}
k_in_test_phase(x, alt, training = NULL)
\end{verbatim}

\textbf{Arguments}

\begin{itemize}
  \item \texttt{x} \hspace{2em} What to return in test phase (tensor or function that returns a tensor).
  \item \texttt{alt} \hspace{2em} What to return otherwise (tensor or function that returns a tensor).
  \item \texttt{training} \hspace{2em} Optional scalar tensor (or R logical or integer) specifying the learning phase.
\end{itemize}

\textbf{Value}

Either \( x \) or \( \text{alt} \) based on \texttt{k\_learning\_phase()}.

\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}.

\textbf{k\_in\_top\_k} \hspace{10em} Returns whether the targets are in the top \( k \) predictions.

\textbf{Description}

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

\textbf{Usage}

\begin{verbatim}
k_in_top_k(predictions, targets, k)
\end{verbatim}

\textbf{Arguments}

\begin{itemize}
  \item \texttt{predictions} \hspace{2em} A tensor of shape (batch\_size, classes) and type float32.
  \item \texttt{targets} \hspace{2em} A 1D tensor of length batch\_size and type int32 or int64.
  \item \texttt{k} \hspace{2em} An int, number of top elements to consider.
\end{itemize}
k_in_train_phase

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.

\[ \text{k_in_train_phase} \]

Selects \( x \) in train phase, and \( \text{alt} \) otherwise.

Description

Note that \( \text{alt} \) should have the same shape as \( x \).

Usage

\[ \text{k_in_train_phase}(x, \text{alt}, \text{training} = \text{NULL}) \]

Arguments

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

Value

Either \( x \) or \( \text{alt} \) based on the training flag. the training flag defaults to \text{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](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
k_is_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_sparse

Returns whether a tensor is a sparse tensor.

**Description**

Returns whether a tensor is a sparse tensor.

**Usage**

k_is_sparse(tensor)

**Arguments**

tensor A tensor instance.

**Value**

A logical

---

k_is_tensor

Returns whether x is a symbolic tensor.

**Description**

Returns whether x is a symbolic tensor.

**Usage**

k_is_tensor(x)

**Arguments**

x A candidate tensor.
### 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

```r
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](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**

```python
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](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**

```python
k_less(x, y)
```

**Arguments**

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

**Value**

A bool tensor.
**k_less_equal**

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

### Description

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

### Usage

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

### Arguments

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

### Value

A bool tensor.

---

**k_local_conv1d**

Apply 1D conv with un-shared weights.

### Description

Apply 1D conv with un-shared weights.

### Usage

```python
k_local_conv1d(inputs, kernel, kernel_size, strides, data_format = NULL)
```
**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](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**

```r
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_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)
```

**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.
k_max

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_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.
**k_maximun**

Element-wise maximum of two tensors.

**Description**

Element-wise maximum of two tensors.

**Usage**

```python
k_maximun(x, y)
```

**Arguments**

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

Mean of a tensor, alongside the specified axis.

**Description**

Mean of a tensor, alongside the specified axis.

**Usage**

```python
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.
**k_min**

**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](https://keras.rstudio.com/articles/backend.html#backend-functions).

---

**k_min**  
*Minimum value in a tensor.*

---

**Description**

Minimum value in a tensor.

**Usage**

```r
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 mimum 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.

**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_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.
**k_ndim**

**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](https://keras.rstudio.com/articles/backend.html#backend-functions).

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

**Description**

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

**Usage**

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

**Arguments**

\(x\) Tensor or variable.

**Value**

Integer (scalar), number of axes.

**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_normalize_batch_in_training**

*Computes mean and std for batch then apply batch_normalization on batch.*

**Description**

Computes mean and std for batch then apply batch_normalization on batch.

**Usage**

```python
k_normalize_batch_in_training(x, gamma, beta, reduction_axes, epsilon = 0.001)
```

**Arguments**

- **x**: Input tensor or variable.
- **gamma**: Tensor by which to scale the input.
- **beta**: Tensor with which to center the input.
- **reduction_axes**: Iterable of integers, axes over which to normalize.
- **epsilon**: 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**

```python
k_not_equal(x, y)
```
$k_{ones}$

**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](https://keras.rstudio.com/articles/backend.html#backend-functions).
k_one_hot

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
k_one_hot(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))
um_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.

Description

Permutes axes in a tensor.

Usage

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.


\textbf{Description}

Instantiates a placeholder tensor and returns it.

\textbf{Usage}

\begin{verbatim}
k_placeholder(
    shape = NULL,
    ndim = NULL,
    dtype = NULL,
    sparse = FALSE,
    name = NULL
)
\end{verbatim}

\textbf{Arguments}

- \texttt{shape} \quad Shape of the placeholder (integer list, may include NULL entries).
- \texttt{ndim} \quad Number of axes of the tensor. At least one of \texttt{shape}, \texttt{ndim} must be specified. If both are specified, \texttt{shape} is used.
- \texttt{dtype} \quad Placeholder type.
- \texttt{sparse} \quad Logical, whether the placeholder should have a sparse type.
- \texttt{name} \quad Optional name string for the placeholder.

\textbf{Value}

Tensor instance (with Keras metadata included).

\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}.  

k_pool2d

2D Pooling.

Description

2D Pooling.

Usage

```
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.
Description

3D Pooling.

Usage

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**

```python
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**

```python
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

k_random_binomial() and k_random_bernoulli() are aliases for the same function. Both are maintained for backwards compatibility. New code should prefer k_random_bernoulli().

Usage

k_random_binomial(shape, p = 0, dtype = NULL, seed = NULL)
k_random_bernoulli(shape, p = 0, dtype = NULL, seed = NULL)

Arguments

- shape: A list of integers, the shape of tensor to create.
- p: A float, 0. <= p <= 1, probability of binomial distribution.
- 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_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.

```
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

```
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.
**k_random_uniform**

**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.
k_random_uniform_variable

*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**

$$k\_relu(x, \alpha = 0, \text{max\_value} = \text{NULL})$$

**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**

$$k\_repeat(x, n)$$

**Arguments**

- **x**: Tensor or variable.
- **n**: Integer, number of times to repeat.
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_repeat_elements  Repeats the elements of a tensor along an axis.

Description

If x has shape (s1, s2, s3) and axis is 2, the output will have shape (s1, s2 * rep, s3).

Usage

k_repeat_elements(x, rep, 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**
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.

**kreshape**
Reshapes a tensor to the specified shape.

**Description**
Reshapes a tensor to the specified shape.

**Usage**
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.
**k_resize_images**  
*Resizes the images contained in a 4D tensor.*

**Description**
Resizes the images contained in a 4D tensor.

**Usage**

```python
k_resize_images(x, height_factor, width_factor, data_format)
```

**Arguments**
- `x` Tensor or variable to resize.
- `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](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**

```python
k_resize_volumes(x, depth_factor, height_factor, width_factor, data_format)
```
**Arguments**

<table>
<thead>
<tr>
<th>x</th>
<th>Tensor or variable to resize.</th>
</tr>
</thead>
<tbody>
<tr>
<td>depth_factor</td>
<td>Positive integer.</td>
</tr>
<tr>
<td>height_factor</td>
<td>Positive integer.</td>
</tr>
<tr>
<td>width_factor</td>
<td>Positive integer.</td>
</tr>
<tr>
<td>data_format</td>
<td>string, &quot;channels_last&quot; or &quot;channels_first&quot;.</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](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**

<table>
<thead>
<tr>
<th>x</th>
<th>Tensor to reverse.</th>
</tr>
</thead>
<tbody>
<tr>
<td>axes</td>
<td>Integer or list of integers of axes to reverse (axis indexes are 1-based).</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](https://keras.rstudio.com/articles/backend.html#backend-functions).
**Description**

Iterates over the time dimension of a tensor

**Usage**

```r
k_rnn(
    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 = FALSE,  # Logical. If TRUE, do the iteration over the time dimension in reverse order and return the reversed sequence.
    mask = NULL,       # Binary tensor with shape (samples, time, 1), with a zero for every element that is masked.
    constants = NULL,  # A list of constant values passed at each step.
    unroll = FALSE,    # Whether to unroll the RNN or to use a symbolic loop (while_loop or scan depending on backend).
    input_length = NULL  # Not relevant in the TensorFlow implementation. Must be specified if using unrolling with Theano.
)
```

**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](https://keras.rstudio.com/articles/backend.html#backend-functions).

---

### `k_round`

**Element-wise rounding to the closest integer.**

**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](https://keras.rstudio.com/articles/backend.html#backend-functions).

---

### `k_separable_conv2d`

**2D convolution with separable filters.**

**Description**

2D convolution with separable filters.
Usage

k_separable_conv2d(
    x,
    depthwise_kernel,
    pointwise_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.
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

k_set_learning_phase(value)
**k_set_value**

Sets the value of a variable, from an R array.

<table>
<thead>
<tr>
<th>Arguments</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>Learning phase value, either 0 or 1 (integers).</td>
</tr>
</tbody>
</table>

**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_set_value(x, value)
```

<table>
<thead>
<tr>
<th>Arguments</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>Tensor to set to a new value.</td>
</tr>
<tr>
<td>value</td>
<td>Value to set the tensor to, as an R array (of the same shape).</td>
</tr>
</tbody>
</table>

**k_shape**

Returns the symbolic shape of a tensor or variable.

```r
ek_shape(x)
```

<table>
<thead>
<tr>
<th>Description</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Returns the symbolic shape of a tensor or variable.</td>
</tr>
</tbody>
</table>
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.

k_sigmoid Element-wise sigmoid.

Description

Element-wise sigmoid.

Usage

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.
**k_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**

**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.

---

**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.

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_softplus**

*Softplus of a tensor.*

**Description**

Softplus of a tensor.

**Usage**

```python
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**

```python
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.
Usage

```python
k_spatial_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.

---

**k_sqrt**

*Element-wise square root.*

Description

Element-wise square root.

Usage

```python
k_sqrt(x)
```

Arguments

- `x`: Tensor or variable.

Value

A tensor.
**k_square**

**Element-wise square.**

**Description**

Element-wise square.

**Usage**

```r
k_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](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**

```r
k_squeeze(x, axis = NULL)
```
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).

---

**k_stack**

Stacks a list of rank `R` tensors into a rank `R+1` tensor.

---

Description

Stacks a list of rank `R` tensors into a rank `R+1` tensor.

Usage

```python
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**
\[
k_{\text{std}}(x, \text{axis} = \text{NULL}, \text{keepdims} = \text{FALSE})
\]

**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**
\[
k_{\text{stop\_gradient}}(\text{variables})
\]

**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

\[
\text{k_sum}(x, \text{axis} = \text{NULL, keepdims} = \text{FALSE})
\]

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**

```python
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](https://keras.rstudio.com/articles/backend.html#backend-functions).

---

**k_tanh**  
*Element-wise tanh.*

**Description**

Element-wise tanh.

**Usage**

```python
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**

```r
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**

*Description*

Creates a tensor by tiling \( x \) by \( n \).

*Usage*

\[k_{\text{tile}}(x, n)\]

*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**

*Description*

Converts a sparse tensor into a dense tensor and returns it.

*Usage*

\[k_{\text{to_dense}}(\text{tensor})\]

*Arguments*

- **tensor**: A tensor instance (potentially sparse).

*Value*

A dense tensor.
**k_transpose**

Transposes a tensor and returns it.

**Usage**

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

**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**

\[
k_{\text{truncated\_normal}}(\text{shape}, \text{mean} = 0, \text{stddev} = 1, \text{dtype} = \text{NULL}, \text{seed} = \text{NULL})
\]
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>mean</td>
<td>Mean of the values.</td>
</tr>
<tr>
<td>stddev</td>
<td>Standard deviation of the values.</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](https://keras.rstudio.com/articles/backend.html#backend-functions).

---

**k_unstack**

Unstack rank R tensor into a list of rank R-1 tensors.

**Description**

Unstack rank R tensor into a list of rank R-1 tensors.

**Usage**

```
k_unstack(x, axis = 1L, num = NULL, name = NULL)
```

**Arguments**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>a tensor.</td>
</tr>
<tr>
<td>axis</td>
<td>Axis along which to perform stacking (axis indexes are 1-based). Negative values wrap around, so the valid range is ([R, -R]).</td>
</tr>
<tr>
<td>num</td>
<td>An int. The length of the dimension axis. Automatically inferred if NULL (the default).</td>
</tr>
<tr>
<td>name</td>
<td>A name for the operation (optional).</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](https://keras.rstudio.com/articles/backend.html#backend-functions).

---

### k_update

**Update the value of** \(x\) **to** \(\text{new\_x}\).

**Description**

Update the value of \(x\) to \(\text{new\_x}\).

**Usage**

\[\text{k\_update}(x, \text{new\_x})\]

**Arguments**

- \(x\): A Variable.
- \(\text{new\_x}\): A tensor of same shape as \(x\).

**Value**

The variable \(x\) updated.

---

### k_update_add

**Update the value of** \(x\) **by adding** \(\text{increment}\).

**Description**

Update the value of \(x\) by adding \(\text{increment}\).

**Usage**

\[\text{k\_update\_add}(x, \text{increment})\]
**Arguments**

- **x**: A Variable.
- **increment**: 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 `decrement`.

---

**Description**

Update the value of `x` by subtracting `decrement`.

**Usage**

```python
k_update_sub(x, decrement)
```

**Arguments**

- **x**: A Variable.
- **decrement**: 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_var**

Variance of a tensor, alongside the specified axis.

**Description**

Variance of a tensor, alongside the specified axis.

**Usage**

```r
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).

**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)
```
k_zeros

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**

```python
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

**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](https://keras.rstudio.com/articles/backend.html#backend-functions).
k_zeros_like

*Instantiates an all-zeros variable of the same shape as another tensor.*

**Description**

Instantiates an all-zeros variable of the same shape as another tensor.

**Usage**

```r
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](https://keras.rstudio.com/articles/backend.html#backend-functions).

layer_activation

*Apply an activation function to an output.*

**Description**

Apply an activation function to an output.

**Usage**

```r
layer_activation(
    object,
    activation,
    input_shape = NULL,
    batch_input_shape = NULL,
    batch_size = NULL,
    dtype = NULL,
    name = NULL,
)```
layer_activation_elu

```r
trainable = NULL,
weights = NULL
```

**Arguments**

- `object` What to compose the new `Layer` instance with. Typically a Sequential model or a Tensor (e.g., as returned by `layer_input()`). The return value depends on `object`. If `object` is:
  - missing or `NULL`, the `Layer` instance is returned.
  - a `Sequential` model, the model with an additional layer is returned.
  - a Tensor, the output tensor from `layer_instance(object)` is returned.

- `activation` Name of activation function to use. If you don’t specify anything, no activation is applied (i.e., "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 (e.g., `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_relu()`, `layer_activation_relu()`, `layer_activation_selu()`, `layer_activation_softmax()`, `layer_activation_thresholded_relu()`

---

**Description**

It follows: \( f(x) = \alpha \times (\exp(x) - 1.0) \) for \( x < 0 \), \( f(x) = x \) for \( x \geq 0 \).
layer_activation_elu

Usage

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 What to compose the new Layer instance with. Typically a Sequential model or a Tensor (e.g., as returned by layer_input()). The return value depends on object. If object is:
  • missing or NULL, the Layer instance is returned.
  • a Sequential model, the model with an additional layer is returned.
  • a Tensor, the output tensor from layer_instance(object) is returned.
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()
**layer_activation_leaky_relu**

*Leaky version of a Rectified Linear Unit.*

**Description**

Allows a small gradient when the unit is not active: \( f(x) = \alpha \cdot x \) 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**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>object</td>
<td>What to compose the new Layer instance with. Typically a Sequential model or a Tensor (e.g., as returned by <code>layer_input()</code>). The return value depends on <code>object</code>. If <code>object</code> is:</td>
</tr>
<tr>
<td></td>
<td>- missing or NULL, the Layer instance is returned.</td>
</tr>
<tr>
<td></td>
<td>- a Sequential model, the model with an additional layer is returned.</td>
</tr>
<tr>
<td></td>
<td>- a Tensor, the output tensor from <code>layer_instance(object)</code> is returned.</td>
</tr>
<tr>
<td>alpha</td>
<td>float ( \geq 0 ). Negative slope coefficient.</td>
</tr>
<tr>
<td>input_shape</td>
<td>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.</td>
</tr>
<tr>
<td>batch_input_shape</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>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_activation_parametric_relu**

*Parametric Rectified Linear Unit.*

**Description**

It follows: \( f(x) = \alpha \times \begin{cases} x \text{ for } x < 0, \\ x \text{ for } x \geq 0 \end{cases} \)\), where \( \alpha \) is a learned array with the same shape as \( x \).

**Usage**

```python
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`: What to compose the new Layer instance with. Typically a Sequential model or a Tensor (e.g., as returned by `layer_input()`). The return value depends on `object`. If `object` is:
  - missing or NULL, the Layer instance is returned.
  - a Sequential model, the model with an additional layer is returned.
  - a Tensor, the output tensor from `layer_instance(object)` is returned.

- `alpha_initializer`: Initializer function for the weights.

- `alpha_regularizer`: Regularizer for the weights.
layer_activation_relu

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).

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_relu

Rectified Linear Unit activation function

Description
Rectified Linear Unit activation function

Usage
layer_activation_relu(
  object,
  max_value = NULL,
  negative_slope = 0,
  threshold = 0,
  input_shape = NULL,
  batch_input_shape = NULL,
  batch_size = NULL,
layer_activation_relu

```r
dtype = NULL,
name = NULL,
trainable = NULL,
weights = NULL
```

Arguments

- **object**: What to compose the new Layer instance with. Typically a Sequential model or a Tensor (e.g., as returned by `layer_input()`). The return value depends on `object`. If `object` is:
  - missing or NULL, the Layer instance is returned.
  - a Sequential model, the model with an additional layer is returned.
  - a Tensor, the output tensor from `layer_instance(object)` is returned.

- **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**

```r
layer_activation_selu(
    object,
    input_shape = NULL,
    batch_input_shape = NULL,
    batch_size = NULL,
    dtype = NULL,
    name = NULL,
    trainable = NULL,
    weights = NULL
)
```

**Arguments**

- **object**  
  What to compose the new Layer instance with. Typically a Sequential model or a Tensor (e.g., as returned by layer_input()). The return value depends on object. If object is:
   - missing or NULL, the Layer instance is returned.
   - a Sequential model, the model with an additional layer is returned.
   - a Tensor, the output tensor from layer_instance(object) is returned.

- **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()
• a Tensor, the output tensor from layer_instance(object) is returned.

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()
Arguments

object  What to compose the new Layer instance with. Typically a Sequential model or a Tensor (e.g., as returned by layer_input()). The return value depends on object. If object is:
  • missing or NULL, the Layer instance is returned.
  • a Sequential model, the model with an additional layer is returned.
  • a Tensor, the output tensor from layer_instance(object) is returned.

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()
Arguments

object What to compose the new Layer instance with. Typically a Sequential model or a Tensor (e.g., as returned by layer_input()). The return value depends on object. If object is:

- missing or NULL, the Layer instance is returned.
- a Sequential model, the model with an additional layer is returned.
- a Tensor, the output tensor from layer_instance(object) is returned.

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

layer_add(inputs, ...)

Arguments

inputs A list of input tensors (at least 2). Can be missing.
... Standard layer arguments (must be named).

Value

A tensor, the sum of the inputs. If inputs is missing, a keras layer instance is returned.

See Also

• https://www.tensorflow.org/api_docs/python/tf/keras/layers/add
• https://www.tensorflow.org/api_docs/python/tf/keras/layers/Add
• https://keras.io/api/layers/merging_layers/add

layer_additive_attention

Additive attention layer, a.k.a. Bahdanau-style attention

Description

Additive attention layer, a.k.a. Bahdanau-style attention

Usage

layer_additive_attention(
  object,
  use_scale = TRUE,
  ...,
  causal = FALSE,
  dropout = 0
)
layer_alpha_dropout

Arguments

object        What to compose the new Layer instance with. Typically a Sequential model or a Tensor (e.g., as returned by layer_input()). The return value depends on object. If object is:
  - missing or NULL, the Layer instance is returned.
  - a Sequential model, the model with an additional layer is returned.
  - a Tensor, the output tensor from layer_instance(object) is returned.

use_scale    If TRUE, will create a 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.

dropout      Float between 0 and 1. Fraction of the units to drop for the attention scores.

Details

Inputs are query tensor of shape [batch_size, Tq, dim], value tensor of shape [batch_size, Tv, dim] and key tensor of shape [batch_size, Tv, dim]. The calculation follows the steps:

1. Reshape query and key into shapes [batch_size, Tq, 1, dim] and [batch_size, 1, Tv, dim] respectively.
2. Calculate scores with shape [batch_size, Tq, Tv] as a non-linear sum: scores = tf.reduce_sum(tf.tanh(query + key), axis=-1)
3. Use scores to calculate a distribution with shape [batch_size, Tq, Tv]: distribution = tf$nn$softmax(scores).
4. Use distribution to create a linear combination of value with shape [batch_size, Tq, dim]: return tf$matmul$(distribution, value).

See Also

- https://www.tensorflow.org/api_docs/python/tf/keras/layers/AdditiveAttention
- https://keras.io/api/layers/attention_layers/additive_attention/

layer_alpha_dropout (object, rate, noise_shape = NULL, seed = NULL, ...)

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

layer_alpha_dropout(object, rate, noise_shape = NULL, seed = NULL, ...)
Arguments

- **object**: What to compose the new Layer instance with. Typically a Sequential model or a Tensor (e.g., as returned by `layer_input()`). The return value depends on object. If object is:
  - missing or NULL, the Layer instance is returned.
  - a Sequential model, the model with an additional layer is returned.
  - a Tensor, the output tensor from `layer_instance(object)` is returned.

- **rate**: float, drop probability (as with `layer_dropout()`). The multiplicative noise will have standard deviation \( \sqrt{\text{rate} / (1 - \text{rate})} \).

- **noise_shape**: Noise shape

- **seed**: An integer to use as random seed.

... standard layer arguments.

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

- [https://www.tensorflow.org/api_docs/python/tf/keras/layers/AlphaDropout](https://www.tensorflow.org/api_docs/python/tf/keras/layers/AlphaDropout)

Other noise layers: `layer_gaussian_dropout()`, `layer_gaussian_noise()`

---

**layer_attention**

*Creates attention layer*

Description

Dot-product attention layer, a.k.a. Luong-style attention.
layer_average

Usage

    layer_attention(
        inputs,
        use_scale = FALSE,
        causal = FALSE,
        batch_size = NULL,
        dtype = NULL,
        name = NULL,
        trainable = NULL,
        weights = NULL
    )

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_regularization(), layer_dense_features(),
layer_dense(), layer_dropout(), layer_flatten(), layer_input(), layer_lambda(), layer_masking(),
layer_permute(), layer_repeat_vector(), layer_reshape()

layer_average          Layer that averages 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_average(inputs, ...)

layer_average_pooling_1d

Arguments

inputs  A list of input tensors (at least 2). Can be missing.
...
Standard layer arguments (must be named).

Value

A tensor, the average of the inputs. If inputs is missing, a keras layer instance is returned.

See Also

- https://www.tensorflow.org/api_docs/python/tf/keras/layers/average
- https://www.tensorflow.org/api_docs/python/tf/keras/layers/Average
- https://keras.io/api/layers/merging_layers/average

Other merge layers: 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

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  What to compose the new Layer instance with. Typically a Sequential model or a Tensor (e.g., as returned by layer_input()). The return value depends on object. If object is:
  • missing or NULL, the Layer instance is returned.
  • a Sequential model, the model with an additional layer is returned.
layer_average_pooling_2d

• a Tensor, the output tensor from layer_instance(object) is returned.

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 di-
            mensions 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

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,
layer_average_pooling_2d

name = NULL,
trainable = NULL,
weights = NULL
)

Arguments

object What to compose the new Layer instance with. Typically a Sequential model or a Tensor (e.g., as returned by layer_input()). The return value depends on object. If object is:
  • missing or NULL, the Layer instance is returned.
  • a Sequential model, the model with an additional layer is returned.
  • a Tensor, the output tensor from layer_instance(object) is returned.
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".
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()
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

```r
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**: What to compose the new Layer instance with. Typically a Sequential model or a Tensor (e.g., as returned by `layer_input()`). The return value depends on `object`. If `object` is:
  - missing or NULL, the Layer instance is returned.
  - a Sequential model, the model with an additional layer is returned.
  - a Tensor, the output tensor from `layer_instance(object)` is returned.

- **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

Description

Batch normalization layer (Ioffe and Szegedy, 2014).

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: What to compose the new Layer instance with. Typically a Sequential model or a Tensor (e.g., as returned by layer_input()). The return value depends on object. If object is:
  • missing or NULL, the Layer instance is returned.
  • a Sequential model, the model with an additional layer is returned.
  • a Tensor, the output tensor from layer_instance(object) is returned.

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.
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.

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 \(\text{corrected\_value} = \text{normalized\_value} \times r + d\), with \(r\) clipped to \([r\text{min}, r\text{max}]\), and \(d\) to \([-d\text{max}, d\text{max}]\). Missing \(r\text{max}\), \(r\text{min}\), \(d\text{max}\) are set to \(\text{Inf}, 0, \text{Inf}\), respectively.

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.

TRUE, use a faster, fused implementation, or raise a ValueError if the fused implementation cannot be used. If NULL, use the faster implementation if possible. If FALSE, do not use the fused implementation.

An integer. By default, virtual\_batch\_size is NULL, which means batch normalization 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.

A function taking the Tensor containing the (dynamic) shape of the input tensor and returning a pair \((\text{scale}, \text{bias})\) to apply to the normalized values (before gamma and beta), only during training. For example, if \(\text{axis}=-1\), \(\text{adjustment} <- \text{function}(\text{shape})\{ \text{tuple}(\text{tf}\$\text{random}\$\text{uniform}(\text{shape}[\text{-1:NULL}], 0.93, 1.07), \text{tf}\$\text{random}\$\text{uniform}(\text{shape}[\text{-1:NULL}, \text{style} = \text{"python"}], -0.1, 0.1)) \}\) 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.

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.

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.

Fixed batch size for layer

The data type expected by the input, as a string (float32, float64, int32...)

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.
**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_category_encoding**

*A preprocessing layer which encodes integer features.*

---

**Description**

This layer provides options for condensing data into a categorical encoding when the total number of tokens are known in advance. It accepts integer values as inputs, and it outputs a dense or sparse representation of those inputs. For integer inputs where the total number of tokens is not known, use layer_integer_lookup() instead.

**Usage**

```r
layer_category_encoding(
  object,
  num_tokens = NULL,
  output_mode = "multi_hot",
  sparse = FALSE,
  ...
)
```

**Arguments**

- **object**
  What to compose the new Layer instance with. Typically a Sequential model or a Tensor (e.g., as returned by layer_input()). The return value depends on object. If object is:
  - missing or NULL, the Layer instance is returned.
  - a Sequential model, the model with an additional layer is returned.
  - a Tensor, the output tensor from layer_instance(object) is returned.

- **num_tokens**
  The total number of tokens the layer should support. All inputs to the layer must integers in the range $0 \leq value < num_tokens$, or an error will be thrown.

- **output_mode**
  Specification for the output of the layer. Defaults to "multi_hot". Values can be "one_hot", "multi_hot" or "count", configuring the layer as follows:
• "one_hot": Encodes each individual element in the input into an array of num_tokens size, containing a 1 at the element index. If the last dimension is size 1, will encode on that dimension. If the last dimension is not size 1, will append a new dimension for the encoded output.

• "multi_hot": Encodes each sample in the input into a single array of num_tokens size, containing a 1 for each vocabulary term present in the sample. Treats the last dimension as the sample dimension, if input shape is (... , sample_length), output shape will be (... , num_tokens).

• "count": Like "multi_hot", but the int array contains a count of the number of times the token at that index appeared in the sample.

For all output modes, currently only output up to rank 2 is supported.

sparse Boolean. If TRUE, returns a SparseTensor instead of a dense Tensor. Defaults to FALSE.

... standard layer arguments.

See Also

- https://www.tensorflow.org/api_docs/python/tf/keras/layers/CategoryEncoding
- https://keras.io/api/layers/preprocessing_layers/categorical/category_encoding/

Other categorical features preprocessing layers: layer_hashing(), layer_integer_lookup(), layer_string_lookup()

Other preprocessing layers: layer_center_crop(), layer_discretization(), layer_hashing(), layer_integer_lookup(), layer_normalization(), layer_random_brightness(), layer_random_contrast(), layer_random_crop(), layer_random_flip(), layer_random_height(), layer_random_rotation(), layer_random_translation(), layer_random_width(), layer_random_zoom(), layer_rescaling(), layer_resizing(), layer_string_lookup(), layer_text_vectorization()
layer_concatenate

Layer that concatenates a list of inputs.

Description

It takes as input a list of tensors, all of the same shape except for the concatenation axis, and returns a single tensor, the concatenation of all inputs.

Usage

layer_concatenate(inputs, axis = -1, ...)

Arguments

inputs A list of input tensors (at least 2). Can be missing.
axis Concatenation axis.
... Standard layer arguments (must be named).
**Value**

A tensor, the concatenation of the inputs alongside axis axis. If inputs is missing, a keras layer instance is returned.

**See Also**

- [https://www.tensorflow.org/api_docs/python/tf/keras/layers/concatenate](https://www.tensorflow.org/api_docs/python/tf/keras/layers/concatenate)
- [https://www.tensorflow.org/api_docs/python/tf/keras/layers/Concatenate](https://www.tensorflow.org/api_docs/python/tf/keras/layers/Concatenate)
- [https://keras.io/api/layers/merging_layers/concatenate](https://keras.io/api/layers/merging_layers/concatenate)

Other merge layers: `layer_average()`, `layer_dot()`, `layer_maximum()`, `layer_minimum()`, `layer_multiply()`, `layer_subtract()`

---

**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**
  - What to compose the new Layer instance with. Typically a Sequential model or a Tensor (e.g., as returned by `layer_input()`). The return value depends on object. If object is:
    - missing or NULL, the Layer instance is returned.
    - a Sequential model, the model with an additional layer is returned.
    - a Tensor, the output tensor from `layer_instance(object)` is returned.

- **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. 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_1d(), 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()
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.

```
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  What to compose the new Layer instance with. Typically a Sequential model or a Tensor (e.g., as returned by layer_input()). The return value depends on object. If object is:
- missing or NULL, the Layer instance is returned.
- a Sequential model, the model with an additional layer is returned.
- a Tensor, the output tensor from layer_instance(object) is returned.

**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.
layer_conv_2d

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<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>

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_timesteps = ((timesteps - 1) * strides + kernel_size - 2 * padding + output_padding)

References

• A guide to convolution arithmetic for deep learning

See Also

Other convolutional layers: 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_1d(), 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_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

```r
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
)
```

Arguments

- **object**
  What to compose the new Layer instance with. Typically a Sequential model or a Tensor (e.g., as returned by `layer_input()`). The return value depends on `object`. If `object` is:
  - missing or `NULL`, the Layer instance is returned.
  - a `Sequential` model, the model with an additional layer is returned.
  - a `Tensor`, the output tensor from `layer_instance(object)` is returned.

- **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.

**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_conv_2d_transpose

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_conv_lstm_2d(), layer_cropping_1d(), layer_cropping_2d(), layer_cropping_3d(), layer_depthwise_conv_1d(), 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_2d_transpose

Transposed 2D 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 (list of integers, does not include the sample axis), e.g. input_shape=c(128L, 128L, 3L) for 128x128 RGB pictures in data_format="channels_last".

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"
layer_conv_2d_transpose

```python
ekernel_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**: What to compose the new Layer instance with. Typically a Sequential model or a Tensor (e.g., as returned by `layer_input()`). The return value depends on `object`. If `object` is:
  - missing or NULL, the Layer instance is returned.
  - a Sequential model, the model with an additional layer is returned.
  - a Tensor, the output tensor from `layer_instance(object)` is returned.

- **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**: Dilation rate.

- **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

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_1d()`, `layer_depthwise_conv_2d()`, `layer_depthwise_conv_3d()`, `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_3d**

3D convolution layer (e.g. spatial convolution over volumes).

**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,
  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_3d

Arguments

object
What to compose the new Layer instance with. Typically a Sequential model
or a Tensor (e.g., as returned by layer_input()). The return value depends on
object. If object is:

- missing or NULL, the Layer instance is returned.
- a Sequential model, the model with an additional layer is returned.
- a Tensor, the output tensor from layer_instance(object) is returned.

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
(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").
layer_conv_3d

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_1d(), 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_3d_transpose

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

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 What to compose the new Layer instance with. Typically a Sequential model or a Tensor (e.g., as returned by layer_input()). The return value depends on object. If object is:

- missing or NULL, the Layer instance is returned.
• a Sequential model, the model with an additional layer is returned.
• a Tensor, the output tensor from layer_instance(object) is returned.

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 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.
layer_conv_lstm_1d

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
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_1d(), 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()
dilation_rate = 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,
go_backwards = FALSE,
stateful = FALSE,
dropout = 0,
recurrent_dropout = 0,
...)

Arguments

object What to compose the new Layer instance with. Typically a Sequential model
or a Tensor (e.g., as returned by layer_input()). The return value depends on
object. If object is:
  • missing or NULL, the Layer instance is returned.
  • a Sequential model, the model with an additional layer is returned.
  • a Tensor, the output tensor from layer_instance(object) is returned.

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 n integers, specifying the dimensions of the convolution
window.

strides An integer or list of n integers, specifying the strides of the convolution. Speci-
fying any stride value != 1 is incompatible with specifying any dilation_rate
value != 1.

padding One of "valid" or "same" (case-insensitive). "valid" means no padding.
"same" results in padding evenly to the left/right or up/down of the input such
that output has the same height/width dimension as the input.

data_format 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 "channels_last".

dilation_rate 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.

activation Activation function to use. By default hyperbolic tangent activation function is applied (tanh(x)).

recurrent_activation Activation function to use for the recurrent step.

use_bias Boolean, whether the layer uses a bias vector.

kernel_initializer Initializer for the kernel weights matrix, used for the linear transformation of the inputs.

recurrent_initializer Initializer for the recurrent_kernel weights matrix, used for the linear transformation of the recurrent state.

bias_initializer Initializer for the bias vector.

unit_forget_bias Boolean. If TRUE, add 1 to the bias of the forget gate at initialization. Use in combination with bias_initializer="zeros". This is recommended in Jozefowicz et al., 2015

kernel_regularizer Regularizer function applied to the kernel weights matrix.

recurrent_regularizer Regularizer function applied to the recurrent_kernel weights matrix.

bias_regularizer Regularizer function applied to the bias vector.

activity_regularizer Regularizer function applied to.

kernel_constraint Constraint function applied to the kernel weights matrix.

recurrent_constraint Constraint function applied to the recurrent_kernel weights matrix.

bias_constraint Constraint function applied to the bias vector.

return_sequences Boolean. Whether to return the last output in the output sequence, or the full sequence. (default FALSE)

return_state Boolean. Whether to return the last state in addition to the output. (default FALSE)

go_backwards Boolean (default FALSE). If TRUE, process the input sequence backwards.

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.
Hello!
bias_constraint = NULL,
return_sequences = FALSE,
return_state = FALSE,
go_backwards = FALSE,
stateful = FALSE,
dropout = 0,
recurrent_dropout = 0,
batch_size = NULL,
name = NULL,
trainable = NULL,
weights = NULL,
input_shape = NULL
}

Arguments

object What to compose the new Layer instance with. Typically a Sequential model or a Tensor (e.g., as returned by `layer_input()`). The return value depends on object. If object is:

- missing or NULL, the Layer instance is returned.
- a Sequential model, the model with an additional layer is returned.
- a Tensor, the output tensor from `layer_instance(object)` is returned.

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 n integers, specifying the dimensions of the convolution window.

strides 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 \(!= 1\).

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, 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 "channels_last".

dilation_rate 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 \(!= 1\).

activation Activation function to use. If you don’t specify anything, 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.
**kernel_initializer**

Initializer for the kernel weights matrix, used for the linear transformation of the inputs..

**recurrent_initializer**

Initializer for the recurrent_kernel weights matrix, used for the linear transformation of the recurrent state..

**bias_initializer**

Initializer for the bias vector.

**unit_forget_bias**

Boolean. If TRUE, add 1 to the bias of the forget gate at initialization. Use in combination with bias_initializer="zeros". This is recommended in Jozefowicz et al.

**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, process 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.

**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_1d(), 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_3d 3D Convolutional LSTM

Description

3D Convolutional LSTM

Usage

```
layer_conv_lstm_3d(
  object,
  filters,
  kernel_size,
  strides = c(1L, 1L, 1L),
  padding = "valid",
  data_format = NULL,
  dilation_rate = c(1L, 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,
)```

layer_conv_lstm_3d

recurrent_constraint = NULL,
bias_constraint = NULL,
return_sequences = FALSE,
return_state = FALSE,
go_backwards = FALSE,
stateful = FALSE,
dropout = 0,
recurrent_dropout = 0,
...)
)

Arguments

object  What to compose the new Layer instance with. Typically a Sequential model or a Tensor (e.g., as returned by layer_input()). The return value depends on object. If object is:
• missing or NULL, the Layer instance is returned.
• a Sequential model, the model with an additional layer is returned.
• a Tensor, the output tensor from layer_instance(object) is returned.

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 n integers, specifying the dimensions of the convolution window.

strides  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.

padding  One of "valid" or "same" (case-insensitive). "valid" means no padding. "same" results in padding evenly to the left/right or up/down of the input such that output has the same height/width dimension as the input.

data_format  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 "channels_last".

dilation_rate  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.

activation  Activation function to use. By default hyperbolic tangent activation function is applied (tanh(x)).

recurrent_activation  Activation function to use for the recurrent step.

use_bias  Boolean, whether the layer uses a bias vector.

kernel_initializer  Initializer for the kernel weights matrix, used for the linear transformation of the inputs.
layer_conv_lstm_3d

- **recurrent_initializer**
  Initializer for the recurrent_kernel weights matrix, used for the linear transformation of the recurrent state.

- **bias_initializer**
  Initializer for the bias vector.

- **unit_forget_bias**
  Boolean. If TRUE, add 1 to the bias of the forget gate at initialization. Use in combination with bias_initializer="zeros". This is recommended in Jozefowicz et al., 2015

- **kernel_regularizer**
  Regularizer function applied to the kernel weights matrix.

- **recurrent_regularizer**
  Regularizer function applied to the recurrent_kernel weights matrix.

- **bias_regularizer**
  Regularizer function applied to the bias vector.

- **activity_regularizer**
  Regularizer function applied to.

- **kernel_constraint**
  Constraint function applied to the kernel weights matrix.

- **recurrent_constraint**
  Constraint function applied to the recurrent_kernel weights matrix.

- **bias_constraint**
  Constraint function applied to the bias vector.

- **return_sequences**
  Boolean. Whether to return the last output in the output sequence, or the full sequence. (default FALSE)

- **return_state**
  Boolean Whether to return the last state in addition to the output. (default FALSE)

- **go_backwards**
  Boolean (default FALSE). If TRUE, process the input sequence backwards.

- **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.

- **...**
  standard layer arguments.

**Details**

Similar to an LSTM layer, but the input transformations and recurrent transformations are both convolutional.

**See Also**

- [https://www.tensorflow.org/api_docs/python/tf/keras/layers/ConvLSTM3D](https://www.tensorflow.org/api_docs/python/tf/keras/layers/ConvLSTM3D)
layer_cropping_1d  

Cropping layer for 1D input (e.g. temporal sequence).

Description

It crops along the time dimension (axis 1).

Usage

```r
layer_cropping_1d(
  object,
  cropping = c(1L, 1L),
  batch_size = NULL,
  name = NULL,
  trainable = NULL,
  weights = NULL
)
```

Arguments

- **object**: What to compose the new Layer instance with. Typically a Sequential model or a Tensor (e.g., as returned by `layer_input()`). The return value depends on `object`. If `object` is:
  - missing or `NULL`, the `Layer` instance is returned.
  - a `Sequential` model, the model with an additional layer is returned.
  - a Tensor, the output tensor from `layer_instance(object)` is returned.

- **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)
layer_cropping_2d

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_1d(), 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

layer_cropping_2d(
  object,
  cropping = list(c(0L, 0L), c(0L, 0L)),
  data_format = NULL,
  batch_size = NULL,
  name = NULL,
  trainable = NULL,
  weights = NULL
)

Arguments

object What to compose the new Layer instance with. Typically a Sequential model or a Tensor (e.g., as returned by layer_input()). The return value depends on object. If object is:
  • missing or NULL, the Layer instance is returned.
  • a Sequential model, the model with an additional layer is returned.
  • a Tensor, the output tensor from layer_instance(object) is returned.
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".
layer_cropping_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
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_1d(), 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
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**
What to compose the new Layer instance with. Typically a Sequential model or a Tensor (e.g., as returned by `layer_input()`). The return value depends on `object`. If `object` is:
- missing or NULL, the Layer instance is returned.
- a Sequential model, the model with an additional layer is returned.
- a Tensor, the output tensor from `layer_instance(object)` is returned.

**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)`

**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_1d()`, `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_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,
    trainable = NULL,
    weights = NULL
)

Arguments

object What to compose the new Layer instance with. Typically a Sequential model or a Tensor (e.g., as returned by layer_input()). The return value depends on object. If object is:
- missing or NULL, the Layer instance is returned.
- a Sequential model, the model with an additional layer is returned.
- a Tensor, the output tensor from layer_instance(object) is returned.
units Positive integer, dimensionality of the output space.
activation Name of activation function to use. If you don’t specify anything, no activation is applied (ie. "linear" activation: a(x) = x).
use_bias: Whether the layer uses a bias vector.

kernel_initializer: Initializer for the kernel weights matrix.

bias_initializer: Initializer for the bias vector.

kernel_regularizer: Regularizer function applied to the kernel weights matrix.

bias_regularizer: Regularizer function applied to the bias vector.

activity_regularizer: Regularizer function applied to the output of the layer (its "activation").

kernel_constraint: Constraint function applied to the kernel weights matrix.

bias_constraint: Constraint function applied to the bias vector.

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 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_regularization(), 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**

```r
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**
  What to compose the new Layer instance with. Typically a Sequential model or a Tensor (e.g., as returned by `layer_input()`). The return value depends on `object`. If `object` is:
    - missing or `NULL`, the Layer instance is returned.
    - a `Sequential` model, the model with an additional layer is returned.
    - a Tensor, the output tensor from `layer_instance(object)` is returned.

- **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_1d

Depthwise 1D convolution

Description

Depthwise 1D convolution

Usage

layer_depthwise_conv_1d(
    object,
    kernel_size,
    strides = 1L,
    padding = "valid",
    depth_multiplier = 1L,
    data_format = NULL,
    dilation_rate = 1L,
    activation = NULL,
    use_bias = TRUE,
    depthwise_initializer = "glorot_uniform",
    bias_initializer = "zeros",
    depthwise_regularizer = NULL,
    bias_regularizer = NULL,
    activity_regularizer = NULL,
    depthwise_constraint = NULL,
    bias_constraint = NULL,
    ...
)

Arguments

object What to compose the new Layer instance with. Typically a Sequential model or a Tensor (e.g., as returned by layer_input()). The return value depends on object. If object is:

• missing or NULL, the Layer instance is returned.
• a Sequential model, the model with an additional layer is returned.
• a Tensor, the output tensor from layer_instance(object) is returned.

kernel_size  An integer, specifying the height and width of the 1D convolution window. Can
be a single integer to specify the same value for all spatial dimensions.

strides  An integer, specifying the strides of the convolution along the 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). "valid" means no padding. "same"
results in padding with zeros evenly to the left/right or up/down of the input such
that output has the same height/width dimension as the input.

depth_multiplier  The number of depthwise convolution output channels for each input channel.
The total number of depthwise convolution output channels will be equal to
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_size, height, width, channels) while channels_first corresponds to inputs with shape
(batch_size, channels, height, width). It defaults to the image_data_format value found in your Keras config file at
~/.keras/keras.json. If you never set it, then it will be 'channels_last'.

dilation_rate  A single integer, specifying the dilation rate to use for dilated convolution. Cur-
rently, specifying any dilation_rate value != 1 is incompatible with specify-
ing any stride value != 1.

activation  Activation function to use. If you don’t specify anything, no activation is applied
(see ?activation_relu).

use_bias  Boolean, whether the layer uses a bias vector.

depthwise_initializer  Initializer for the depthwise kernel matrix (see initializer_glorot_uniform).
If NULL, the default initializer ("glorot_uniform") will be used.

bias_initializer  Initializer for the bias vector (see keras.initializers). If NULL, the default
initializer (‘zeros’) will be used.

depthwise_regularizer  Regularizer function applied to the depthwise kernel matrix (see regularizer_l1()).

bias_regularizer  Regularizer function applied to the bias vector (see regularizer_l1()).

activity_regularizer  Regularizer function applied to the output of the layer (its 'activation') (see
regularizer_l1()).

depthwise_constraint  Constraint function applied to the depthwise kernel matrix (see constraint_maxnorm()).

bias_constraint  Constraint function applied to the bias vector (see constraint_maxnorm()).

...  standard layer arguments.
Details

Depthwise convolution is a type of convolution in which each input channel is convolved with a different kernel (called a depthwise kernel). You can understand depthwise convolution as the first step in a depthwise separable convolution.

It is implemented via the following steps:

- Split the input into individual channels.
- Convolve each channel with an individual depthwise kernel with depth_multiplier output channels.
- Concatenate the convolved outputs along the channels axis.

Unlike a regular 1D convolution, depthwise convolution does not mix information across different input channels.

The depth_multiplier argument determines how many filter are applied to one input channel. As such, it controls the amount of output channels that are generated per input channel in the depthwise step.

See Also

- [https://www.tensorflow.org/api_docs/python/tf/keras/layers/DepthwiseConv1D](https://www.tensorflow.org/api_docs/python/tf/keras/layers/DepthwiseConv1D)

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_1stl_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()`

---

layer_depthwise_conv_2d

*Depthwise separable 2D convolution.*

---

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
)

Arguments

object What to compose the new Layer instance with. Typically a Sequential model or a Tensor (e.g., as returned by layer_input()). The return value depends on object. If object is:
• missing or NULL, the Layer instance is returned.
• a Sequential model, the model with an additional layer is returned.
• a Tensor, the output tensor from layer_instance(object) is returned.
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 di-
layer_depthwise_conv_2d

- 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 (ie. "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.

- batch_input_shape Shapes, including the batch size. For instance, \( \text{batch_input_shape} = c(10, 32) \) indicates that the expected input will be batches of 10 32-dimensional vectors. \( \text{batch_input_shape} = \text{list}(\text{NULL}, 32) \) 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 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_1d(), 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**

A preprocessing layer which buckets continuous features by ranges.

**Usage**

```r
layer_discretization(
  object,
  bin_boundaries = NULL,
  num_bins = NULL,
  epsilon = 0.01,
  ...
)
```

**Arguments**

- **object**
  What to compose the new Layer instance with. Typically a Sequential model or a Tensor (e.g., as returned by `layer_input()`). The return value depends on `object`. If `object` is:
  - missing or NULL, the Layer instance is returned.
  - a Sequential model, the model with an additional layer is returned.
  - a Tensor, the output tensor from `layer_instance(object)` is returned.

- **bin_boundaries**
  A list of bin boundaries. The leftmost and rightmost bins will always extend to -Inf and Inf, so `bin_boundaries = c(0., 1., 2.)` generates bins (-Inf, 0.), [0., 1.), [1., 2.), and [2., +Inf). If this option is set, `adapt` should not be called.

- **num_bins**
  The integer number of bins to compute. If this option is set, `adapt` should be called to learn the bin boundaries.

- **epsilon**
  Error tolerance, typically a small fraction close to zero (e.g. 0.01). Higher values of epsilon increase the quantile approximation, and hence result in more unequal buckets, but could improve performance and resource consumption.

- **...**
  standard layer arguments.

**Details**

This layer will place each element of its input data into one of several contiguous ranges and output an integer index indicating which range each element was placed in.

Input shape: Any `tf.Tensor` or `tf.RaggedTensor` of dimension 2 or higher.

Output shape: Same as input shape.
See Also

- `adapt()`
- https://www.tensorflow.org/api_docs/python/tf/keras/layers/Discretization
- https://keras.io/api/layers/preprocessing_layers/numerical/discretization

Other numerical features preprocessing layers: `layer_normalization()`

Other preprocessing layers: `layer_category_encoding()`, `layer_center_crop()`, `layer_hashing()`, `layer_integer_lookup()`, `layer_normalization()`, `layer_random_brightness()`, `layer_random_contrast()`, `layer_random_crop()`, `layer_random_flip()`, `layer_random_height()`, `layer_random_rotation()`, `layer_random_translation()`, `layer_random_width()`, `layer_random_zoom()`, `layer_rescaling()`, `layer_resizing()`, `layer_string_lookup()`, `layer_text_vectorization()`

---

**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**

```
layer_dot(inputs, axes, normalize = FALSE, ...)
```

**Arguments**

- **inputs**
  A list of input tensors (at least 2). Can be missing.
- **axes**
  Integer or list of integers, axis or axes along which to take the dot product.
- **normalize**
  Whether to L2-normalize samples along the dot product axis before taking the dot product. If set to TRUE, then the output of the dot product is the cosine proximity between the two samples.
- **...**
  Standard layer arguments (must be named).

**Value**

If `inputs` is supplied: A tensor, the dot product of the samples from the inputs. If `inputs` is missing, a keras layer instance is returned.

**See Also**

- https://www.tensorflow.org/api_docs/python/tf/keras/layers/dot
- https://www.tensorflow.org/api_docs/python/tf/keras/layers/Dot
- https://keras.io/api/layers/merging_layers/dot/

Other merge layers: `layer_average()`, `layer_concatenate()`, `layer_maximum()`, `layer_minimum()`, `layer_multiply()`, `layer_subtract()`
**Description**

Dropout consists in randomly setting a fraction 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**: What to compose the new Layer instance with. Typically a Sequential model or a Tensor (e.g., as returned by `layer_input()`). The return value depends on `object`. If `object` is:
  - missing or NULL, the Layer instance is returned.
  - a Sequential model, the model with an additional layer is returned.
  - a Tensor, the output tensor from `layer_instance(object)` is returned.

- **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,
  activity_regularizer = NULL,
  embeddings_constraint = NULL,
  mask_zero = FALSE,
  input_length = NULL,
  batch_size = NULL,
  name = NULL,
  trainable = NULL,
  weights = NULL
)
```

Arguments

object  What to compose the new Layer instance with. Typically a Sequential model or a Tensor (e.g., as returned by layer_input()). The return value depends on object. If object is:

- missing or NULL, the Layer instance is returned.
- a Sequential model, the model with an additional layer is returned.
- a Tensor, the output tensor from layer_instance(object) is returned.
layer_embedding

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**: What to compose the new Layer instance with. Typically a Sequential model or a Tensor (e.g., as returned by `layer_input()`). The return value depends on `object`. If `object` is:
  - missing or NULL, the Layer instance is returned.
  - a Sequential model, the model with an additional layer is returned.
  - a Tensor, the output tensor from `layer_instance(object)` is returned.

- **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

```r
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**
  What to compose the new Layer instance with. Typically a Sequential model or a Tensor (e.g., as returned by `layer_input()`). The return value depends on `object`. If `object` is:
    - missing or NULL, the Layer instance is returned.
    - a Sequential model, the model with an additional layer is returned.
    - a Tensor, the output tensor from `layer_instance(object)` is returned.

- **rate**
  float, drop probability (as with Dropout). The multiplicative noise will have standard deviation $\sqrt{\text{rate} / (1 - \text{rate})}$.

- **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
layer_gaussian_noise

- **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**


**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**

```python
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**
  What to compose the new Layer instance with. Typically a Sequential model or a Tensor (e.g., as returned by `layer_input()`). The return value depends on `object`. If `object` is:
  - missing or NULL, the Layer instance is returned.
  - a Sequential model, the model with an additional layer is returned.
  - a Tensor, the output tensor from `layer_instance(object)` is returned.

- **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.

- **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.
layer_global_average_pooling_1d

Usage

layer_global_average_pooling_1d(
    object,
    data_format = "channels_last",
    keepdims = FALSE,
    ...
)

Arguments

object What to compose the new Layer instance with. Typically a Sequential model or a Tensor (e.g., as returned by `layer_input()`). The return value depends on object. If object is:

- missing or NULL, the Layer instance is returned.
- a Sequential model, the model with an additional layer is returned.
- a Tensor, the output tensor from `layer_instance(object)` is returned.

data_format One of `channels_last` (default) or `channels_first`. The ordering of the dimensions in the inputs.

keepdims A boolean, whether to keep the spatial dimensions or not. If keepdims is FALSE (default), the rank of the tensor is reduced for spatial dimensions. If keepdims is TRUE, the spatial dimensions are retained with length 1. The behavior is the same as for `tf.reduce_mean` or `np.mean`.

... standard layer arguments.

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()`
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,
    keepdims = FALSE,
    ...
)

Arguments

object
What to compose the new Layer instance with. Typically a Sequential model or a Tensor (e.g., as returned by layer_input()). The return value depends on object. If object is:
  • missing or NULL, the Layer instance is returned.
  • a Sequential model, the model with an additional layer is returned.
  • a Tensor, the output tensor from layer_instance(object) is returned.

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".

keepdims
A boolean, whether to keep the spatial dimensions or not. If keepdims is FALSE (default), the rank of the tensor is reduced for spatial dimensions. If keepdims is TRUE, the spatial dimensions are retained with length 1. The behavior is the same as for tf.reduce_mean or np.mean.

...standard layer arguments.

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()`

---

**layer_global_average_pooling_3d**

*Global Average pooling operation for 3D data.*

---

**Description**

Global Average pooling operation for 3D data.

**Usage**

```r
layer_global_average_pooling_3d(
  object,
  data_format = NULL,
  keepdims = FALSE,
  ...
)
```

**Arguments**

- **object** What to compose the new Layer instance with. Typically a Sequential model or a Tensor (e.g., as returned by `layer_input()`). The return value depends on `object`. If `object` is:
  - missing or NULL, the Layer instance is returned.
  - a Sequential model, the model with an additional layer is returned.
  - a Tensor, the output tensor from `layer_instance(object)` is returned.

- **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".

- **keepdims** A boolean, whether to keep the spatial dimensions or not. If `keepdims` is `FALSE` (default), the rank of the tensor is reduced for spatial dimensions. If `keepdims` is `TRUE`, the spatial dimensions are retained with length 1. The behavior is the same as for `tf.reduce_mean` or `np.mean`.

... standard layer arguments.
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.

Usage

layer_global_max_pooling_1d(
    object,
    data_format = "channels_last",
    keepdims = FALSE,
    ...
)

Arguments

- object: What to compose the new Layer instance with. Typically a Sequential model or a Tensor (e.g., as returned by layer_input()). The return value depends on object. If object is:
  - missing or NULL, the Layer instance is returned.
  - a Sequential model, the model with an additional layer is returned.
  - a Tensor, the output tensor from layer_instance(object) is returned.
- data_format: One of channels_last (default) or channels_first. The ordering of the dimensions in the inputs.
- keepdims: A boolean, whether to keep the spatial dimensions or not. If keepdims is FALSE (default), the rank of the tensor is reduced for spatial dimensions. If keepdims is TRUE, the spatial dimensions are retained with length 1. The behavior is the same as for tf.reduce_mean or np.mean.
- ...: standard layer arguments.
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.

Usage

layer_global_max_pooling_2d(object, data_format = NULL, keepdims = FALSE, ...)

Arguments

object

What to compose the new Layer instance with. Typically a Sequential model or a Tensor (e.g., as returned by layer_input()). The return value depends on object: If object is:

- missing or NULL, the Layer instance is returned.
- a Sequential model, the model with an additional layer is returned.
- a Tensor, the output tensor from layer_instance(object) is returned.

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".

keepdims

A boolean, whether to keep the spatial dimensions or not. If keepdims is FALSE (default), the rank of the tensor is reduced for spatial dimensions. If keepdims is TRUE, the spatial dimensions are retained with length 1. The behavior is the same as for tf.reduce_mean or np.mean.

... standard layer arguments.
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()

---

layer_global_max_pooling_3d

Global Max pooling operation for 3D data.

Description

Global Max pooling operation for 3D data.

Usage

`layer_global_max_pooling_3d(object, data_format = NULL, keepdims = FALSE, ...)`

Arguments

- **object**: What to compose the new Layer instance with. Typically a Sequential model or a Tensor (e.g., as returned by `layer_input()`). The return value depends on `object`. If `object` is:
  - missing or NULL, the Layer instance is returned.
  - a Sequential model, the model with an additional layer is returned.
  - a Tensor, the output tensor from `layer_instance(object)` is returned.

- **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".

- **keepdims**: A boolean, whether to keep the spatial dimensions or not. If `keepdims` is FALSE (default), the rank of the tensor is reduced for spatial dimensions. If `keepdims` is TRUE, the spatial dimensions are retained with length 1. The behavior is the same as for `tf.reduce_mean` or `np.mean`.

... standard layer arguments.
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

```r
layer_gru(
  object,
  units,
  activation = "tanh",
  recurrent_activation = "sigmoid",
  use_bias = TRUE,
  return_sequences = FALSE,
  return_state = FALSE,
  go_backwards = FALSE,
  stateful = FALSE,
  unroll = FALSE,
  time_major = FALSE,
  reset_after = TRUE,
  kernel_initializer = "glorot_uniform",
  recurrent_initializer = "orthogonal",
  bias_initializer = "zeros",
  kernel_regularizer = NULL,
  recurrent_regularizer = NULL,
  bias_regularizer = NULL,
  activity_regularizer = NULL,
```
layer_gru

kernel_constraint = NULL,
recurrent_constraint = NULL,
bias_constraint = NULL,
dropout = 0,
recurrent_dropout = 0,
...
)

Arguments

- **object**: What to compose the new Layer instance with. Typically a Sequential model or a Tensor (e.g., as returned by `layer_input()`). The return value depends on `object`. If `object` is:
  - missing or NULL, the Layer instance is returned.
  - a Sequential model, the model with an additional layer is returned.
  - a Tensor, the output tensor from `layer_instance(object)` is returned.

- **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 \([\text{timesteps}, \text{batch}, \text{feature}]\), whereas in the False case, it will be \([\text{batch}, \text{timesteps}, \text{feature}]\). 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**: 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.

... Standard Layer args.

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
N-D tensor with shape (batch_size, timesteps, ...), or (timesteps, batch_size, ...) when time_major = TRUE.

Output shape
• if return_state: a list of tensors. The first tensor is the output. The remaining tensors are the last states, each with shape (batch_size, state_size), where state_size could be a high dimension tensor shape.
• if return_sequences: N-D tensor with shape [batch_size, timesteps, output_size], where output_size could be a high dimension tensor shape, or [timesteps, batch_size, output_size] when time_major is TRUE
• else, N-D tensor with shape [batch_size, output_size], where output_size could be a high dimension tensor shape.
Masking

This layer supports masking for input data with a variable number of timesteps. To introduce masks to your data, use `layer_embedding()` 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 = list(...)` to the first layer in your model. For functional models with 1 or more Input layers, pass `batch_shape = list(...)` to all the first layers in your model. This is the expected shape of your inputs including the batch size. It should be a list of integers, e.g. `list(32, 10, 100)`. For dimensions which can vary (are not known ahead of time), use NULL in place of an integer, e.g. `list(32, NULL, NULL)`.
- Specify `shuffle = FALSE` when calling `fit()`.

To reset the states of your model, call `layer$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 named argument `states`. The value of `states` should be an array or list of arrays representing the initial state of the RNN layer.

Passing external constants to RNNs

You can pass "external" constants to the cell using the `constants` named argument of `RNN$call` (as well as `RNN$call`) method. This requires that the `cell$call` method accepts the same keyword argument constants. Such constants can be used to condition the cell transformation on additional static inputs (not changing over time), a.k.a. an attention mechanism.

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*
layer_gru_cell

Cell class for the GRU layer

Description
Cell class for the GRU layer

Usage

```r
layer_gru_cell(
    units,
    activation = "tanh",
    recurrent_activation = "sigmoid",
    use_bias = TRUE,
    kernel_initializer = "glorot_uniform",
    recurrent_initializer = "orthogonal",
    bias_initializer = "zeros",
    kernel_regularizer = NULL,
    recurrent_regularizer = NULL,
    bias_regularizer = NULL,
    kernel_constraint = NULL,
    recurrent_constraint = NULL,
    bias_constraint = NULL,
    dropout = 0,
    recurrent_dropout = 0,
    reset_after = TRUE,
    ...
)
```

Arguments

- **units**: Positive integer, dimensionality of the output space.
- **activation**: Activation function to use. Default: hyperbolic tangent (tanh). If you pass NULL, no activation is applied (ie. "linear" activation: \( a(x) = x \)).
- **recurrent_activation**: Activation function to use for the recurrent step. Default: sigmoid (sigmoid). If you pass NULL, no activation is applied (ie. "linear" activation: \( a(x) = x \)).
- **use_bias**: Boolean, (default TRUE), whether the layer uses a bias vector.
- **kernel_initializer**: Initializer for the kernel weights matrix, used for the linear transformation of the inputs. Default: glorot_uniform.

See Also

- [https://www.tensorflow.org/guide/keras/rnn](https://www.tensorflow.org/guide/keras/rnn)

Other recurrent layers: layer_cudnn_gru(), layer_cudnn_lstm(), layer_lstm(), layer_rnn(), layer_simple_rnn()
recurr**ent**_initializer
  **Initializer for the recurrent_kernel weights matrix, used for the linear transformation of the recurrent state. Default:** orthogonal.

bias_initializer
  **Initializer for the bias vector. Default:** zeros.

kernel_regularizer
  **Regularizer function applied to the kernel weights matrix. Default:** NULL.

recurrent_regularizer
  **Regularizer function applied to the recurrent_kernel weights matrix. Default:** NULL.

bias_regularizer
  **Regularizer function applied to the bias vector. Default:** NULL.

kernel_constraint
  **Constraint function applied to the kernel weights matrix. Default:** NULL.

recurrent_constraint
  **Constraint function applied to the recurrent_kernel weights matrix. Default:** NULL.

bias_constraint
  **Constraint function applied to the bias vector. Default:** NULL.

dropout
  Float between 0 and 1. Fraction of the units to drop for the linear transformation of the inputs. Default: 0.

recurrent_dropout
  Float between 0 and 1. Fraction of the units to drop for the linear transformation of the recurrent state. Default: 0.

reset_after
  **GRU convention (whether to apply reset gate after or before matrix multiplication).** FALSE = "before", TRUE = "after" (default and CuDNN compatible).

... standard layer arguments.

**Details**

See the Keras RNN API guide for details about the usage of RNN API.

This class processes one step within the whole time sequence input, whereas tf.keras.layer.GRU processes the whole sequence.

For example:

```r
inputs <- k_random_uniform(c(32, 10, 8))
output <- inputs %>% layer_rnn(layer_gru_cell(4))
output$shape # TensorShape([32, 4])

rnn <- layer_rnn(cell = layer_gru_cell(4),
                 return_sequence = TRUE,
                 return_state = TRUE)
c(whole_sequence_output, final_state) %<-% rnn(inputs)
whole_sequence_output$shape # TensorShape([32, 10, 4])
final_state$shape # TensorShape([32, 4])
```
See Also

- https://www.tensorflow.org/api_docs/python/tf/keras/layers/GRUCell

Other RNN cell layers: layer_lstm_cell(), layer_simple_rnn_cell(), layer_stacked_rnn_cells()

---

layer_hashing

A preprocessing layer which hashes and bins categorical features.

**Description**

A preprocessing layer which hashes and bins categorical features.

**Usage**

layer_hashing(object, num_bins, mask_value = NULL, salt = NULL, ...)

**Arguments**

- object
  - What to compose the new Layer instance with. Typically a Sequential model or a Tensor (e.g., as returned by layer_input()). The return value depends on object. If object is:
    - missing or NULL, the Layer instance is returned.
    - a Sequential model, the model with an additional layer is returned.
    - a Tensor, the output tensor from layer_instance(object) is returned.
- num_bins
  - Number of hash bins. Note that this includes the mask_value bin, so the effective number of bins is \((num\_bins - 1)\) if mask_value is set.
- mask_value
  - A value that represents masked inputs, which are mapped to index 0. Defaults to NULL, meaning no mask term will be added and the hashing will start at index 0.
- salt
  - A single unsigned integer or NULL. If passed, the hash function used will be SipHash64, with these values used as an additional input (known as a "salt" in cryptography). These should be non-zero. Defaults to NULL (in that case, the FarmHash64 hash function is used). It also supports list of 2 unsigned integer numbers, see reference paper for details.
- ...
- standard layer arguments.

**Details**

This layer transforms single or multiple categorical inputs to hashed output. It converts a sequence of int or string to a sequence of int. The stable hash function uses tensorflow::ops::Fingerprint to produce the same output consistently across all platforms.

This layer uses FarmHash64 by default, which provides a consistent hashed output across different platforms and is stable across invocations, regardless of device and context, by mixing the input bits thoroughly.
If you want to obfuscate the hashed output, you can also pass a random salt argument in the constructor. In that case, the layer will use the SipHash64 hash function, with the salt value serving as additional input to the hash function.

**Example (FarmHash64)**

```r
layer <- layer_hashing(num_bins=3)
inp <- matrix(c('A', 'B', 'C', 'D', 'E'))
layer(inp)
# <tf.Tensor: shape=(5, 1), dtype=int64, numpy=
# array([[1],
#         [0],
#         [1],
#         [1],
#         [2]])>
```

**Example (FarmHash64) with a mask value**

```r
layer <- layer_hashing(num_bins=3, mask_value='')
inp <- matrix(c('A', 'B', 'C', 'D', 'E'))
layer(inp)
# <tf.Tensor: shape=(5, 1), dtype=int64, numpy=
# array([[1],
#         [0],
#         [2],
#         [2]])>
```

**Example (SipHash64)**

```r
layer <- layer_hashing(num_bins=3, salt=c(133, 137))
inp <- matrix(c('A', 'B', 'C', 'D', 'E'))
layer(inp)
# <tf.Tensor: shape=(5, 1), dtype=int64, numpy=
# array([[2],
#         [1],
#         [0],
#         [2]])>
```

**Example (Siphash64 with a single integer, same as salt=c(133, 137))**

```r
layer <- layer_hashing(num_bins=3, salt=133)
inp <- matrix(c('A', 'B', 'C', 'D', 'E'))
layer(inp)
# <tf.Tensor: shape=(5, 1), dtype=int64, numpy=
# array([[0],
#         [0],
#         [2],
#         [1],
#         [0]])>
```
See Also

- https://www.tensorflow.org/api_docs/python/tf/keras/layers/Hashing
- https://keras.io/api/layers/preprocessing_layers/categorical/hashing/

Other categorical features preprocessing layers:
layer_category_encoding(), layer_integer_lookup(), layer_string_lookup()

Other preprocessing layers:
layer_category_encoding(), layer_center_crop(), layer_discretization(),
layer_integer_lookup(), layer_normalization(), layer_random_brightness(), layer_random_contrast(),
layer_random_crop(), layer_random_flip(), layer_random_height(), layer_random_rotation(),
layer_random_translation(), layer_random_width(), layer_random_zoom(), layer_rescaling(),
layer_rescaling(), layer_string_lookup(), layer_text_vectorization()

layer_input | Input layer

Description

Layer to be used as an entry point into a graph.

Usage

layer_input(
  shape = NULL,
  batch_shape = NULL,
  name = NULL,
  dtype = NULL,
  sparse = FALSE,
  tensor = NULL,
  ragged = FALSE
)

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>shape</td>
<td>Shape, not including the batch size. For instance, shape=c(32) indicates that the expected input will be batches of 32-dimensional vectors.</td>
</tr>
<tr>
<td>batch_shape</td>
<td>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.</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>dtype</td>
<td>The data type expected by the input, as a string (float32, float64, int32...)</td>
</tr>
<tr>
<td>sparse</td>
<td>Boolean, whether the placeholder created is meant to be sparse.</td>
</tr>
<tr>
<td>tensor</td>
<td>Existing tensor to wrap into the Input layer. If set, the layer will not create a placeholder tensor.</td>
</tr>
</tbody>
</table>

layer_input
layer_integer_lookup

A preprocessing layer which maps integer features to contiguous ranges.

Description

A preprocessing layer which maps integer features to contiguous ranges.

Usage

```r
layer_integer_lookup(
  object,
  max_tokens = NULL,
  num_oov_indices = 1L,
  mask_token = NULL,
  oov_token = -1L,
  vocabulary = NULL,
  invert = FALSE,
  output_mode = "int",
  sparse = FALSE,
  pad_to_max_tokens = FALSE,
  ...)
```

Arguments

- `object` What to compose the new Layer instance with. Typically a Sequential model or a Tensor (e.g., as returned by `layer_input()`). The return value depends on `object`. If `object` is:
  - missing or NULL, the Layer instance is returned.
  - a Sequential model, the model with an additional layer is returned.
  - a Tensor, the output tensor from `layer_instance(object)` is returned.

- `max_tokens` 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.

- `num_oov_indices` 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()`
max_tokens
The maximum size of the vocabulary for this layer. If NULL, there is no cap on the size of the vocabulary. Note that this size includes the OOV and mask tokens. Default to NULL.

num_oov_indices
The number of out-of-vocabulary tokens to use. If this value is more than 1, OOV inputs are modulated to determine their OOV value. If this value is 0, OOV inputs will cause an error when calling the layer. Defaults to 1.

mask_token
An integer token that represents masked inputs. When output_mode is "int", the token is included in vocabulary and mapped to index 0. In other output modes, the token will not appear in the vocabulary and instances of the mask token in the input will be dropped. If set to NULL, no mask term will be added. Defaults to NULL.

oov_token
Only used when invert is TRUE. The token to return for OOV indices. Defaults to -1.

ead_ory
Optional. Either an array of integers or a string path to a text file. If passing an array, can pass a list, list, 1D numpy array, or 1D tensor containing the integer vocabulary terms. If passing a file path, the file should contain one line per term in the vocabulary. If this argument is set, there is no need to adapt the layer.

invert
Only valid when output_mode is "int". If TRUE, this layer will map indices to vocabulary items instead of mapping vocabulary items to indices. Default to FALSE.

output_mode
Specification for the output of the layer. Defaults to "int". Values can be "int", "one_hot", "multi_hot", "count", or "tf_idf" configuring the layer as follows:

- "int": Return the vocabulary indices of the input tokens.
- "one_hot": Encodes each individual element in the input into an array the same size as the vocabulary, containing a 1 at the element index. If the last dimension is size 1, will encode on that dimension. If the last dimension is not size 1, will append a new dimension for the encoded output.
- "multi_hot": Encodes each sample in the input into a single array the same size as the vocabulary, containing a 1 for each vocabulary term present in the sample. Treats the last dimension as the sample dimension, if input shape is (... , sample_length), output shape will be (... , num_tokens).
- "count": As "multi_hot", but the int array contains a count of the number of times the token at that index appeared in the sample.
- "tf_idf": As "multi_hot", but the TF-IDF algorithm is applied to find the value in each token slot. For "int" output, any shape of input and output is supported. For all other output modes, currently only output up to rank 2 is supported.

sparse
Boolean. Only applicable when output_mode is "multi_hot", "count", or "tf_idf". If TRUE, returns a SparseTensor instead of a dense Tensor. Defaults to FALSE.

pad_to_max_tokens
Only applicable when output_mode is "multi_hot", "count", or "tf_idf". If TRUE, the output will have its feature axis padded to max_tokens even if the number of unique tokens in the vocabulary is less than max_tokens, resulting in
a tensor of shape [batch_size, max_tokens] regardless of vocabulary size. Defaults to FALSE.

... standard layer arguments.

Details

This layer maps a set of arbitrary integer input tokens into indexed integer output via a table-based vocabulary lookup. The layer’s output indices will be contiguously arranged up to the maximum vocab size, even if the input tokens are non-contiguous or unbounded. The layer supports multiple options for encoding the output via output_mode, and has optional support for out-of-vocabulary (OOV) tokens and masking.

The vocabulary for the layer can be supplied on construction or learned via adapt(). During adapt(), the layer will analyze a data set, determine the frequency of individual integer tokens, and create a vocabulary from them. If the vocabulary is capped in size, the most frequent tokens will be used to create the vocabulary and all others will be treated as OOV.

There are two possible output modes for the layer. When output_mode is "int", input integers are converted to their index in the vocabulary (an integer). When output_mode is "multi_hot", "count", or "tf_idf", input integers are encoded into an array where each dimension corresponds to an element in the vocabulary.

The vocabulary for the layer must be either supplied on construction or learned via adapt(). During adapt(), the layer will analyze a data set, determine the frequency of individual integer tokens, and create a vocabulary from them. If the vocabulary is capped in size, the most frequent tokens will be used to create the vocabulary and all others will be treated as OOV.

See Also

- adapt()
- https://www.tensorflow.org/api_docs/python/tf/keras/layers/IntegerLookup
- https://keras.io/api/layers/preprocessing_layers/categorical/integer_lookup

Other categorical features preprocessing layers: layer_category_encoding(), layer_hashing(), layer_string_lookup() 

Other preprocessing layers: layer_category_encoding(), layer_center_crop(), layer_discretization(), layer_hashing(), layer_normalization(), layer_random_brightness(), layer_random_contrast(), layer_random_crop(), layer_random_flip(), layer_random_height(), layer_random_rotation(), layer_random_translation(), layer_random_width(), layer_random_zoom(), layer_rescaling(), layer_resizing(), layer_string_lookup(), layer_text_vectorization()
Usage

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

object What to compose the new Layer instance with. Typically a Sequential model or a Tensor (e.g., as returned by layer_input()). The return value depends on object. If object is:
  • missing or NULL, the Layer instance is returned.
  • a Sequential model, the model with an additional layer is returned.
  • a Tensor, the output tensor from layer_instance(object) is returned.
f The function to be evaluated. Takes input tensor as first argument.
output_shape Expected output shape from the function (not required when using TensorFlow back-end).
mask
arguments optional named list of keyword arguments to be passed to the function.
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 normalization layer \cite{ba2016layer}.

**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**

```r
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**: What to compose the new Layer instance with. Typically a Sequential model or a Tensor (e.g., as returned by `layer_input()`). The return value depends on `object`. If `object` is:

- **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_regularization()`, `layer_attention()`, `layer_dense_features()`, `layer_dense()`, `layer_dropout()`, `layer_flatten()`, `layer_input()`, `layer_masking()`, `layer_permute()`, `layer_repeat_vector()`, `layer_reshape()`
• missing or NULL, the Layer instance is returned.
• a Sequential model, the model with an additional layer is returned.
• a Tensor, the output tensor from layer_instance(object) is returned.

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

Usage

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 What to compose the new Layer instance with. Typically a Sequential model or a Tensor (e.g., as returned by layer_input()). The return value depends on object. If object is:
  • missing or NULL, the Layer instance is returned.
  • a Sequential model, the model with an additional layer is returned.
  • a Tensor, the output tensor from layer_instance(object) is returned.

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 convolution. 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 (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

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.
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 unshared, 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 What to compose the new Layer instance with. Typically a Sequential model or a Tensor (e.g., as returned by layer_input()). The return value depends on object. If object is:
  • missing or NULL, the Layer instance is returned.
  • a Sequential model, the model with an additional layer is returned.
  • a Tensor, the output tensor from layer_instance(object) is returned.

See Also

Other locally connected layers: layer_locally_connected_2d()
filters
An 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 (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.

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),
layer_lstm

(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 Long Short-Term Memory unit - Hochreiter 1997.

Description

For a step-by-step description of the algorithm, see this tutorial.

Usage

layer_lstm(
    object,
    units,
    activation = "tanh",
    recurrent_activation = "sigmoid",
    use_bias = TRUE,
    return_sequences = FALSE,
    return_state = FALSE,
    go_backwards = FALSE,
    stateful = FALSE,
    time_major = FALSE,
unroll = FALSE,
kernel_initializer = "glorot_uniform",
recurrent_initializer = "orthogonal",
bias_initializer = "zeros",
unit_forget_bias = TRUE,
kernel_regularizer = NULL,
recurrent_regularizer = NULL,
bias_regularizer = NULL,
activity_regularizer = NULL,
kernel_constraint = NULL,
recurrent_constraint = NULL,
bias_constraint = NULL,
dropout = 0,
recurrent_dropout = 0,
...)

Arguments

object What to compose the new Layer instance with. Typically a Sequential model
or a Tensor (e.g., as returned by layer_input()). The return value depends on
object. If object is:
• missing or NULL, the Layer instance is returned.
• a Sequential model, the model with an additional layer is returned.
• a Tensor, the output tensor from layer_instance(object) is returned.

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.

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.
**layer_lstm**

**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.

**...**
- Standard Layer args.

**Input shapes**

N-D tensor with shape (batch_size, timesteps, ...), or (timesteps, batch_size, ...) when time_major = TRUE.

**Output shape**

- if return_state: a list of tensors. The first tensor is the output. The remaining tensors are the last states, each with shape (batch_size, state_size), where state_size could be a high dimension tensor shape.
layer_lstm

- if `return_sequences`: N-D tensor with shape `[batch_size, timesteps, output_size]`, where output_size could be a high dimension tensor shape, or `[timesteps, batch_size, output_size]` when `time_major` is TRUE
- else, N-D tensor with shape `[batch_size, output_size]`, where output_size could be a high dimension tensor shape.

**Masking**

This layer supports masking for input data with a variable number of timesteps. To introduce masks to your data, use `layer_embedding()` 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 = list(...)` to the first layer in your model. For functional models with 1 or more Input layers, pass `batch_shape = list(...)` to all the first layers in your model. This is the expected shape of your inputs including the batch size. It should be a list of integers, e.g. `list(32, 10, 100)`. For dimensions which can vary (are not known ahead of time), use NULL in place of an integer, e.g. `list(32, NULL, NULL).
- Specify `shuffle = FALSE` when calling `fit()`.

To reset the states of your model, call `layer$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 named argument `states`. The value of `states` should be an array or list of arrays representing the initial state of the RNN layer.

**Passing external constants to RNNs**

You can pass "external" constants to the cell using the constants named argument of `RNN$__call__` (as well as `RNN$call`) method. This requires that the cell$call method accepts the same keyword argument constants. Such constants can be used to condition the cell transformation on additional static inputs (not changing over time), a.k.a. an attention mechanism.
layer_lstm_cell

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

- https://www.tensorflow.org/guide/keras/rnn

Other recurrent layers: layer_cudnn_gru(), layer_cudnn_lstm(), layer_gru(), layer_rnn(), layer_simple_rnn()

Other recurrent layers: layer_cudnn_gru(), layer_cudnn_lstm(), layer_gru(), layer_rnn(), layer_simple_rnn()

layer_lstm_cell  

Cell class for the LSTM layer

Description

Cell class for the LSTM layer

Usage

```r
layer_lstm_cell(
  units,
  activation = "tanh",
  recurrent_activation = "sigmoid",
  use_bias = TRUE,
  kernel_initializer = "glorot_uniform",
  recurrent_initializer = "orthogonal",
  bias_initializer = "zeros",
  unit_forget_bias = TRUE,
  kernel_regularizer = NULL,
  recurrent_regularizer = NULL,
  bias_regularizer = NULL,
  kernel_constraint = NULL,
  recurrent_constraint = NULL,
  bias_constraint = NULL,
  dropout = 0,
  recurrent_dropout = 0,
  ...
)
```
Arguments

- **units**: Positive integer, dimensionality of the output space.
- **activation**: Activation function to use. Default: hyperbolic tangent (tanh). If you pass NULL, no activation is applied (ie. "linear" activation: $a(x) = x$).
- **recurrent_activation**: Activation function to use for the recurrent step. Default: sigmoid (sigmoid). If you pass NULL, no activation is applied (ie. "linear" activation: $a(x) = x$).
- **use_bias**: Boolean, (default TRUE), whether the layer uses a bias vector.
- **kernel_initializer**: Initializer for the kernel weights matrix, used for the linear transformation of the inputs. Default: glorot_uniform.
- **recurrent_initializer**: Initializer for the recurrent kernel weights matrix, used for the linear transformation of the recurrent state. Default: orthogonal.
- **bias_initializer**: Initializer for the bias vector. Default: zeros.
- **unit_forget_bias**: Boolean (default TRUE). If TRUE, add 1 to the bias of the forget gate at initialization. Setting it to true will also force bias_initializer="zeros". This is recommended in Jozefowicz et al.
- **kernel_regularizer**: Regularizer function applied to the kernel weights matrix. Default: NULL.
- **recurrent_regularizer**: Regularizer function applied to the recurrent_kernel weights matrix. Default: NULL.
- **bias_regularizer**: Regularizer function applied to the bias vector. Default: NULL.
- **kernel_constraint**: Constraint function applied to the kernel weights matrix. Default: NULL.
- **recurrent_constraint**: Constraint function applied to the recurrent_kernel weights matrix. Default: NULL.
- **bias_constraint**: Constraint function applied to the bias vector. Default: NULL.
- **dropout**: Float between 0 and 1. Fraction of the units to drop for the linear transformation of the inputs. Default: 0.
- **recurrent_dropout**: Float between 0 and 1. Fraction of the units to drop for the linear transformation of the recurrent state. Default: 0.

... standard layer arguments.

Details

See the Keras RNN API guide for details about the usage of RNN API.
This class processes one step within the whole time sequence input, whereas `tf$keras$layer$LSTM` processes the whole sequence.

For example:

```r
inputs <- k_random_normal(c(32, 10, 8))
rnn <- layer_rnn(cell = layer_lstm_cell(units = 4))
output <- rnn(inputs)
dim(output) # (32, 4)

rnn <- layer_rnn(cell = layer_lstm_cell(units = 4),
                  return_sequences = TRUE,
                  return_state = TRUE)
c(whole_seq_output, final_memory_state, final_carry_state) %<-% rnn(inputs)

dim(whole_seq_output) # (32, 10, 4)
dim(final_memory_state) # (32, 4)
dim(final_carry_state) # (32, 4)
```

See Also

- [https://www.tensorflow.org/api_docs/python/tf/keras/layers/LSTMCell](https://www.tensorflow.org/api_docs/python/tf/keras/layers/LSTMCell)

Other RNN cell layers: `layer_gru_cell()`, `layer_simple_rnn_cell()`, `layer_stacked_rnn_cells()`

---

**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**

```r
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**

- **object**: What to compose the new `Layer` instance with. Typically a Sequential model or a Tensor (e.g., as returned by `layer_input()`). The return value depends on `object`. If `object` is:
  - missing or NULL, the `Layer` instance is returned.
  - a Sequential model, the model with an additional layer is returned.
  - a Tensor, the output tensor from `layer_instance(object)` is returned.

- **mask_value**: float, mask value

- **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 core layers: `layer_activation()`, `layer_activity_regularizer()`, `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**

`layer_maximum(inputs, ...)`

**Arguments**

- **inputs**: A list of input tensors (at least 2). Can be missing.

  ... Standard layer arguments (must be named).
layer_max_pooling_1d

Value
A tensor, the element-wise maximum of the inputs. If inputs is missing, a keras layer instance is returned.

See Also
- https://www.tensorflow.org/api_docs/python/tf/keras/layers/maximum
- https://www.tensorflow.org/api_docs/python/tf/keras/layers/Maximum
- https://keras.io/api/layers/merging_layers/maximum

Other merge layers: 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
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: What to compose the new Layer instance with. Typically a Sequential model or a Tensor (e.g., as returned by layer_input()). The return value depends on object. If object is:
  - missing or NULL, the Layer instance is returned.
  - a Sequential model, the model with an additional layer is returned.
  - a Tensor, the output tensor from layer_instance(object) is returned.
- 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.
layer_max_pooling_2d

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).

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

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**
What to compose the new Layer instance with. Typically a Sequential model or a Tensor (e.g., as returned by `layer_input()`). The return value depends on object. If object is:

- missing or NULL, the Layer instance is returned.
- a Sequential model, the model with an additional layer is returned.
- a Tensor, the output tensor from `layer_instance(object)` is returned.

**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".

**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**

```r
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
)
```

**Arguments**

- `object` What to compose the new Layer instance with. Typically a Sequential model or a Tensor (e.g., as returned by `layer_input()`). The return value depends on `object`. If `object` is:
  - missing or `NULL`, the Layer instance is returned.
  - a `Sequential` model, the model with an additional layer is returned.
  - a Tensor, the output tensor from `layer_instance(object)` is returned.
- `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_minimum

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

- `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, ...)`

Arguments

- `inputs` A list of input tensors (at least 2). Can be missing.
- `...` Standard layer arguments (must be named).

Value

A tensor, the element-wise maximum of the inputs. If `inputs` is missing, a keras layer instance is returned.

See Also

- https://www.tensorflow.org/api_docs/python/tf/keras/layers/minimum
- https://www.tensorflow.org/api_docs/python/tf/keras/layers/Minimum
- https://keras.io/api/layers/merging_layers/minimum

Other merge layers: `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, ...)
```

**Arguments**

- **inputs**: A list of input tensors (at least 2). Can be missing.
- **...**: Standard layer arguments (must be named).

**Value**

A tensor, the element-wise product of the inputs. If `inputs` is missing, a keras layer instance is returned.

**See Also**

- [https://www.tensorflow.org/api_docs/python/tf/keras/layers/multiply](https://www.tensorflow.org/api_docs/python/tf/keras/layers/multiply)
- [https://www.tensorflow.org/api_docs/python/tf/keras/layers/Multiply](https://www.tensorflow.org/api_docs/python/tf/keras/layers/Multiply)
- [https://keras.io/api/layers/merging_layers/multiply](https://keras.io/api/layers/merging_layers/multiply)

Other merge layers: `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.
layer_multi_head_attention

Usage

layer_multi_head_attention(
  inputs,
  num_heads,
  key_dim,
  value_dim = NULL,
  dropout = 0,
  use_bias = TRUE,
  output_shape = NULL,
  attention_axes = NULL,
  kernel_initializer = "glorot_uniform",
  bias_initializer = "zeros",
  kernel_regularizer = NULL,
  bias_regularizer = NULL,
  activity_regularizer = NULL,
  kernel_constraint = NULL,
  bias_constraint = NULL,
  ...
)

Arguments

inputs          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.
kernellnitializer  Initialization for dense layer kernels.
bias_initializer  Initialization for dense layer biases.
kernellnitializer  Regularizer for dense layer kernels.
bias_regularizer   Regularizer for dense layer biases.
activity_regularizer   Regularizer for dense layer activity.
kernellnitializer   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 coeffients 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_normalization

A preprocessing layer which normalizes continuous features.

Description

A preprocessing layer which normalizes continuous features.

Usage

layer_normalization(object, axis = -1L, mean = NULL, variance = NULL, ...)
layer_normalization

Arguments

object  What to compose the new Layer instance with. Typically a Sequential model or a Tensor (e.g., as returned by `layer_input()`). The return value depends on object. If object is:
  • missing or NULL, the Layer instance is returned.
  • a Sequential model, the model with an additional layer is returned.
  • a Tensor, the output tensor from `layer_instance(object)` is returned.

axis  Integer, list of integers, or NULL. The axis or axes that should have a separate mean and variance for each index in the shape. For example, if shape is (NULL, 5) and axis=1, the layer will track 5 separate mean and variance values for the last axis. If axis is set to NULL, the layer will normalize all elements in the input by a scalar mean and variance. Defaults to -1, where the last axis of the input is assumed to be a feature dimension and is normalized per index. Note that in the specific case of batched scalar inputs where the only axis is the batch axis, the default will normalize each index in the batch separately. In this case, consider passing axis = NULL.

mean  The mean value(s) to use during normalization. The passed value(s) will be broadcast to the shape of the kept axes above; if the value(s) cannot be broadcast, an error will be raised when this layer’s build() method is called.

variance  The variance value(s) to use during normalization. The passed value(s) will be broadcast to the shape of the kept axes above; if the value(s) cannot be broadcast, an error will be raised when this layer’s build() method is called.

...  standard layer arguments.

Details

This layer will shift and scale inputs into a distribution centered around 0 with standard deviation 1. It accomplishes this by precomputing the mean and variance of the data, and calling (input - mean) / sqrt(var) at runtime.

The mean and variance values for the layer must be either supplied on construction or learned via adapt(). adapt() will compute the mean and variance of the data and store them as the layer’s weights. adapt() should be called before fit(), evaluate(), or predict().

See Also

• adapt()
• https://www.tensorflow.org/api_docs/python/tf/keras/layers/Normalization
• https://keras.io/api/layers/preprocessing_layers/numerical/normalization

Other numerical features preprocessing layers: layer_discretization()

Other preprocessing layers: layer_category_encoding(), layer_center_crop(), layer_discretization(), layer_hashing(), layer_integer_lookup(), layer_random_brightness(), layer_random_contrast(), layer_random_crop(), layer_random_flip(), layer_random_height(), layer_random_rotation(), layer_random_translation(), layer_random_width(), layer_random_zoom(), layer_rescaling(), layer_resizing(), layer_string_lookup(), layer_text_vectorization()
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  What to compose the new Layer instance with. Typically a Sequential model or a Tensor (e.g., as returned by layer_input()). The return value depends on object. If object is:
• missing or NULL, the Layer instance is returned.
• a Sequential model, the model with an additional layer is returned.
• a Tensor, the output tensor from layer_instance(object) is returned.

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_random_brightness

A preprocessing layer which randomly adjusts brightness during training

Description
A preprocessing layer which randomly adjusts brightness during training

Usage

```r
layer_random_brightness(
  object,
  factor,
  value_range = c(0, 255),
  seed = NULL,
  ...
)
```

Arguments

- **object**: What to compose the new Layer instance with. Typically a Sequential model or a Tensor (e.g., as returned by `layer_input()`). The return value depends on object. If object is:
  - missing or NULL, the Layer instance is returned.
  - a Sequential model, the model with an additional layer is returned.
  - a Tensor, the output tensor from `layer_instance(object)` is returned.
layer_random_contrast

factor

Float or a list of 2 floats between -1.0 and 1.0. The factor is used to determine the lower bound and upper bound of the brightness adjustment. A float value will be chosen randomly between the limits. When -1.0 is chosen, the output image will be black, and when 1.0 is chosen, the image will be fully white. When only one float is provided, eg, 0.2, then -0.2 will be used for lower bound and 0.2 will be used for upper bound.

value_range

Optional list of 2 floats for the lower and upper limit of the values of the input data. Defaults to [0.0, 255.0]. Can be changed to e.g. [0.0, 1.0] if the image input has been scaled before this layer. The brightness adjustment will be scaled to this range, and the output values will be clipped to this range.

seed

optional integer, for fixed RNG behavior.

...  

standard layer arguments.

Details

This layer will randomly increase/reduce the brightness for the input RGB images. At inference time, the output will be identical to the input. Call the layer with training=TRUE to adjust the brightness of the input.

Note that different brightness adjustment factors will be apply to each the images in the batch.

For an overview and full list of preprocessing layers, see the preprocessing guide.

See Also

- https://www.tensorflow.org/api_docs/python/tf/keras/layers/RandomBrightness
- https://keras.io/api/layers

Other image augmentation layers: layer_random_contrast(), layer_random_crop(), layer_random_flip(), layer_random_height(), layer_random_rotation(), layer_random_translation(), layer_random_width(), layer_random_zoom()

Other preprocessing layers: layer_category_encoding(), layer_center_crop(), layer_discretization(), layer_hashing(), layer_integer_lookup(), layer_normalization(), layer_random_contrast(), layer_random_crop(), layer_random_flip(), layer_random_height(), layer_random_rotation(), layer_random_translation(), layer_random_width(), layer_random_zoom(), layer_rescaling(), layer_resizing(), layer_string_lookup(), layer_text_vectorization()

layer_random_contrast  Adjust the contrast of an image or images by a random factor

Description

Adjust the contrast of an image or images by a random factor

Usage

layer_random_contrast(object, factor, seed = NULL, ...)
**layer_random_crop**

**Arguments**

- **object**: What to compose the new Layer instance with. Typically a Sequential model or a Tensor (e.g., as returned by `layer_input()`). The return value depends on object. If object is:
  - missing or NULL, the Layer instance is returned.
  - a Sequential model, the model with an additional layer is returned.
  - a Tensor, the output tensor from `layer_instance(object)` is returned.

- **factor**: a positive float represented as fraction of value, or a list of size 2 representing lower and upper bound. When represented as a single float, lower = upper. The contrast factor will be randomly picked between \([1.0 - \text{lower}, 1.0 + \text{upper}]\).

- **seed**: Integer. Used to create a random seed.

... standard layer arguments.

**Details**

Contrast is adjusted independently for each channel of each image during training.

For each channel, this layer computes the mean of the image pixels in the channel and then adjusts each component \(x\) of each pixel to \((x - \text{mean}) \times \text{contrast_factor} + \text{mean}\).

Input shape: 3D (unbatched) or 4D (batched) tensor with shape: \((..., \text{height}, \text{width}, \text{channels})\), in "channels_last" format.

Output shape: 3D (unbatched) or 4D (batched) tensor with shape: \((..., \text{height}, \text{width}, \text{channels})\), in "channels_last" format.

**See Also**

- [https://www.tensorflow.org/api_docs/python/tf/keras/layers/RandomContrast](https://www.tensorflow.org/api_docs/python/tf/keras/layers/RandomContrast)
- [https://keras.io/api/layers/preprocessing_layers/](https://keras.io/api/layers/preprocessing_layers/)

Other image augmentation layers: `layer_random_brightness()`, `layer_random_crop()`, `layer_random_flip()`, `layer_random_height()`, `layer_random_rotation()`, `layer_random_translation()`, `layer_random_width()`, `layer_random_zoom()`

Other preprocessing layers: `layer_category_encoding()`, `layer_center_crop()`, `layer_discretization()`, `layer_hashing()`, `layer_integer_lookup()`, `layer_normalization()`, `layer_random_brightness()`, `layer_random_crop()`, `layer_random_flip()`, `layer_random_height()`, `layer_random_rotation()`, `layer_random_translation()`, `layer_random_width()`, `layer_random_zoom()`, `layer_rescaling()`, `layer_resizing()`, `layer_string_lookup()`, `layer_text_vectorization()`

---

**Description**

Randomly crop the images to target height and width.
Usage

layer_random_crop(object, height, width, seed = NULL, ...)

Arguments

object What to compose the new Layer instance with. Typically a Sequential model or a Tensor (e.g., as returned by layer_input()). The return value depends on object. If object is:
  • missing or NULL, the Layer instance is returned.
  • a Sequential model, the model with an additional layer is returned.
  • a Tensor, the output tensor from layer_instance(object) is returned.
height Integer, the height of the output shape.
width Integer, the width of the output shape.
seed Integer. Used to create a random seed.
... standard layer arguments.

Details

This layer will crop all the images in the same batch to the same cropping location. By default, random cropping is only applied during training. At inference time, the images will be first rescaled to preserve the shorter side, and center cropped. If you need to apply random cropping at inference time, set training to TRUE when calling the layer.

Input shape: 3D (unbatched) or 4D (batched) tensor with shape: (... , height, width, channels), in "channels_last" format.

Output shape: 3D (unbatched) or 4D (batched) tensor with shape: (... , target_height, target_width, channels).

See Also

- https://www.tensorflow.org/api_docs/python/tf/keras/layers/RandomCrop
- https://keras.io/api/layers/preprocessing_layers/image_augmentation/random_crop

Other image augmentation layers: layer_random_brightness(), layer_random_contrast(), layer_random_flip(), layer_random_height(), layer_random_rotation(), layer_random_translation(), layer_random_width(), layer_random_zoom()

Other preprocessing layers: layer_category_encoding(), layer_center_crop(), layer_discretization(), layer_hashing(), layer_integer_lookup(), layer_normalization(), layer_random_brightness(), layer_random_contrast(), layer_random_flip(), layer_random_height(), layer_random_rotation(), layer_random_translation(), layer_random_width(), layer_random_zoom(), layer_rescaling(), layer_resizing(), layer_string_lookup(), layer_text_vectorization()
layer_random_flip

Randomly flip each image horizontally and vertically

Description

Randomly flip each image horizontally and vertically

Usage

layer_random_flip(object, mode = "horizontal_and_vertical", seed = NULL, ...)

Arguments

- **object**: What to compose the new Layer instance with. Typically a Sequential model or a Tensor (e.g., as returned by `layer_input()`). The return value depends on `object`. If `object` is:
  - missing or NULL, the Layer instance is returned.
  - a Sequential model, the model with an additional layer is returned.
  - a Tensor, the output tensor from `layer_instance(object)` is returned.

- **mode**: String indicating which flip mode to use. Can be "horizontal", "vertical", or "horizontal_and_vertical". Defaults to "horizontal_and_vertical". "horizontal" is a left-right flip and "vertical" is a top-bottom flip.

- **seed**: Integer. Used to create a random seed.

- **...**: standard layer arguments.

Details

This layer will flip the images based on the `mode` attribute. During inference time, the output will be identical to input. Call the layer with `training = TRUE` to flip the input.

Input shape: 3D (unbatched) or 4D (batched) tensor with shape: (... height, width, channels), in "channels_last" format.

Output shape: 3D (unbatched) or 4D (batched) tensor with shape: (... height, width, channels), in "channels_last" format.

See Also

- https://www.tensorflow.org/api_docs/python/tf/keras/layers/RandomFlip
- https://keras.io/api/layers/preprocessing_layers/image_augmentation/random_flip

Other image augmentation layers: `layer_random_brightness()`, `layer_random_contrast()`, `layer_random_crop()`, `layer_random_height()`, `layer_random_rotation()`, `layer_random_translation()`, `layer_random_width()`, `layer_random_zoom()`

Other preprocessing layers: `layer_category_encoding()`, `layer_center_crop()`, `layer_discretization()`, `layer_hashing()`, `layer_integer_lookup()`, `layer_normalization()`, `layer_random_brightness()`,
layer_random_height

layer_random_contrast(), layer_random_crop(), layer_random_height(), layer_random_rotation(), layer_random_translation(), layer_random_width(), layer_random_zoom(), layer_rescaling(), layer_resizing(), layer_string_lookup(), layer_text_vectorization()

layer_random_height  Randomly vary the height of a batch of images during training

Description

Randomly vary the height of a batch of images during training

Usage

layer_random_height(
  object,  
factor,      
interpolation = "bilinear",    
seed = NULL,     
...    
)

Arguments

  object What to compose the new Layer instance with. Typically a Sequential model or a Tensor (e.g., as returned by layer_input()). The return value depends on object. If object is:
    • missing or NULL, the Layer instance is returned.
    • a Sequential model, the model with an additional layer is returned.
    • a Tensor, the output tensor from layer_instance(object) is returned.
  
factor A positive float (fraction of original height), or a list of size 2 representing lower and upper bound for resizing vertically. When represented as a single float, this value is used for both the upper and lower bound. For instance, factor = c(0.2, 0.3) results in an output with height changed by a random amount in the range [20%, 30%]. factor = c(-0.2, 0.3) results in an output with height changed by a random amount in the range [-20%, +30%]. factor = 0.2 results in an output with height changed by a random amount in the range [-20%, +20%].
  
interpolation String, the interpolation method. Defaults to "bilinear". Supports "bilinear", "nearest", "bicubic", "area", "lanczos3", "lanczos5", "gaussian", "mitchellcubic".

seed Integer. Used to create a random seed.

Details

Adjusts the height of a batch of images by a random factor. The input should be a 3D (unbatched) or 4D (batched) tensor in the "channels_last" image data format.

By default, this layer is inactive during inference.
**layer_random_rotation**  
Randomly rotate each image

### Description

Randomly rotate each image

### Usage

```r
layer_random_rotation(
  object,
  factor,
  fill_mode = "reflect",
  interpolation = "bilinear",
  seed = NULL,
  fill_value = 0,
  ...
)
```

### Arguments

- **object**: What to compose the new Layer instance with. Typically a Sequential model or a Tensor (e.g., as returned by `layer_input()`). The return value depends on `object`. If `object` is:
  - missing or NULL, the Layer instance is returned.
  - a Sequential model, the model with an additional layer is returned.
  - a Tensor, the output tensor from `layer_instance(object)` is returned.

- **factor**: a float represented as fraction of 2 Pi, or a list of size 2 representing lower and upper bound for rotating clockwise and counter-clockwise. A positive value means rotating counter-clockwise, while a negative value means clock-wise. When represented as a single float, this value is used for both the upper and lower bound. For instance, `factor = c(-0.2, 0.3)` results in an output rotation by a random amount in the range `[-20% * 2pi, 30% * 2pi]`. `factor = 0.2` results in an output rotating by a random amount in the range `[-20% * 2pi, 20% * 2pi]`. See Also

  - https://www.tensorflow.org/api_docs/python/tf/keras/layers/RandomHeight
  - https://keras.io/api/layers/preprocessing_layers/

Other image augmentation layers: `layer_random_brightness()`, `layer_random_contrast()`, `layer_random_crop()`, `layer_random_flip()`, `layer_random_rotation()`, `layer_random_translation()`, `layer_random_width()`, `layer_random_zoom()`

Other preprocessing layers: `layer_category_encoding()`, `layer_center_crop()`, `layer_discretization()`, `layer_hashing()`, `layer_integer_lookup()`, `layer_normalization()`, `layer_random_brightness()`, `layer_random_contrast()`, `layer_random_crop()`, `layer_random_flip()`, `layer_random_rotation()`, `layer_random_translation()`, `layer_random_width()`, `layer_random_zoom()`, `layer_rescaling()`, `layer_resizing()`, `layer_string_lookup()`, `layer_text_vectorization()`
layer_random_rotation

fill_mode Points outside the boundaries of the input are filled according to the given mode (one of \{"constant", "reflect", "wrap", "nearest"\}).

- \textbf{reflect}: (d c b a | a b c d | d c b a) The input is extended by reflecting about the edge of the last pixel.
- \textbf{constant}: (k k k k | a b c d | k k k k) The input is extended by filling all values beyond the edge with the same constant value k = 0.
- \textbf{wrap}: (a b c d | a b c d | a b c d) The input is extended by wrapping around to the opposite edge.
- \textbf{nearest}: (a a a a | a b c d | d d d d) The input is extended by the nearest pixel.

interpolation Interpolation mode. Supported values: \textit{"nearest", "bilinear"}.

seed Integer. Used to create a random seed.

fill_value a float represents the value to be filled outside the boundaries when fill_mode="constant".

... standard layer arguments.

Details

By default, random rotations are only applied during training. At inference time, the layer does nothing. If you need to apply random rotations at inference time, set \texttt{training} to \texttt{TRUE} when calling the layer.

Input shape: 3D (unbatched) or 4D (batched) tensor with shape: (..., height, width, channels), in \textit{"channels_last"} format.

Output shape: 3D (unbatched) or 4D (batched) tensor with shape: (..., height, width, channels), in \textit{"channels_last"} format.

See Also

- \url{https://www.tensorflow.org/api_docs/python/tf/keras/layers/RandomRotation}
- \url{https://keras.io/api/layers/preprocessing_layers/}

Other image augmentation layers: \texttt{layer_random_brightness()}, \texttt{layer_random_contrast()}, \texttt{layer_random_crop()}, \texttt{layer_random_flip()}, \texttt{layer_random_height()}, \texttt{layer_random_translation()}, \texttt{layer_random_width()}, \texttt{layer_random_zoom()}

Other preprocessing layers: \texttt{layer_category_encoding()}, \texttt{layer_center_crop()}, \texttt{layer_discretization()}, \texttt{layer_hashing()}, \texttt{layer_integer_lookup()}, \texttt{layer_normalization()}, \texttt{layer_random_brightness()}, \texttt{layer_random_contrast()}, \texttt{layer_random_crop()}, \texttt{layer_random_flip()}, \texttt{layer_random_height()}, \texttt{layer_random_translation()}, \texttt{layer_random_width()}, \texttt{layer_random_zoom()}, \texttt{layer_rescaling()}, \texttt{layer_resizing()}, \texttt{layer_string_lookup()}, \texttt{layer_text_vectorization()}
layer_random_translation

Randomly translate each image during training

Description

Randomly translate each image during training

Usage

layer_random_translation(
  object,
  height_factor,
  width_factor,
  fill_mode = "reflect",
  interpolation = "bilinear",
  seed = NULL,
  fill_value = 0,
  ...
)

Arguments

object What to compose the new Layer instance with. Typically a Sequential model or a Tensor (e.g., as returned by layer_input()). The return value depends on object. If object is:
  • missing or NULL, the Layer instance is returned.
  • a Sequential model, the model with an additional layer is returned.
  • a Tensor, the output tensor from layer_instance(object) is returned.

height_factor a float represented as fraction of value, or a list of size 2 representing lower and upper bound for shifting vertically. A negative value means shifting image up, while a positive value means shifting image down. When represented as a single positive float, this value is used for both the upper and lower bound. For instance, height_factor = c(-0.2, 0.3) results in an output shifted by a random amount in the range [-20%, +30%]. height_factor = 0.2 results in an output height shifted by a random amount in the range [-20%, +20%].

width_factor a float represented as fraction of value, or a list of size 2 representing lower and upper bound for shifting horizontally. A negative value means shifting image left, while a positive value means shifting image right. When represented as a single positive float, this value is used for both the upper and lower bound. For instance, width_factor = c(-0.2, 0.3) results in an output shifted left by 20%, and shifted right by 30%. width_factor = 0.2 results in an output height shifted left or right by 20%.

fill_mode Points outside the boundaries of the input are filled according to the given mode (one of {"constant", "reflect", "wrap", "nearest"}).
layer_random_width

Randomly vary the width of a batch of images during training

Description

Randomly vary the width of a batch of images during training

Usage

layer_random_width(
    object,
    factor,
    interpolation = "bilinear",
    seed = NULL,
    ...
)
Arguments

object       What to compose the new Layer instance with. Typically a Sequential model or a Tensor (e.g., as returned by layer_input()). The return value depends on object. If object is:
               • missing or NULL, the Layer instance is returned.
               • a Sequential model, the model with an additional layer is returned.
               • a Tensor, the output tensor from layer_instance(object) is returned.

factor      A positive float (fraction of original height), or a list of size 2 representing lower and upper bound for resizing vertically. When represented as a single float, this value is used for both the upper and lower bound. For instance, factor = c(0.2, 0.3) results in an output with width changed by a random amount in the range [20%, 30%]. factor=(-0.2, 0.3) results in an output with width changed by a random amount in the range [-20%, +30%]. factor = 0.2 results in an output with width changed by a random amount in the range [-20%, +20%].

interpolation  String, the interpolation method. Defaults to bicubic. Supports "bilinear", "nearest", "bicubic", "area", "lanczos3", "lanczos5", "gaussian", "mitchellcubic".

seed         Integer. Used to create a random seed.

...  standard layer arguments.

Details

Adjusts the width of a batch of images by a random factor. The input should be a 3D (unbatched) or 4D (batched) tensor in the "channels_last" image data format.

By default, this layer is inactive during inference.

See Also

• https://www.tensorflow.org/api_docs/python/tf/keras/layers/RandomWidth
• https://keras.io/api/layers/preprocessing_layers/

Other image augmentation layers: layer_random_brightness(), layer_random_contrast(), layer_random_crop(), layer_random_flip(), layer_random_height(), layer_random_rotation(), layer_random_translation(), layer_random_zoom()

Other preprocessing layers: layer_category_encoding(), layer_center_crop(), layer_discretization(), layer_hashing(), layer_integer_lookup(), layer_normalization(), layer_random_brightness(), layer_random_contrast(), layer_random_crop(), layer_random_flip(), layer_random_height(), layer_random_rotation(), layer_random_translation(), layer_random_zoom(), layer_rescaling(), layer_resizing(), layer_string_lookup(), layer_text_vectorization()
A preprocessing layer which randomly zooms images during training.

Description

This layer will randomly zoom in or out on each axis of an image independently, filling empty space according to fill_mode.

Usage

```r
layer_random_zoom(
  object,
  height_factor,
  width_factor = NULL,
  fill_mode = "reflect",
  interpolation = "bilinear",
  seed = NULL,
  fill_value = 0,
  ...
)
```

Arguments

- `object`: What to compose the new `Layer` instance with. Typically a `Sequential` model or a `Tensor` (e.g., as returned by `layer_input()`). The return value depends on `object`. If `object` is:
  - missing or `NULL`, the `Layer` instance is returned.
  - a `Sequential` model, the model with an additional layer is returned.
  - a `Tensor`, the output tensor from `layer_instance(object)` is returned.
- `height_factor`: a float represented as fraction of value, or a list of size 2 representing lower and upper bound for zooming vertically. When represented as a single float, this value is used for both the upper and lower bound. A positive value means zooming out, while a negative value means zooming in. For instance, `height_factor = c(0.2, 0.3)` result in an output zoomed out by a random amount in the range [+20%, +30%]. `height_factor = c(-0.3, -0.2)` result in an output zoomed in by a random amount in the range [+20%, +30%].
- `width_factor`: a float represented as fraction of value, or a list of size 2 representing lower and upper bound for zooming horizontally. When represented as a single float, this value is used for both the upper and lower bound. For instance, `width_factor = c(0.2, 0.3)` result in an output zooming out between 20% to 30%. `width_factor = c(-0.3, -0.2)` result in an output zooming in between 20% to 30%. Defaults to `NULL`, i.e., zooming vertical and horizontal directions by preserving the aspect ratio.
- `fill_mode`: Points outside the boundaries of the input are filled according to the given mode (one of {"constant", "reflect", "wrap", "nearest"}).
layer_repeat_vector

Repeats the input \( n \) times.

**Description**

Repeats the input \( n \) times.

**Usage**

```python
layer_repeat_vector(
    object,
    n,
    batch_size = NULL,
    name = NULL,
    trainable = NULL,
    weights = NULL
)
```
Arguments

- **object**: What to compose the new Layer instance with. Typically a Sequential model or a Tensor (e.g., as returned by `layer_input()`). The return value depends on `object`. If `object` is:
  - missing or NULL, the Layer instance is returned.
  - a Sequential model, the model with an additional layer is returned.
  - a Tensor, the output tensor from `layer_instance(object)` is returned.
- **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_rescaling**  
Multiply inputs by scale and adds offset

Description

Multiply inputs by scale and adds offset

Usage

`layer_rescaling(object, scale, offset = 0, ...)`
layer_reshape

Reshapes an output to a certain shape.

Description

Reshapes an output to a certain shape.

Arguments

- **object**: What to compose the new Layer instance with. Typically a Sequential model or a Tensor (e.g., as returned by `layer_input()`). The return value depends on `object`. If `object` is:
  - missing or NULL, the Layer instance is returned.
  - a Sequential model, the model with an additional layer is returned.
  - a Tensor, the output tensor from `layer_instance(object)` is returned.

- **scale**: Float, the scale to apply to the inputs.
- **offset**: Float, the offset to apply to the inputs.

... standard layer arguments.

Details

For instance:

1. To rescale an input in the \([0, 255]\) range to be in the \([0, 1]\) range, you would pass `scale=1./255`.
2. To rescale an input in the \([0, 255]\) range to be in the \([-1, 1]\) range, you would pass `scale = 1/127.5, offset = -1`.

The rescaling is applied both during training and inference.

Input shape: Arbitrary.

Output shape: Same as input.

See Also

- [https://www.tensorflow.org/api_docs/python/tf/keras/layers/Rescaling](https://www.tensorflow.org/api_docs/python/tf/keras/layers/Rescaling)
- [https://keras.io/api/layers/preprocessing_layers/image_preprocessing/rescaling](https://keras.io/api/layers/preprocessing_layers/image_preprocessing/rescaling)

Other image preprocessing layers: `layer_center_crop()`, `layer_resizing()`

Other preprocessing layers: `layer_category_encoding()`, `layer_center_crop()`, `layer_discretization()`, `layer_hashing()`, `layer_integer_lookup()`, `layer_normalization()`, `layer_random_brightness()`, `layer_random_contrast()`, `layer_random_crop()`, `layer_random_flip()`, `layer_random_height()`, `layer_random_rotation()`, `layer_random_translation()`, `layer_random_width()`, `layer_random_zoom()`, `layer_resizing()`, `layer_string_lookup()`, `layer_text_vectorization()`
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 What to compose the new Layer instance with. Typically a Sequential model or a Tensor (e.g., as returned by layer_input()). The return value depends on object. If object is:
  • missing or NULL, the Layer instance is returned.
  • a Sequential model, the model with an additional layer is returned.
  • a Tensor, the output tensor from layer_instance(object) is returned.
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()
layer_resizing

Image resizing layer

Description
Image resizing layer

Usage

```r
layer_resizing(
  object,  # What to compose the new Layer instance with. Typically a Sequential model or a Tensor (e.g., as returned by layer_input()). The return value depends on object. If object is:
  height,  # Integer, the height of the output shape.
  width,  # Integer, the width of the output shape.
  interpolation = "bilinear",  # String, the interpolation method. Defaults to "bilinear". Supports "bilinear", "nearest", "bicubic", "area", "lanczos3", "lanczos5", "gaussian", and "mitchellcubic".
  crop_to_aspect_ratio = FALSE,  # If TRUE, resize the images without aspect ratio distortion. When the original aspect ratio differs from the target aspect ratio, the output image will be cropped so as to return the largest possible window in the image (of size (height, width)) that matches the target aspect ratio. By default (crop_to_aspect_ratio = FALSE), aspect ratio may not be preserved.
  ...
)
```

Arguments

- `object`: What to compose the new Layer instance with. Typically a Sequential model or a Tensor (e.g., as returned by layer_input()). The return value depends on object. If object is:
  - missing or NULL, the Layer instance is returned.
  - a Sequential model, the model with an additional layer is returned.
  - a Tensor, the output tensor from layer_instance(object) is returned.

- `height`: Integer, the height of the output shape.

- `width`: Integer, the width of the output shape.

- `interpolation`: String, the interpolation method. Defaults to "bilinear". Supports "bilinear", "nearest", "bicubic", "area", "lanczos3", "lanczos5", "gaussian", and "mitchellcubic".

- `crop_to_aspect_ratio`: If TRUE, resize the images without aspect ratio distortion. When the original aspect ratio differs from the target aspect ratio, the output image will be cropped so as to return the largest possible window in the image (of size (height, width)) that matches the target aspect ratio. By default (crop_to_aspect_ratio = FALSE), aspect ratio may not be preserved.

- `...`: standard layer arguments.

Details

Resize the batched image input to target height and width. The input should be a 4D (batched) or 3D (unbatched) tensor in "channels_last" format.
layer_rnn

Description

Base class for recurrent layers

Usage

layer_rnn(
  object,
  cell,
  return_sequences = FALSE,
  return_state = FALSE,
  go_backwards = FALSE,
  stateful = FALSE,
  unroll = FALSE,
  time_major = FALSE,
  ...,
  zero_output_for_mask = FALSE
)

Arguments

<table>
<thead>
<tr>
<th>argument</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>object</td>
<td>What to compose the new Layer instance with. Typically a Sequential model or a Tensor (e.g., as returned by layer_input()). The return value depends on object. If object is:</td>
</tr>
<tr>
<td></td>
<td>• missing or NULL, the Layer instance is returned.</td>
</tr>
<tr>
<td></td>
<td>• a Sequential model, the model with an additional layer is returned.</td>
</tr>
<tr>
<td></td>
<td>• a Tensor, the output tensor from layer_instance(object) is returned.</td>
</tr>
<tr>
<td>cell</td>
<td>A RNN cell instance or a list of RNN cell instances. A RNN cell is a class that has:</td>
</tr>
<tr>
<td></td>
<td>• A call(input_at_t, states_at_t) method, returning (output_at_t, states_at_t_plus_1). The call method of the cell can also take the optional argument constants, see section &quot;Note on passing external constants&quot; below.</td>
</tr>
</tbody>
</table>
layer_rnn

- A state_size attribute. This can be a single integer (single state) in which case it is the size of the recurrent state. This can also be a list of integers (one size per state). The state_size can also be TensorShape or list of TensorShape, to represent high dimension state.
- A output_size attribute. This can be a single integer or a TensorShape, which represent the shape of the output. For backward compatible reason, if this attribute is not available for the cell, the value will be inferred by the first element of the state_size.
- A get_initial_state(inputs=NULL, batch_size=NULL, dtype=NULL) method that creates a tensor meant to be fed to call() as the initial state, if the user didn’t specify any initial state via other means. The returned initial state should have a shape of [batch_size, cell$state_size]. The cell might choose to create a tensor full of zeros, or full of other values based on the cell’s implementation. inputs is the input tensor to the RNN layer, which should contain the batch size as first dimension (inputs$shape[1]), and also dtype (inputs$dtype). Note that the shape[1] might be NULL during the graph construction. Either the inputs or the pair of batch_size and dtype are provided. batch_size is a scalar tensor that represents the batch size of the inputs. dtype is tf.DType that represents the dtype of the inputs. For backward compatibility, if this method is not implemented by the cell, the RNN layer will create a zero filled tensor with the size of [batch_size, cell$state_size]. In the case that cell is a list of RNN cell instances, the cells will be stacked on top of each other in the RNN, resulting in an efficient stacked RNN.

return_sequences

Boolean (default FALSE). Whether to return the last output in the output sequence, or the full sequence.

return_state

Boolean (default FALSE). Whether to return the last state in addition to the output.

go_backwards

Boolean (default FALSE). If TRUE, process the input sequence backwards and return the reversed sequence.

stateful

Boolean (default FALSE). If TRUE, the last state for each sample at index i in a batch will be used as initial state for the sample of index i in the following batch.

unroll

Boolean (default FALSE). If TRUE, the network will be unrolled, else a symbolic loop will be used. Unrolling can speed-up a RNN, although it tends to be more memory-intensive. Unrolling is only suitable for short sequences.

time_major

The shape format of the inputs and outputs tensors. If TRUE, the inputs and outputs will be in shape (timesteps, batch, ...), whereas in the FALSE case, it will be (batch, timesteps, ...). 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.

... standard layer arguments.

zero_output_for_mask

Boolean (default FALSE). Whether the output should use zeros for the masked timesteps. Note that this field is only used when return_sequences is TRUE
and mask is provided. It can useful if you want to reuse the raw output sequence of the RNN without interference from the masked timesteps, eg, merging bidirectional RNNs.

Details

See the Keras RNN API guide for details about the usage of RNN API.

Call arguments

- **inputs**: Input tensor.
- **mask**: Binary tensor of shape `[batch_size, timesteps]` indicating whether a given timestep should be masked. An individual `TRUE` entry indicates that the corresponding timestep should be utilized, while a `FALSE` entry indicates that the corresponding timestep should be ignored.
- **training**: R or Python Boolean indicating whether the layer should behave in training mode or in inference mode. This argument is passed to the cell when calling it. This is for use with cells that use dropout.
- **initial_state**: List of initial state tensors to be passed to the first call of the cell.
- **constants**: List of constant tensors to be passed to the cell at each timestep.

Input shapes

N-D tensor with shape `(batch_size, timesteps, ...)`, or `(timesteps, batch_size, ...) when time_major = TRUE.

Output shape

- if `return_state`: a list of tensors. The first tensor is the output. The remaining tensors are the last states, each with shape `(batch_size, state_size)`, where `state_size` could be a high dimension tensor shape.
- if `return_sequences`: N-D tensor with shape `[batch_size, timesteps, output_size]`, where `output_size` could be a high dimension tensor shape, or `[timesteps, batch_size, output_size]` when `time_major` is TRUE
- else, N-D tensor with shape `[batch_size, output_size]`, where `output_size` could be a high dimension tensor shape.

Masking

This layer supports masking for input data with a variable number of timesteps. To introduce masks to your data, use `layer_embedding()` 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 = list(...)` to the first layer in your model. For functional models with 1 or more Input layers, pass `batch_shape = list(...)` to all the first layers in your model. This is the expected shape of your inputs including the batch size. It should be a list of integers, e.g. `list(32, 10, 100)`. For dimensions which can vary (are not known ahead of time), use `NULL` in place of an integer, e.g. `list(32, NULL, NULL).
- Specify `shuffle = FALSE` when calling `fit()`.

To reset the states of your model, call `layer$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 named argument `states`. The value of `states` should be an array or list of arrays representing the initial state of the RNN layer.

### Passing external constants to RNNs

You can pass "external" constants to the cell using the constants named argument of `RNN$__call__` (as well as `RNN$call`) method. This requires that the cell$call method accepts the same keyword argument constants. Such constants can be used to condition the cell transformation on additional static inputs (not changing over time), a.k.a. an attention mechanism.

### See Also

- [https://www.tensorflow.org/guide/keras/rnn](https://www.tensorflow.org/guide/keras/rnn)
- [https://www.tensorflow.org/api_docs/python/tf/keras/layers/RNN](https://www.tensorflow.org/api_docs/python/tf/keras/layers/RNN)
- [https://keras.io/api/layers/recurrent_layers/rnn](https://keras.io/api/layers/recurrent_layers/rnn)
- `reticulate::py_help(keras$layers$RNN)

Other recurrent layers: `layer_cudnn_gru()`, `layer_cudnn_lstm()`, `layer_gru()`, `layer_lstm()`, `layer_simple_rnn()`
layer_separable_conv_1d

Depthwise separable 1D 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_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,
    input_shape = NULL,
    batch_input_shape = NULL,
    batch_size = NULL,
    dtype = NULL,
    name = NULL,
    trainable = NULL,
    weights = NULL
)
Arguments

**object**
What to compose the new Layer instance with. Typically a Sequential model or a Tensor (e.g., as returned by `layer_input()`). The return value depends on object. If object is:
- missing or NULL, the Layer instance is returned.
- a Sequential model, the model with an additional layer is returned.
- a Tensor, the output tensor from `layer_instance(object)` is returned.

**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.

**depthwise_regularizer**
Regularizer function applied to the depthwise kernel matrix.

**pointwise_regularizer**
Regularizer function applied to the pointwise kernel matrix.
layer_separable_conv_1d

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_1d(), 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

object What to compose the new Layer instance with. Typically a Sequential model or a Tensor (e.g., as returned by layer_input()). The return value depends on object. If object is:
  • missing or NULL, the Layer instance is returned.
  • a Sequential model, the model with an additional layer is returned.
  • a Tensor, the output tensor from layer_instance(object) is returned.

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.

depthwise_regularizer Regularizer function applied to the depthwise kernel matrix.

pointwise_regularizer Regularizer function applied to the pointwise kernel matrix.
**layer_separable_conv_2d**

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**

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_1d()`, `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

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,
  ...
)

Arguments

object
What to compose the new Layer instance with. Typically a Sequential model
or a Tensor (e.g., as returned by layer_input()). The return value depends on
object. If object is:

* missing or NULL, the Layer instance is returned.
* a Sequential model, the model with an additional layer is returned.
* a Tensor, the output tensor from layer_instance(object) is returned.

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

return_sequences

Boolean, whether the layer uses a bias vector.

Boolean. Whether to return the last output in the output sequence, or the full sequence.

Boolean (default FALSE). Whether to return the last state in addition to the output.

Boolean (default FALSE). If TRUE, process the input sequence backwards and return the reversed sequence.

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.

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.

Initializer for the kernel weights matrix, used for the linear transformation of the inputs.

Initializer for the recurrent_kernel weights matrix, used for the linear transformation of the recurrent state.

Initializer for the bias vector.

Regularizer function applied to the kernel weights matrix.

Regularizer function applied to the recurrent_kernel weights matrix.

Regularizer function applied to the bias vector.

Regularizer function applied to the output of the layer (its "activation").

Constraint function applied to the kernel weights matrix.

Constraint function applied to the recurrent_kernel weights matrix.

Constraint function applied to the bias vector.

Float between 0 and 1. Fraction of the units to drop for the linear transformation of the inputs.

Float between 0 and 1. Fraction of the units to drop for the linear transformation of the recurrent state.

... Standard Layer args.

Input shapes

N-D tensor with shape (batch_size, timesteps, ...), or (timesteps, batch_size, ...) when time_major = TRUE.
Output shape

- if return_state: a list of tensors. The first tensor is the output. The remaining tensors are the last states, each with shape (batch_size, state_size), where state_size could be a high dimension tensor shape.

- if return_sequences: N-D tensor with shape [batch_size, timesteps, output_size], where output_size could be a high dimension tensor shape, or [timesteps, batch_size, output_size] when time_major is TRUE

- else, N-D tensor with shape [batch_size, output_size], where output_size could be a high dimension tensor shape.

Masking

This layer supports masking for input data with a variable number of timesteps. To introduce masks to your data, use layer_embedding() 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 = list(...) to the first layer in your model. For functional models with 1 or more Input layers, pass batch_shape = list(...) to all the first layers in your model. This is the expected shape of your inputs including the batch size. It should be a list of integers, e.g. list(32, 10, 100). For dimensions which can vary (are not known ahead of time), use NULL in place of an integer, e.g. list(32, NULL, NULL).

- Specify shuffle = FALSE when calling fit().

To reset the states of your model, call layer$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 named argument states. The value of states should be an array or list of arrays representing the initial state of the RNN layer.
Passing external constants to RNNs

You can pass "external" constants to the cell using the constants named argument of RNN$\_\text{call\_}$ (as well as RNN$\text{call}$) method. This requires that the cell$\text{call}$ method accepts the same keyword argument constants. Such constants can be used to condition the cell transformation on additional static inputs (not changing over time), a.k.a. an attention mechanism.

References

• A Theoretically Grounded Application of Dropout in Recurrent Neural Networks

See Also

• [https://www.tensorflow.org/guide/keras/rnn](https://www.tensorflow.org/guide/keras/rnn)

Other recurrent layers: layer_cudnn_gru(), layer_cudnn_lstm(), layer_gru(), layer_lstm(), layer_rnn()

---

layer_simple_rnn_cell Cell class for SimpleRNN

Description

Cell class for SimpleRNN

Usage

```r
layer_simple_rnn_cell(
  units,
  activation = "tanh",
  use_bias = TRUE,
  kernel_initializer = "glorot_uniform",
  recurrent_initializer = "orthogonal",
  bias_initializer = "zeros",
  kernel_regularizer = NULL,
  recurrent_regularizer = NULL,
  bias_regularizer = NULL,
  kernel_constraint = NULL,
  recurrent_constraint = NULL,
  bias_constraint = NULL,
  dropout = 0,
  recurrent_dropout = 0,
  ...
)
```
Arguments

units Positive integer, dimensionality of the output space.
activation Activation function to use. Default: hyperbolic tangent (tanh). If you pass NULL, no activation is applied (ie. "linear" activation: $a(x) = x$).
use_bias Boolean, (default TRUE), whether the layer uses a bias vector.
kernel_initializer Initializer for the kernel weights matrix, used for the linear transformation of the inputs. Default: glorot_uniform.
recurrent_initializer Initializer for the recurrent_kernel weights matrix, used for the linear transformation of the recurrent state. Default: orthogonal.
bias_initializer Initializer for the bias vector. Default: zeros.
kernel_regularizer Regularizer function applied to the kernel weights matrix. Default: NULL.
recurrent_regularizer Regularizer function applied to the recurrent_kernel weights matrix. Default: NULL.
bias_regularizer Regularizer function applied to the bias vector. Default: NULL.
kernel_constraint Constraint function applied to the kernel weights matrix. Default: NULL.
recurrent_constraint Constraint function applied to the recurrent_kernel weights matrix. Default: NULL.
bias_constraint Constraint function applied to the bias vector. Default: NULL.
dropout Float between 0 and 1. Fraction of the units to drop for the linear transformation of the inputs. Default: 0.
recurrent_dropout Float between 0 and 1. Fraction of the units to drop for the linear transformation of the recurrent state. Default: 0.

Details

See the Keras RNN API guide for details about the usage of RNN API.

This class processes one step within the whole time sequence input, whereas tf.keras.layer.SimpleRNN processes the whole sequence.

See Also

- https://www.tensorflow.org/api_docs/python/tf/keras/layers/SimpleRNNCell
- https://keras.io/api/layers

Other RNN cell layers: layer_gru_cell(), layer_lstm_cell(), layer_stacked_rnn_cells()
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 What to compose the new Layer instance with. Typically a Sequential model or a Tensor (e.g., as returned by layer_input()). The return value depends on object. If object is:
    • missing or NULL, the Layer instance is returned.
    • a Sequential model, the model with an additional layer is returned.
    • a Tensor, the output tensor from layer_instance(object) is returned.
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.
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
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
)

Arguments

object What to compose the new Layer instance with. Typically a Sequential model or a Tensor (e.g., as returned by layer_input()). The return value depends on object. If object is:

• missing or NULL, the Layer instance is returned.
• a Sequential model, the model with an additional layer is returned.
• a Tensor, the output tensor from layer_instance(object) is returned.

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
layer_spatial_dropout_3d

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()

layer_spatial_dropout_3d

Spatial 3D version of Dropout.

Description

This version performs the same function as Dropout, however it drops entire 3D feature maps instead of individual elements. If adjacent voxels within feature maps are strongly correlated (as is normally the case in early convolution layers) then regular dropout will not regularize the activations and will otherwise just result in an effective learning rate decrease. In this case, layer_spatial_dropout_3d will help promote independence between feature maps and should be used instead.

Usage

layer_spatial_dropout_3d(
    object,
    rate,
    data_format = NULL,
    batch_size = NULL,
    name = NULL,
    trainable = NULL,
    weights = NULL
)

________________________________________
layer_stacked_rnn_cells

Wrapper allowing a stack of RNN cells to behave as a single cell

Description

Used to implement efficient stacked RNNs.

Arguments

object  What to compose the new Layer instance with. Typically a Sequential model or a Tensor (e.g., as returned by layer_input()). The return value depends on object. If object is:
  • missing or NULL, the Layer instance is returned.
  • a Sequential model, the model with an additional layer is returned.
  • a Tensor, the output tensor from layer_instance(object) is returned.
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_string_lookup

Usage

layer_stacked_rnn_cells(cells, ...)

Arguments

cells List of RNN cell instances.
...
standard layer arguments.

See Also

• https://www.tensorflow.org/api_docs/python/tf/keras/layers/StackedRNNCells

Other RNN cell layers: layer_gru_cell(), layer_lstm_cell(), layer_simple_rnn_cell()

layer_string_lookup  A preprocessing layer which maps string features to integer indices.

Description

A preprocessing layer which maps string features to integer indices.

Usage

layer_string_lookup(
    object,
    max_tokens = NULL,
    num_oov_indices = 1L,
    mask_token = NULL,
    oov_token = "[UNK]",
    vocabulary = NULL,
    encoding = NULL,
    invert = FALSE,
    output_mode = "int",
    sparse = FALSE,
    pad_to_max_tokens = FALSE,
    ...
)

Arguments

object What to compose the new Layer instance with. Typically a Sequential model or a Tensor (e.g., as returned by layer_input()). The return value depends on object. If object is:
• missing or NULL, the Layer instance is returned.
• a Sequential model, the model with an additional layer is returned.
• a Tensor, the output tensor from layer_instance(object) is returned.
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>max_tokens</td>
<td>The maximum size of the vocabulary for this layer. If NULL, there is no cap</td>
</tr>
<tr>
<td></td>
<td>on the size of the vocabulary. Note that this size includes the OOV and</td>
</tr>
<tr>
<td></td>
<td>mask tokens. Default to NULL.</td>
</tr>
<tr>
<td>num_oov_indices</td>
<td>The number of out-of-vocabulary tokens to use. If this value is more than 1,</td>
</tr>
<tr>
<td></td>
<td>OOV inputs are hashed to determine their OOV value. If this value is 0, OOV</td>
</tr>
<tr>
<td></td>
<td>inputs will cause an error when calling the layer. Defaults to 1.</td>
</tr>
<tr>
<td>mask_token</td>
<td>A token that represents masked inputs. When output_mode is &quot;int&quot;, the token</td>
</tr>
<tr>
<td></td>
<td>is included in vocabulary and mapped to index 0. In other output modes, the</td>
</tr>
<tr>
<td></td>
<td>token will not appear in the vocabulary and instances of the mask token in</td>
</tr>
<tr>
<td></td>
<td>the input will be dropped. If set to NULL, no mask term will be added.</td>
</tr>
<tr>
<td>num_oov_indices</td>
<td>The number of out-of-vocabulary tokens to use. If this value is more than 1,</td>
</tr>
<tr>
<td></td>
<td>OOV inputs are hashed to determine their OOV value. If this value is 0, OOV</td>
</tr>
<tr>
<td></td>
<td>inputs will cause an error when calling the layer. Defaults to 1.</td>
</tr>
<tr>
<td>mask_token</td>
<td>A token that represents masked inputs. When output_mode is &quot;int&quot;, the token</td>
</tr>
<tr>
<td></td>
<td>is included in vocabulary and mapped to index 0. In other output modes, the</td>
</tr>
<tr>
<td></td>
<td>token will not appear in the vocabulary and instances of the mask token in</td>
</tr>
<tr>
<td></td>
<td>the input will be dropped. If set to NULL, no mask term will be added.</td>
</tr>
<tr>
<td>vocabulary</td>
<td>Optional. Either an array of strings or a string path to a text file. If</td>
</tr>
<tr>
<td></td>
<td>passing an array, can pass a list, list, 1D numpy array, or 1D tensor</td>
</tr>
<tr>
<td></td>
<td>containing the string vocabulary terms. If passing a file path, the file</td>
</tr>
<tr>
<td></td>
<td>should contain one line per term in the vocabulary. If this argument is set,</td>
</tr>
<tr>
<td></td>
<td>there is no need to adapt the layer.</td>
</tr>
<tr>
<td>encoding</td>
<td>String encoding. Default of NULL is equivalent to &quot;utf-8&quot;.</td>
</tr>
<tr>
<td>invert</td>
<td>Only valid when output_mode is &quot;int&quot;. If TRUE, this layer will map indices</td>
</tr>
<tr>
<td></td>
<td>to vocabulary items instead of mapping vocabulary items to indices. Default</td>
</tr>
<tr>
<td></td>
<td>to FALSE.</td>
</tr>
<tr>
<td>output_mode</td>
<td>Specification for the output of the layer. Defaults to &quot;int&quot;. Values can be</td>
</tr>
<tr>
<td></td>
<td>&quot;int&quot;, &quot;one_hot&quot;, &quot;multi_hot&quot;, &quot;count&quot;, or &quot;tf_idf&quot; configuring the layer</td>
</tr>
<tr>
<td></td>
<td>as follows:</td>
</tr>
<tr>
<td></td>
<td>• &quot;int&quot;: Return the raw integer indices of the input tokens.</td>
</tr>
<tr>
<td></td>
<td>• &quot;one_hot&quot;: Encodes each individual element in the input into an array the</td>
</tr>
<tr>
<td></td>
<td>same size as the vocabulary, containing a 1 at the element index. If the last</td>
</tr>
<tr>
<td></td>
<td>dimension is size 1, will encode on that dimension. If the last dimension is</td>
</tr>
<tr>
<td></td>
<td>not size 1, will append a new dimension for the encoded output.</td>
</tr>
<tr>
<td></td>
<td>• &quot;multi_hot&quot;: Encodes each sample in the input into a single array the</td>
</tr>
<tr>
<td></td>
<td>same size as the vocabulary, containing a 1 for each vocabulary term present</td>
</tr>
<tr>
<td></td>
<td>in the sample. Treats the last dimension as the sample dimension, if input</td>
</tr>
<tr>
<td></td>
<td>shape is (..., sample_length), output shape will be (..., num_tokens).</td>
</tr>
<tr>
<td></td>
<td>• &quot;count&quot;: As &quot;multi_hot&quot;, but the int array contains a count of the number</td>
</tr>
<tr>
<td></td>
<td>of times the token at that index appeared in the sample.</td>
</tr>
<tr>
<td></td>
<td>• &quot;tf_idf&quot;: As &quot;multi_hot&quot;, but the TF-IDF algorithm is applied to find the</td>
</tr>
<tr>
<td></td>
<td>value in each token slot. For &quot;int&quot; output, any shape of input and output is</td>
</tr>
<tr>
<td></td>
<td>supported. For all other output modes, currently only output up to rank 2 is</td>
</tr>
<tr>
<td></td>
<td>supported.</td>
</tr>
<tr>
<td>sparse</td>
<td>Boolean. Only applicable when output_mode is &quot;multi_hot&quot;, &quot;count&quot;, or</td>
</tr>
<tr>
<td></td>
<td>&quot;tf_idf&quot;. If TRUE, returns a SparseTensor instead of a dense Tensor.</td>
</tr>
<tr>
<td></td>
<td>Defaults to FALSE.</td>
</tr>
<tr>
<td>pad_to_max_tokens</td>
<td>Only applicable when output_mode is &quot;multi_hot&quot;, &quot;count&quot;, or &quot;tf_idf&quot;. If</td>
</tr>
<tr>
<td></td>
<td>TRUE, the output will have its feature axis padded to max_tokens even if the</td>
</tr>
</tbody>
</table>
number of unique tokens in the vocabulary is less than max_tokens, resulting in a tensor of shape [batch_size, max_tokens] regardless of vocabulary size. Defaults to FALSE.

... standard layer arguments.

Details

This layer translates a set of arbitrary strings into integer output via a table-based vocabulary lookup. The vocabulary for the layer must be either supplied on construction or learned via adapt(). During adapt(), the layer will analyze a data set, determine the frequency of individual strings tokens, and create a vocabulary from them. If the vocabulary is capped in size, the most frequent tokens will be used to create the vocabulary and all others will be treated as out-of-vocabulary (OOV).

There are two possible output modes for the layer. When output_mode is "int", input strings are converted to their index in the vocabulary (an integer). When output_mode is "multi_hot", "count", or "tf_idf", input strings are encoded into an array where each dimension corresponds to an element in the vocabulary.

The vocabulary can optionally contain a mask token as well as an OOV token (which can optionally occupy multiple indices in the vocabulary, as set by num_oov_indices). The position of these tokens in the vocabulary is fixed. When output_mode is "int", the vocabulary will begin with the mask token (if set), followed by OOV indices, followed by the rest of the vocabulary. When output_mode is "multi_hot", "count", or "tf_idf" the vocabulary will begin with OOV indices and instances of the mask token will be dropped.

See Also

- adapt()
- https://www.tensorflow.org/api_docs/python/tf/keras/layers/StringLookup
- https://keras.io/api/layers/preprocessing_layers/categorical/string_lookup

Other categorical features preprocessing layers: layer_category_encoding(), layer hashing(), layer_integer_lookup()

Other preprocessing layers: layer_category_encoding(), layer_center_crop(), layer_discretization(), layer_hashing(), layer_integer_lookup(), layer_normalization(), layer_random_brightness(), layer_random_contrast(), layer_random_crop(), layer_random_flip(), layer_random_height(), layer_random_rotation(), layer_random_translation(), layer_random_width(), layer_random_zoom(), layer_rescaling(), layer_resizing(), layer_text_vectorization()

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, (inputs[[1]] - inputs[[2]]), also of the same shape.
layer_text_vectorization

A preprocessing layer which maps text features to integer sequences.

Usage

    layer_text_vectorization(
        object,
        max_tokens = NULL,
        standardize = "lower_and_strip_punctuation",
        split = "whitespace",
        ngrams = NULL,
        output_mode = "int",
        output_sequence_length = NULL,
        pad_to_max_tokens = FALSE,
        vocabulary = NULL,
        ...
    )

    get_vocabulary(object, include_special_tokens = TRUE)

    set_vocabulary(object, vocabulary, idf_weights = NULL, ...)

Description

A preprocessing layer which maps text features to integer sequences.

Usage

    layer_subtract(inputs, ...)

Arguments

    inputs               A list of input tensors (exactly 2). Can be missing.
    ...                 Standard layer arguments (must be named).

Value

A tensor, the difference of the inputs. If inputs is missing, a keras layer instance is returned.

See Also

- https://www.tensorflow.org/api_docs/python/tf/keras/layers/subtract
- https://www.tensorflow.org/api_docs/python/tf/keras/layers/Subtract
- https://keras.io/api/layers/merging_layers/subtract

Other merge layers: layer_average(), layer_concatenate(), layer_dot(), layer_maximum(), layer_minimum(), layer_multiply()
Arguments

**object**
What to compose the new Layer instance with. Typically a Sequential model or a Tensor (e.g., as returned by `layer_input()`). The return value depends on `object`. If `object` is:

- missing or NULL, the Layer instance is returned.
- a Sequential model, the model with an additional layer is returned.
- a Tensor, the output tensor from `layer_instance(object)` is returned.

**max_tokens**
The maximum size of the vocabulary for this layer. If NULL, there is no cap on the size of the vocabulary. Note that this vocabulary contains 1 OOV token, so the effective number of tokens is `max_tokens - 1 - (1 if output_mode == "int" else 0)`.

**standardize**
Optional specification for standardization to apply to the input text. Values can be NULL (no standardization), "lower_and_strip_punctuation" (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), "whitespace" (split on ASCII whitespace), or a Callable. The default is "whitespace".

**ngrams**
Optional specification for ngrams to create from the possibly-split input text. Values can be NULL, an integer or list of integers; passing an integer will create ngrams up to that integer, and passing a list of integers will create ngrams for the specified values in the list. Passing NULL means that no ngrams will be created.

**output_mode**
Optional specification for the output of the layer. Values can be "int", "multi_hot", "count" or "tf_idf", configuring the layer as follows:

- "int": Outputs integer indices, one integer index per split string token. When `output_mode == "int"`, 0 is reserved for masked locations; this reduces the vocab size to `max_tokens - 2` instead of `max_tokens - 1`.
- "multi_hot": Outputs a single int array per batch, of either vocab_size or `max_tokens` size, containing 1s in all elements where the token mapped to that index exists at least once in the batch item.
- "count": Like "multi_hot", but the int array contains a count of the number of times the token at that index appeared in the batch item.
- "tf_idf": Like "multi_hot", but the TF-IDF algorithm is applied to find the value in each token slot. For "int" output, any shape of input and output is supported. For all other output modes, currently only rank 1 inputs (and rank 2 outputs after splitting) are supported.

**output_sequence_length**
Only valid in INT mode. If set, the output will have its time dimension padded or truncated to exactly `output_sequence_length` values, resulting in a tensor of shape `(batch_size, output_sequence_length)` regardless of how many tokens resulted from the splitting step. Defaults to NULL.

**pad_to_max_tokens**
Only valid in "multi_hot", "count", and "tf_idf" modes. If TRUE, the output will have its feature axis padded to `max_tokens` even if the number of unique tokens in the vocabulary is less than `max_tokens`, resulting in a tensor of shape `(batch_size, max_tokens)` regardless of vocabulary size. Defaults to FALSE.
layer_text_vectorization

vocabulary Optional for layer_text_vectorization(). Either an array of strings or a string path to a text file. If passing an array, can pass an R list or character vector, 1D numpy array, or 1D tensor containing the string vocabulary terms. If passing a file path, the file should contain one line per term in the vocabulary. If vocabulary is set (either by passing layer_text_vectorization(vocabulary = ...) or by calling set_vocabulary(layer, vocabulary = ...)), there is no need to adapt() the layer.

... standard layer arguments.

include_special_tokens

If True, the returned vocabulary will include the padding and OOV tokens, and a term’s index in the vocabulary will equal the term’s index when calling the layer. If False, the returned vocabulary will not include any padding or OOV tokens.

idf_weights An R vector, 1D numpy array, or 1D tensor of inverse document frequency weights with equal length to vocabulary. Must be set if output_mode is "tf_idf". Should not be set otherwise.

Details

This layer has basic options for managing text in a Keras model. It transforms a batch of strings (one example = one string) into either a list of token indices (one example = 1D tensor of integer token indices) or a dense representation (one example = 1D tensor of float values representing data about the example’s tokens).

The vocabulary for the layer must be either supplied on construction or learned via adapt(). When this layer is adapted, it will analyze the dataset, determine the frequency of individual string values, and create a vocabulary from them. This vocabulary can have unlimited size or be capped, depending on the configuration options for this layer; if there are more unique values in the input than the maximum vocabulary size, the most frequent terms will be used to create the vocabulary.

The processing of each example contains the following steps:

1. Standardize each example (usually lowercasing + punctuation stripping)
2. Split each example into substrings (usually words)
3. Recombine substrings into tokens (usually ngrams)
4. Index tokens (associate a unique int value with each token)
5. Transform each example using this index, either into a vector of ints or a dense float vector.

Some notes on passing callables to customize splitting and normalization for this layer:

1. Any callable can be passed to this Layer, but if you want to serialize this object you should only pass functions that are registered Keras serializables (see tf$keras$utils$register_keras_serializable for more details).
2. When using a custom callable for standardize, the data received by the callable will be exactly as passed to this layer. The callable should return a tensor of the same shape as the input.
3. When using a custom callable for split, the data received by the callable will have the 1st dimension squeezed out - instead of matrix(c("string to split", "another string to
layer_unit_normalization

Unit normalization layer

Description

Unit normalization layer

Usage

layer_unit_normalization(object, axis = -1L, ...)

Arguments

object What to compose the new Layer instance with. Typically a Sequential model or a Tensor (e.g., as returned by layer_input()). The return value depends on object. If object is:
  • missing or NULL, the Layer instance is returned.
  • a Sequential model, the model with an additional layer is returned.
  • a Tensor, the output tensor from layer_instance(object) is returned.
axis Integer or list. The axis or axes to normalize across. Typically this is the features axis or axes. The left-out axes are typically the batch axis or axes. Defaults to −1, the last dimension in the input.
...
  • standard layer arguments.

data <- as_tensor(1:6, shape = c(2, 3), dtype = "float32")
normalized_data <- data %>% layer_unit_normalization()
for(row in 1:2)
  normalized_data[row, ] %>%
See Also

• adapt()
• https://www.tensorflow.org/api_docs/python/tf/keras/layers/TextVectorization
• https://keras.io/api/layers/preprocessing_layers/text/text_vectorization

Other preprocessing layers: layer_category_encoding(), layer_center_crop(), layer_discretization(), layer_hashing(), layer_integer_lookup(), layer_normalization(), layer_random_brightness(), layer_random_contrast(), layer_random_crop(), layer_random_flip(), layer_random_height(), layer_random_rotation(), layer_random_translation(), layer_random_width(), layer_random_zoom(), layer_rescaling(), layer_resizing(), layer_string_lookup()
layer_upsampling_1d

```r
\{(sum(.^2))\} \>%>
print()
# tf.Tensor(0.9999999, shape=(), dtype=float32)
# tf.Tensor(1.0, shape=(), dtype=float32)
```

Details

Normalize a batch of inputs so that each input in the batch has a L2 norm equal to 1 (across the axes specified in `axis`).

See Also

- [https://www.tensorflow.org/api_docs/python/tf/keras/layers/UnitNormalization](https://www.tensorflow.org/api_docs/python/tf/keras/layers/UnitNormalization)

---

**layer_upsampling_1d**  
*Upsampling layer for 1D inputs.*

Description

Repeats each temporal step `size` times along the time axis.

Usage

```
layer_upsampling_1d(
  object,
  size = 2L,
  batch_size = NULL,
  name = NULL,
  trainable = NULL,
  weights = NULL
)
```

Arguments

- **object**: What to compose the new `Layer` instance with. Typically a `Sequential` model or a `Tensor` (e.g., as returned by `layer_input()`). The return value depends on `object`. If `object` is:
  - missing or `NULL`, the `Layer` instance is returned.
  - a `Sequential` model, the model with an additional layer is returned.
  - a `Tensor`, the output tensor from `layer_instance(object)` is returned.
- **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_1d(), 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_3d

- **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 upscaling 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)

**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_1d(), layer_depthwise_conv_2d(), layer_separable_conv_1d(), layer_separable_conv_2d(), layer_upsampling_1d(), layer_upsampling_3d(), layer_zero_padding_1d(), layer_zero_padding_2d(), layer_zero_padding_3d()

---

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.
layer_upsampling_3d

Usage

layer_upsampling_3d(
  object,
  size = c(2L, 2L, 2L),
  data_format = NULL,
  batch_size = NULL,
  name = NULL,
  trainable = NULL,
  weights = NULL
)

Arguments

object What to compose the new Layer instance with. Typically a Sequential model or a Tensor (e.g., as returned by `layer_input()`). The return value depends on `object`. If `object` is:
  • missing or `NULL`, the Layer instance is returned.
  • a Sequential model, the model with an additional layer is returned.
  • a Tensor, the output tensor from `layer_instance(object)` is returned.

size int, or list of 3 integers. The upsampling factors for dim1, dim2 and dim3.

data_format A string, one of `channels_last` (default) or `channels_first`. The ordering of the dimensions in the inputs. `channels_last` corresponds to inputs with shape (batch, 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, 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)
layer_zero_padding_1d

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_1d(),
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
layer_zero_padding_1d(
  object,  
  padding = 1L,  
  batch_size = NULL,  
  name = NULL,  
  trainable = NULL,  
  weights = NULL
)

Arguments
object  What to compose the new Layer instance with. Typically a Sequential model
        or a Tensor (e.g., as returned by layer_input()). The return value depends on
        object. If object is:
        • missing or NULL, the Layer instance is returned.
        • a Sequential model, the model with an additional layer is returned.
        • a Tensor, the output tensor from layer_instance(object) is returned.
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.
**layer_zero_padding_2d**

### 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_1d()`, `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

```r
layer_zero_padding_2d(
  object,
  padding = c(1L, 1L),
  data_format = NULL,
  batch_size = NULL,
  name = NULL,
  trainable = NULL,
  weights = NULL
)
```

### Arguments

- **object**: What to compose the new Layer instance with. Typically a Sequential model or a Tensor (e.g., as returned by `layer_input()`). The return value depends on `object`. If `object` is:
  - missing or NULL, the Layer instance is returned.
  - a Sequential model, the model with an additional layer is returned.
  - a Tensor, the output tensor from `layer_instance(object)` is returned.

- **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.
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).

- 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_1d()`, `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_3d()`
Usage

layer_zero_padding_3d(
    object,
    padding = c(1L, 1L, 1L),
    data_format = NULL,
    batch_size = NULL,
    name = NULL,
    trainable = NULL,
    weights = NULL
)

Arguments

object

What to compose the new Layer instance with. Typically a Sequential model or a Tensor (e.g., as returned by `layer_input()`). The return value depends on object. If object is:

- missing or NULL, the Layer instance is returned.
- a Sequential model, the model with an additional layer is returned.
- a Tensor, the output tensor from `layer_instance(object)` is returned.

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_depthwise_conv_1d()`, `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()`

---

```python
learning_rate_schedule_cosine_decay

A LearningRateSchedule that uses a cosine decay schedule

Description

A LearningRateSchedule that uses a cosine decay schedule

Usage

```python
learning_rate_schedule_cosine_decay(
    initial_learning_rate,
    decay_steps,
    alpha = 0,
    ...
    name = NULL
)
```

Arguments

- **initial_learning_rate**
  A scalar `float32` or `float64` Tensor or a R number. The initial learning rate.
- **decay_steps**
  A scalar `int32` or `int64` Tensor or an R number. Number of steps to decay over.
- **alpha**
  A scalar `float32` or `float64` Tensor or an R number. Minimum learning rate value as a fraction of initial_learning_rate.
- **...**
  For backwards and forwards compatibility
- **name**
  String. Optional name of the operation. Defaults to 'CosineDecay'.
Details

See Loshchilov & Hutter, ICLR2016, SGDR: Stochastic Gradient Descent with Warm Restarts.

When training a model, it is often useful to lower the learning rate as the training progresses. This schedule applies a cosine decay function to an optimizer step, given a provided initial learning rate. It requires a step value to compute the decayed learning rate. You can just pass a TensorFlow variable that you increment at each training step.

The schedule is a 1-arg callable that produces a decayed learning rate when passed the current optimizer step. This can be useful for changing the learning rate value across different invocations of optimizer functions. It is computed as:

\[
\text{decayed}_\text{learning}_\text{rate} \leftarrow \text{function}(\text{step}) \{
\text{step} \leftarrow \min(\text{step}, \text{decay}_\text{steps})
\text{cosine}_\text{decay} = \leftarrow 0.5 \times (1 + \cos(\pi \times \text{step} / \text{decay}_\text{steps}))
\text{decayed} \leftarrow (1 - \alpha) \times \text{cosine}_\text{decay} + \alpha 
\text{initial}_\text{learning}_\text{rate} \times \text{decayed}
\}
\]

Example usage:

```r
decay_steps <- 1000
lr_decayed_fn <-
  learning_rate_schedule_cosine_decay(initial_learning_rate, decay_steps)
```

You can pass this schedule directly into a keras Optimizer as the learning_rate.

See Also

- [https://www.tensorflow.org/api_docs/python/tf/keras/optimizers/schedules/CosineDecay](https://www.tensorflow.org/api_docs/python/tf/keras/optimizers/schedules/CosineDecay)

---

**learning_rate_schedule_cosine_decay_restarts**

A LearningRateSchedule that uses a cosine decay schedule with restarts

---

Description

A LearningRateSchedule that uses a cosine decay schedule with restarts

Usage

```r
learning_rate_schedule_cosine_decay_restarts(
  initial_learning_rate,
  first_decay_steps,
  t_mul = 2,
  m_mul = 1,
  alpha = 0,
  ...,
  name = NULL
)
```
Arguments

initial_learning_rate
A scalar float32 or float64 Tensor or an R number. The initial learning rate.

first_decay_steps
A scalar int32 or int64 Tensor or an R number. Number of steps to decay over.

t_mul
A scalar float32 or float64 Tensor or an R number. Used to derive the number of iterations in the i-th period.

m_mul
A scalar float32 or float64 Tensor or an R number. Used to derive the initial learning rate of the i-th period.

alpha
A scalar float32 or float64 Tensor or an R number. Minimum learning rate value as a fraction of the initial_learning_rate.

For backwards and forwards compatibility

name
String. Optional name of the operation. Defaults to 'SGDRDecay'.

Details

See Loshchilov & Hutter, ICLR2016, SGDR: Stochastic Gradient Descent with Warm Restarts.

When training a model, it is often useful to lower the learning rate as the training progresses. This schedule applies a cosine decay function with restarts to an optimizer step, given a provided initial learning rate. It requires a step value to compute the decayed learning rate. You can just pass a TensorFlow variable that you increment at each training step.

The schedule is a 1-arg callable that produces a decayed learning rate when passed the current optimizer step. This can be useful for changing the learning rate value across different invocations of optimizer functions.

The learning rate multiplier first decays from 1 to alpha for first_decay_steps steps. Then, a warm restart is performed. Each new warm restart runs for t_mul times more steps and with m_mul times initial learning rate as the new learning rate.

You can pass this schedule directly into a keras Optimizer as the learning_rate.

See Also

- https://www.tensorflow.org/api_docs/python/tf/keras/optimizers/schedules/CosineDecayRestarts

Description

A LearningRateSchedule that uses an exponential decay schedule
**Usage**

```r
learning_rate_schedule_exponential_decay(
    initial_learning_rate,
    decay_steps,
    decay_rate,
    staircase = FALSE,
    ..., 
    name = NULL
)
```

**Arguments**

- `initial_learning_rate`: A scalar `float32` or `float64` Tensor or a `R` number. The initial learning rate.
- `decay_steps`: A scalar `int32` or `int64` Tensor or an `R` number. Must be positive. See the decay computation above.
- `decay_rate`: A scalar `float32` or `float64` Tensor or an `R` number. The decay rate.
- `staircase`: Boolean. If `TRUE` decay the learning rate at discrete intervals.
- `...`: For backwards and forwards compatibility
- `name`: String. Optional name of the operation. Defaults to `'ExponentialDecay'`.

**Details**

When training a model, it is often useful to lower the learning rate as the training progresses. This schedule applies an exponential decay function to an optimizer step, given a provided initial learning rate.

The schedule is a 1-arg callable that produces a decayed learning rate when passed the current optimizer step. This can be useful for changing the learning rate value across different invocations of optimizer functions. It is computed as:

```r
decayed_learning_rate <- function(step)
    initial_learning_rate * decay_rate ^ (step / decay_steps)
```

If the argument `staircase` is `TRUE`, then `step / decay_steps` is an integer division (`%/%`) and the decayed learning rate follows a staircase function.

You can pass this schedule directly into an optimizer as the learning rate (see example) Example: When fitting a Keras model, decay every 100000 steps with a base of 0.96:

```r
initial_learning_rate <- 0.1
lr_schedule <- learning_rate_schedule_exponential_decay(
    initial_learning_rate,
    decay_steps = 100000,
    decay_rate = 0.96,
    staircase = TRUE)

model %>% compile()
```
optimizer = optimizer_sgd(learning_rate = lr_schedule),
loss = 'sparse_categorical_crossentropy',
metrics = 'accuracy')

model %>% fit(data, labels, epochs = 5)

See Also

- https://www.tensorflow.org/api_docs/python/tf/keras/optimizers/schedules/ExponentialDecay

learning_rate_schedule_inverse_time_decay

A LearningRateSchedule that uses an inverse time decay schedule

Description

A LearningRateSchedule that uses an inverse time decay schedule

Usage

learning_rate_schedule_inverse_time_decay(
  initial_learning_rate,
  decay_steps,
  decay_rate,
  staircase = FALSE,
  ..., 
  name = NULL 
)

Arguments

initial_learning_rate
  A scalar float32 or float64 Tensor or an R number. The initial learning rate.

decay_steps
  A scalar int32 or int64 Tensor or an R number. How often to apply decay.

decay_rate
  An R number. The decay rate.

staircase
  Boolean. Whether to apply decay in a discrete staircase, as opposed to continuous, fashion.

... 
  For backwards and forwards compatibility

name
  String. Optional name of the operation. Defaults to 'InverseTimeDecay'.
Details

When training a model, it is often useful to lower the learning rate as the training progresses. This schedule applies the inverse decay function to an optimizer step, given a provided initial learning rate. It requires a step value to compute the decayed learning rate. You can just pass a TensorFlow variable that you increment at each training step.

The schedule is a 1-arg callable that produces a decayed learning rate when passed the current optimizer step. This can be useful for changing the learning rate value across different invocations of optimizer functions. It is computed as:

\[
\text{decayed_learning_rate} \leftarrow \text{function}(\text{step}) \{
\text{initial_learning_rate} / (1 + \text{decay_rate} \times \text{step} / \text{decay_step})
\}
\]

or, if `staircase` is `TRUE`, as:

\[
\text{decayed_learning_rate} \text{ function}(\text{step}) \{
\text{initial_learning_rate} / (1 + \text{decay_rate} \times \text{floor}(\text{step} / \text{decay_step}))
\}
\]

You can pass this schedule directly into a keras Optimizer as the `learning_rate`.

Example: Fit a Keras model when decaying $1/t$ with a rate of $0.5$:

```r
...
initial_learning_rate <- 0.1
decay_steps <- 1.0
decay_rate <- 0.5
learning_rate_fn <- learning_rate_schedule_inverse_time_decay(
  initial_learning_rate, decay_steps, decay_rate)
model %>%
  compile(optimizer = optimizer_sgd(learning_rate = learning_rate_fn),
  loss = 'sparse_categorical_crossentropy',
  metrics = 'accuracy')
model %>% fit(data, labels, epochs = 5)
```

See Also

- https://www.tensorflow.org/api_docs/python/tf/keras/optimizers/schedules/InverseTimeDecay
A LearningRateSchedule that uses a piecewise constant decay schedule

Usage

```r
learning_rate_schedule_piecewise_constant_decay(
    boundaries,
    values,
    ..., 
    name = NULL
)
```

Arguments

- `boundaries` A list of Tensors or R numerics with strictly increasing entries, and with all elements having the same type as the optimizer step.
- `values` A list of Tensors or R numerics that specifies the values for the intervals defined by `boundaries`. It should have one more element than `boundaries`, and all elements should have the same type.
- `...` For backwards and forwards compatibility
- `name` A string. Optional name of the operation. Defaults to 'PiecewiseConstant'.

Details

The function returns a 1-arg callable to compute the piecewise constant when passed the current optimizer step. This can be useful for changing the learning rate value across different invocations of optimizer functions.

Example: use a learning rate that's 1.0 for the first 100001 steps, 0.5 for the next 10000 steps, and 0.1 for any additional steps.

```r
step <- tf$Variable(0, trainable=FALSE)
boundaries <- as.integer(c(100000, 110000))
values <- c(1.0, 0.5, 0.1)
learning_rate_fn <- learning_rate_schedule_piecewise_constant_decay(
    boundaries, values)
# Later, whenever we perform an optimization step, we pass in the step.
learning_rate <- learning_rate_fn(step)
```

You can pass this schedule directly into a keras Optimizer as the `learning_rate`. 
See Also

- https://www.tensorflow.org/api_docs/python/tf/keras/optimizers/schedules/PiecewiseConstantDecay

---

**learning_rate_schedule_polynomial_decay**

*A LearningRateSchedule that uses a polynomial decay schedule*

---

**Description**

A LearningRateSchedule that uses a polynomial decay schedule

**Usage**

```python
learning_rate_schedule_polynomial_decay(
    initial_learning_rate,
    decay_steps,
    end_learning_rate = 1e-04,
    power = 1,
    cycle = FALSE,
    ...,
    name = NULL
)
```

**Arguments**

- `initial_learning_rate`  
  A scalar float32 or float64 Tensor or an R number. The initial learning rate.
- `decay_steps`  
  A scalar int32 or int64 Tensor or an R number. Must be positive. See the decay computation above.
- `end_learning_rate`  
  A scalar float32 or float64 Tensor or an R number. The minimal end learning rate.
- `power`  
  A scalar float32 or float64 Tensor or an R number. The power of the polynomial. Defaults to linear, 1.0.
- `cycle`  
  A boolean, whether or not it should cycle beyond decay_steps.
- `...`  
  For backwards and forwards compatibility
- `name`  
  String. Optional name of the operation. Defaults to 'PolynomialDecay'.

**Details**

It is commonly observed that a monotonically decreasing learning rate, whose degree of change is carefully chosen, results in a better performing model. This schedule applies a polynomial decay function to an optimizer step, given a provided `initial_learning_rate`, to reach an `end_learning_rate` in the given `decay_steps`.
It requires a step value to compute the decayed learning rate. You can just pass a TensorFlow variable that you increment at each training step.

The schedule is a 1-arg callable that produces a decayed learning rate when passed the current optimizer step. This can be useful for changing the learning rate value across different invocations of optimizer functions. It is computed as:

```r
decayed_learning_rate <- function(step) {
  step <- min(step, decay_steps)
  ((initial_learning_rate - end_learning_rate) *
   (1 - step / decay_steps) ^ (power)
  ) + end_learning_rate
}
```

If `cycle` is `TRUE` then a multiple of `decay_steps` is used, the first one that is bigger than `step`.

```r
decayed_learning_rate <- function(step) {
  decay_steps <- decay_steps * ceiling(step / decay_steps)
  ((initial_learning_rate - end_learning_rate) *
   (1 - step / decay_steps) ^ (power)
  ) + end_learning_rate
}
```

You can pass this schedule directly into a keras Optimizer as the `learning_rate`.

Example: Fit a model while decaying from 0.1 to 0.01 in 10000 steps using sqrt (i.e. `power=0.5`):

```r
... 
starter_learning_rate <- 0.1
end_learning_rate <- 0.01
decay_steps <- 10000
learning_rate_fn <- learning_rate_schedule_polynomial_decay(
  starter_learning_rate, decay_steps, end_learning_rate, power = 0.5)

model %>%
  compile(optimizer = optimizer_sgd(learning_rate = learning_rate_fn),
          loss = 'sparse_categorical_crossentropy',
          metrics = 'accuracy')

model %>% fit(data, labels, epochs = 5)
```

See Also

- https://www.tensorflow.org/api_docs/python/tf/keras/optimizers/schedules/PolynomialDecay
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"
)
```

```r
loss_categorical_crossentropy(
  y_true,
  y_pred,
  from_logits = FALSE,
  label_smoothing = 0L,
  axis = -1L,
  ..., 
  reduction = "auto",
  name = "categorical_crossentropy"
)
```

```r
loss_categorical_hinge(
  y_true,
  y_pred,
  ..., 
  reduction = "auto",
  name = "categorical_hinge"
)
```

```r
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,
    ...,
    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()`:
  ```r
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.

\[
\log(\cosh(x)) \approx \frac{x^2}{2} \quad \text{for small } x
\]

and

\[
\log(\cosh(x)) \approx |x| - \log(2) \quad \text{for large } x.
\]

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 \( \cosh(y_{\text{pred}} - y_{\text{true}}) \) is too large to be represented in the chosen precision.

**See Also**

compile.keras.engine.training.Model(), loss_binary_crossentropy()

---

**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:

\[
p(\text{word}) = \min(1, \sqrt{\frac{\text{word}_f}{\text{sampling}_f}} / \left(\frac{\text{word}_f}{\text{sampling}_f}\right))
\]

We assume that the word frequencies follow Zipf's law \( (s=1) \) to derive a numerical approximation of frequency(rank):

\[
\text{frequency(rank)} \sim \frac{1}{(\text{rank} + \gamma + 1/2)} - \frac{1}{(12 \text{rank})}
\]

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: \( p(\text{word}) = \min(1, \sqrt{\frac{\text{word}\.\text{frequency}}{\text{sampling\_factor}}} / \frac{\text{word}\.\text{frequency}}{\text{sampling\_factor}}) \)

See Also

Other text preprocessing: \texttt{pad\_sequences()}, \texttt{skipgrams()}, \texttt{text\_hashing\_trick()}, \texttt{text\_one\_hot()}, \texttt{text\_to\_word\_sequence()}

---

**Define new keras types**

**Description**

These functions can be used to make custom objects that fit in the family of existing keras types. For example, \texttt{new\_layer\_class()} will return a class constructor, an object that behaves like other layer functions such as \texttt{layer\_dense()}. \texttt{new\_callback\_class()} will return an object that behaves similarly to other callback functions, like \texttt{callback\_reduce\_lr\_on\_plateau()}, and so on. All arguments with a default NULL value are optional methods that can be provided.

**Usage**

\texttt{mark\_active(x)}

\texttt{new\_metric\_class(classname, ..., initialize, update\_state, result)}

\texttt{new\_loss\_class(classname, ..., call = NULL)}

\texttt{new\_callback\_class(}
\begin{verbatim}
  classname,
  ..., 
  on\_epoch\_begin = NULL, 
  on\_epoch\_end = NULL, 
  on\_train\_begin = NULL, 
  on\_train\_end = NULL, 
  on\_batch\_begin = NULL, 
  on\_batch\_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, 
  on\_train\_batch\_begin = NULL, 
  on\_train\_batch\_end = NULL 
\end{verbatim}
\texttt{)}

Metric

```python
new_model_class(
    classname,
    ...
    initialize = NULL,
    call = NULL,
    train_step = NULL,
    predict_step = NULL,
    test_step = NULL,
    compute_loss = NULL,
    compute_metrics = NULL
)
```

```python
new_layer_class(
    classname,
    ...
    initialize = NULL,
    build = NULL,
    call = NULL,
    get_config = NULL
)
```

Arguments

- **x**: A function that should be converted to an active property of the class type.
- **classname**: The classname as a string. Convention is for the classname to be a CamelCase version of the constructor.
- **...**: Additional fields and methods for the new type.
- **initialize, build, call, get_config, on_epoch_begin, on_epoch_end, on_train_begin, on_train_end, on_batch_begin, ... on_train_batch_end, update_state, result, train_step, predict_step, test_step, compute_loss, compute_metrics**: Optional methods that can be overridden.

Details

`mark_active()` is a decorator that can be used to indicate functions that should become active properties of the class instances.

Value

A new class generator object that inherits from the appropriate Keras base class.

---

<table>
<thead>
<tr>
<th>Metric</th>
<th>Metric</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Description

A `Metric` object encapsulates metric logic and state that can be used to track model performance during training. It is what is returned by the family of metric functions that start with prefix `metric_*`. 
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$add_weight(...)
  ```
- **update_state()**: Has all updates to the state variables like:
  ```r
  self$var$assign_add(...)
  ```
- **result()**: Computes and returns a value for the metric from the state variables.

Example custom metric subclass:

```r
metric_binary_true_positives <- new_metric_class(
  classname = "BinaryTruePositives",
  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()))

The same metric_binary_true_positives could be built with %py_class% like this:

metric_binary_true_positives(keras$metrics$Metric) %py_class% {
  initialize <- <same-as-above>,
  update_state <- <same-as-above>,
  result <- <same-as-above>
}

---

**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.
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()`

---

**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
)
```
Arguments

Passed on to the underlying metric. Used for forwards and backwards compatibility.

num_thresholds (Optional) Defaults to 200. The number of thresholds to use when discretizing the roc curve. Values must be > 1.

curve (Optional) Specifies the name of the curve to be computed, 'ROC' (default) or 'PR' for the Precision-Recall-curve.

summation_method (Optional) Specifies the Riemann summation method used. 'interpolation' (default) applies mid-point summation scheme for ROC. For PR-AUC, interpolates (true/false) positives but not the ratio that is precision (see Davis & Goadrich 2006 for details); 'minoring' applies left summation for increasing intervals and right summation for decreasing intervals; 'majoring' does the opposite.

thresholds (Optional) A list of floating point values to use as the thresholds for discretizing the curve. If set, the num_thresholds parameter is ignored. Values should be in [0, 1]. Endpoint thresholds equal to -epsilon, 1+epsilon for a small positive epsilon value will be automatically included with these to correctly handle predictions equal to exactly 0 or 1.

multi_label boolean indicating whether multilabel data should be treated as such, wherein AUC is computed separately for each label and then averaged across labels, or (when FALSE) if the data should be flattened into a single label before AUC computation. In the latter case, when multilabel data is passed to AUC, each label-prediction pair is treated as an individual data point. Should be set to FALSE for multi-class data.

num_labels (Optional) The number of labels, used when multi_label is TRUE. If num_labels is not specified, then state variables get created on the first call to update_state.

label_weights (Optional) list, array, or tensor of non-negative weights used to compute AUCs for multilabel data. When multi_label is TRUE, the weights are applied to the individual label AUCs when they are averaged to produce the multi-label AUC. When it’s FALSE, they are used to weight the individual label predictions in computing the confusion matrix on the flattened data. Note that this is unlike class_weights in that class_weights weights the example depending on the value of its label, whereas label_weights depends only on the index of that label before flattening; therefore label_weights should not be used for multi-class data.

from_logits boolean indicating whether the predictions (y_pred in update_state) are probabilities or sigmoid logits. As a rule of thumb, when using a keras loss, the from_logits constructor argument of the loss should match the AUC from_logits constructor argument.

name (Optional) string name of the metric instance.

dtype (Optional) data type of the metric result.

Details

The AUC (Area under the curve) of the ROC (Receiver operating characteristic; default) or PR (Precision Recall) curves are quality measures of binary classifiers. Unlike the accuracy, and like cross-entropy losses, ROC-AUC and PR-AUC evaluate all the operational points of a model.
This class approximates AUCs using a Riemann sum. During the metric accumulation phrase, predictions are accumulated within predefined buckets by value. The AUC is then computed by interpolating per-bucket averages. These buckets define the evaluated operational points.

This metric creates four local variables, `true_positives`, `true_negatives`, `false_positives` and `false_negatives` that are used to compute the AUC. To discretize the AUC curve, a linearly spaced set of thresholds is used to compute pairs of recall and precision values. The area under the ROC-curve is therefore computed using the height of the recall values by the false positive rate, while the area under the PR-curve is the computed using the height of the precision values by the recall.

This value is ultimately returned as `auc`, an idempotent operation that computes the area under a discretized curve of precision versus recall values (computed using the aforementioned variables). The `num_thresholds` variable controls the degree of discretization with larger numbers of thresholds more closely approximating the true AUC. The quality of the approximation may vary dramatically depending on `num_thresholds`. The thresholds parameter can be used to manually specify thresholds which split the predictions more evenly.

For a best approximation of the real AUC, predictions should be distributed approximately uniformly in the range `[0, 1]` (if `from_logits=FALSE`). The quality of the AUC approximation may be poor if this is not the case. Setting `summation_method` to 'minoring' or 'majoring' can help quantify the error in the approximation by providing lower or upper bound estimate of the AUC.

If `sample_weight` is NULL, weights default to 1. Use `sample_weight` of 0 to mask values.

**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_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_binary_accuracy**

*Calculates how often predictions match binary labels*

**Description**

Calculates how often predictions match binary labels
metric_binary_accuracy

Usage

```r
code
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()`. 
**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.
Details

You can provide logits of classes as \( y_{\text{pred}} \), since argmax of logits and probabilities are same.

This metric creates two local variables, \( \text{total} \) and \( \text{count} \) that are used to compute the frequency with which \( y_{\text{pred}} \) matches \( y_{\text{true}} \). This frequency is ultimately returned as categorical accuracy: an idempotent operation that simply divides total by count.

\( y_{\text{pred}} \) and \( y_{\text{true}} \) should be passed in as vectors of probabilities, rather than as labels. If necessary, use \text{tf.one_hot} to expand \( y_{\text{true}} \) as a vector.

If \text{sample_weight} is NULL, weights default to 1. Use \text{sample_weight} of 0 to mask values.

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}() 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: \text{custom_metric()}, \text{metric_accuracy()}, \text{metric_auc()}, \text{metric_binary_accuracy()}, \text{metric_binary_crossentropy()}, \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_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()}

\[ \text{metric\_categorical\_crossentropy} \]

*Computes the crossentropy metric between the labels and predictions*

Description

Computes the crossentropy metric between the labels and predictions

Usage

\[
\text{metric\_categorical\_crossentropy}( \\
y_{\text{true}}, \\
y_{\text{pred}}, \\
\text{from\_logits} = \text{FALSE},
\]

---

\[
\text{metric\_categorical\_crossentropy} \]

*Computes the crossentropy metric between the labels and predictions*
label_smoothing = 0,
axis = -1L,
...,
name = "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.
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 compati-
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 multiple label classes (2 or more).
Here we assume that labels are given as a one_hot representation. eg., When labels values are c(2, 
0, 1):

    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(),
Computes the categorical hinge metric between y_true and y_pred

Description
Computes the categorical hinge metric between y_true and y_pred

Usage
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_categorical_hinge(), metric_categorical_iou(), metric_categorical_loss(), metric_categorical_mean_absolute_error(), metric_categorical_mean_squared_error(), metric_categorical_mean_squared_logarithmic_error(), metric_categorical_mean_tensor(), metric_categorical_mean_wrapper(), metric_categorical_mean(), metric_categorical_poisson(), metric_categorical_precision_at_recall(), metric_categorical_precision(), metric_categorical_recall_at_precision(), metric_categorical_recall(), metric_categorical_root_mean_squared_error(), metric_categorical_sensitivity_at_specificity(), metric_categorical_sparse_categorical_accuracy(), metric_categorical_sparse_categorical_crossentropy(), metric_categorical_sparse_top_k_categorical_accuracy(), metric_categorical_specificity_at_sensitivity(), metric_categorical_squared_hinge(), metric_categorical_sum(), metric_categorical_top_k_categorical_accuracy(), metric_categorical_true_negatives(), metric_categorical_true_positives()
metric_cosine_similarity

Computes the cosine similarity between the labels and predictions

Description

Computes the cosine similarity between the labels and predictions

Usage

```
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

\[
\text{cosine similarity} = \frac{(a \cdot b)}{\|a\| \|b\|}
\]

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()`.
metric_false_negatives

 Calculates the number of false negatives

Description

Calculates the number of false negatives

Usage

metric_false_negatives(..., 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 negatives. This metric creates
one local variable, accumulator that is used to keep track of the number of false negatives.
If sample_weight is NULL, weights default to 1. Use sample_weight of 0 to mask values.

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_false_positives

Calculates the number of false positives

Description

Calculates the number of false positives

Usage

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.
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

loss = tf$reduce_mean(tf$maximum(1 - y_true * y_pred, 0L), 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.
Computes Kullback-Leibler divergence

Description
Computes Kullback-Leibler divergence

Usage
```
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
```
metric = y_true * log(y_true / y_pred)
```

See: https://en.wikipedia.org/wiki/Kullback%E2%80%93Leibler_divergence
**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_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_specificity_at_sensitivity()`, `metric_squared_hinge()`, `metric_sum()`, `metric_top_k_categorical_accuracy()`, `metric_true_negatives()`, `metric_true_positives()`

---

**Description**

\[ \text{logcosh} = \log((\exp(x) + \exp(-x))/2) \]

where \( x \) is the error \( (y_{\text{pred}} - y_{\text{true}}) \)

**Usage**

`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.
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():

```r
model$add_metric(metric_mean(name='mean_1')(outputs))
model %>% compile(optimizer='sgd', loss='mse')
```

Example standalone usage:

```r
m <- metric_mean()
m$update_state(c(1, 3, 5, 7))
m$result()

m$reset_state()
m$update_state(c(1, 3, 5, 7), sample_weight=c(1, 1, 0, 0))
m$result()
as.numeric(m$result())
```

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_ranking()`, `metric_categorical_squared_hinge()`, `metric_categorical_top_k_categorical_accuracy()`, `metric_categorical_weighted_crossentropy()`, `metric_categorical_weighted_hinge()`, `metric_categorical_weighted_ranking()`, `metric_categorical_weighted_squared_hinge()`, `metric_cosine_similarity()`, `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_weighted_crossentropy()`, `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_sparse_weighted_crossentropy()`, `metric_sparse_weighted_hinge()`, `metric_sparse_weighted_ranking()`, `metric_sparse_weighted_squared_hinge()`, `metric_sum()`, `metric_top_k_categorical_accuracy()`, `metric_true_negatives()`, `metric_true_positives()`

---

**metric_mean_absolute_error**

*Computes the mean absolute error between the labels and predictions*

**Description**

Computes the mean absolute error between the labels and predictions

**Usage**

```r
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**

```
loss = mean(abs(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_iou()`, `metric_categorical_crossentropy()`, `metric_false_negatives()`, `metric_false_positives()`, `metric_hinge()`, `metric_binary_crossentropy()`, `metric_kullback_leibler_divergence()`, `metric_logcosh_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_precision_at_specificity()`, `metric_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_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_precision_at_specificity()`, `metric_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_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_precision_at_specificity()`, `metric_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_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_precision_at_specificity()`, `metric_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_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_precision_at_specificity()`, `metric_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_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_precision_at_specificity()`, `metric_accuracy()`.
metric_mean_absolute_percentage_error

Usage

```python
def metric_mean_absolute_percentage_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

```python
loss = 100 * mean(abs((y_true - y_pred) / y_true), 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_cosine_similarity()`, `metric_false_negatives()`, `metric_false_positives()`, `metric_hinge()`, `metric_kullback_leibler_divergence()`, `metric_logcosh_error()`, `metric_mean_absolute_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_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 dim c(num_classes, num_classes) 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 sample_weight and the metric is then calculated from it.

If sample_weight is NULL, weights default to 1. Use sample_weight of 0 to mask values.

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_relative_error(), metric_mean_squared_error(), metric_mean_squared_logarithmic_error(), metric_mean_tensor().
metric_mean_relative_error

Computes the mean relative error by normalizing with the given values

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$update_state(c(1, 3, 2, 3), c(2, 4, 6, 8))
# result = mean(c(1, 1, 4, 5) / c(1, 3, 2, 3)) = mean(c(1, 1/3, 2, 5/3))
# = 5/4 = 1.25
m$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,
    ...,  # Passed on to the underlying metric. Used for forwards and backwards compatibility.
    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.

```r
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_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_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_mean_squared_logarithmic_error**

*Computes the mean squared logarithmic error*

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.
**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) \(\text{Metric}\) instance is returned. The \(\text{Metric}\) 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_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_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_mean_tensor

*Computes the element-wise (weighted) mean of the given tensors*

---

**Description**

Computes the element-wise (weighted) mean of the given tensors.

**Usage**

```
metric_mean_tensor(..., shape = NULL, name = NULL, dtype = NULL)
```

**Arguments**

- `...`: Passed on to the underlying metric. Used for forwards and backwards compatibility.
- `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 update_state.
- `name` (Optional) string name of the metric instance.
- `dtype` (Optional) data type of the metric result.
Details

MeanTensor returns a tensor with the same shape of the input tensors. The mean value is updated by keeping local variables total and count. The total tracks the sum of the weighted values, and count stores the sum of the weighted counts.

Value

A (subclassped) 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_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()
**Details**

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
code
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_categorical_hinge()`, `metric_categorical_hinge()`, `metric_categorical_hinge()`, `metric_categorical_hinge()`, `metric_crossentropy()`, `metric_accuracy()`, `metric_auc()`, `metric_binary_accuracy()`, `metric_binary_crossentropy()`, `metric_categorical_accuracy()`, `metric_categorical_crossentropy()`, `metric_categorical_hinge()`, `metric_crossentropy()`. 

**Description**

The Poisson metric between `y_true` and `y_pred` is computed as:

\[ \text{metric} = y_{pred} - y_{true} \times \log(y_{pred}) \]

**Usage**

```r
code
metric_poisson(y_true, y_pred, ..., name = "poisson", dtye = 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.

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_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

```
metric_precision()
    ...
    thresholds = NULL,
    top_k = NULL,
    class_id = NULL,
    name = NULL,
    dtype = NULL
)```
**Arguments**

- **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 \([0, \text{num_classes})\), where \(\text{num_classes}\) 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.
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
metric_recall(
  ...,
  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 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.

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_squared_hinge(), metric_sum(), metric_top_k_categorical_accuracy(), metric_true_negatives(), metric_true_positives()
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_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_sparse_top_k_categorical_accuracy()`, `metric_specificity_at_sensitivity()`, `metric_squared_hinge()`, `metric_sum()`, `metric_top_k_categorical_accuracy()`, `metric_true_negatives()`, `metric_true_positives()`

---

**metric_recall_at_precision**

*Computes best recall where precision is >= specified value*

Description

Computes best recall where precision is >= specified value

Usage

```r
metric_recall_at_precision(
  ..., precision,
  num_thresholds = 200L,
  class_id = NULL,
  name = NULL,
)```
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, 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

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_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_recall(), metric_root_mean_squared_error(), metric_sensitivity_at_specificity(), metric_squared_hinge().
metric_root_mean_squared_error

Computes root mean squared error metric between y_true and y_pred

Description

Computes root mean squared error metric between y_true and y_pred

Usage

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_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_spherical_categorical_accuracy(), metric_spherical_categorical_crossentropy(), metric_spherical_hinge(), metric_sum(), metric_top_k_categorical_accuracy(), metric_true_negatives(), metric_true_positives()
metric_sensitivity_at_specificity

Computes best sensitivity where specificity is >= specified value

Description

The sensitivity at a given specificity.

Usage

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].
um_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

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_sparse_categorical_accuracy(), metric_sparse_categorical_crossentropy(), metric_sparse_top_k_categorical_accuracy(), metric_specificity_at_sensitivity(), metric_squared_hinge(), metric_sum(), metric_top_k_categorical_accuracy(), metric_true_negatives(), metric_true_positives()

metric_sparse_categorical_accuracy

Calculates how often predictions match integer labels

Description

Calculates how often predictions match integer labels

Usage

metric_sparse_categorical_accuracy(
  y_true,
  y_pred,
  ...,
  name = "sparse_categorical_accuracy",
  dtype = NULL
)

Arguments

y_true Tensor of true targets.
y_pred Tensor of predicted targets.
... 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

metric_sparse_categorical_crossentropy(
  y_true,
  y_pred,
  from_logits = FALSE,
  axis = -1L,
)
..., 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,
**metric_sparse_top_k_categorical_accuracy**

*Computes how often integer targets are in the top K predictions*

**Description**

Computes how often integer targets are in the top K predictions

**Usage**

```
metric_sparse_top_k_categorical_accuracy(
    y_true,
    y_pred,
    k = 5L,
    ...,
    name = "sparse_top_k_categorical_accuracy",
    dtype = NULL
)
```

**Arguments**

- **y_true**: Tensor of true targets.
- **y_pred**: Tensor of predicted targets.
- **k**: (Optional) Number of top elements to look at for computing accuracy. Defaults to 5.
- **...**: 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 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 specificity at the given sensitivity. The threshold for the given sensitivity value is computed and used to evaluate the corresponding specificity.

If `sample_weight` is NULL, weights default to 1. Use `sample_weight` of 0 to mask values.

If `class_id` is specified, we calculate precision by considering only the entries in the batch for which `class_id` is above the threshold predictions, and computing the fraction of them for which `class_id` is indeed a correct label.

For additional information about specificity and sensitivity, see the following.

**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)`
**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_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_sum()`, `metric_top_k_categorical_accuracy()`, `metric_true_negatives()`, `metric_true_positives()`

---

**metric_sum**

Computes the (weighted) sum of the given values

**Description**

Computes the (weighted) sum of the given values

**Usage**

```
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.
**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_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
)
```
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.

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_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**

Calculates the number of true positives

### Description

Calculates the number of true positives

### Usage

```
metric_true_positives(..., thresholds = NULL, name = NULL, dtype = NULL)
```
model_from_saved_model

**Description**

Load a Keras model from the Saved Model format

**Usage**

```r
model_from_saved_model(saved_model_path, custom_objects = NULL)
```
**model_to_json**

**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)`
- `model_from_json(json, 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()`
new_learning_rate_schedule_class

Description

Create a new learning rate schedule type

Usage

new_learning_rate_schedule_class(
    classname,
    ...,
    initialize = NULL,
    call,
    get_config = NULL
)
normalize

Arguments

classname string
... methods and properties of the schedule class
initialize, get_config
Additional recommended methods to implement.
  • https://www.tensorflow.org/api_docs/python/tf/keras/optimizers/schedules/LearningRateSchedule
call function which takes a step argument (scalar integer tensor, the current training step count, and returns the new learning rate). For tracking additional state, objects self and private are automatically injected into the scope of the function.

Value

A LearningRateSchedule class generator.

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.
Adadelta optimizer as described in ADADELTA: An Adaptive Learning Rate Method.

Usage

```r
optimizer_adadelta(
  learning_rate = 1,
  rho = 0.95,
  epsilon = NULL,
  decay = 0,
  clipnorm = NULL,
  clipvalue = NULL,
...
)
```

Arguments

- `learning_rate` float $\geq 0$. Learning rate.
- `rho` float $\geq 0$. Decay factor.
- `epsilon` float $\geq 0$. Fuzz factor. If `NULL`, defaults to `k_epsilon()`.
- `decay` float $\geq 0$. 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.

See Also

Other optimizers: `optimizer_adagrad()`, `optimizer_adamax()`, `optimizer_adam()`, `optimizer_nadam()`, `optimizer_rmsprop()`, `optimizer_sgd()`
optimizer_adagrad

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

- `learning_rate` float >= 0. Learning rate.
- `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.

See Also

Other optimizers: `optimizer_adadelta()`, `optimizer_adamax()`, `optimizer_adam()`, `optimizer_nadam()`, `optimizer_rmsprop()`, `optimizer_sgd()`
optimizer_adam  

---

**Optimizer** Adam optimizer

---

**Description**

Adam optimizer as described in Adam - A Method for Stochastic Optimization.

**Usage**

```r
optimizer_adam(
  learning_rate = 0.001,  # float >= 0. Learning rate.
  beta_1 = 0.9,           # The exponential decay rate for the 1st moment estimates. float, 0 < beta < 1. Generally close to 1.
  beta_2 = 0.999,         # The exponential decay rate for the 2nd moment estimates. float, 0 < beta < 1. Generally close to 1.
  epsilon = NULL,         # float >= 0. Fuzz factor. If NULL, defaults to \( k_{\text{epsilon}} \).
  decay = 0,              # float >= 0. Learning rate decay over each update.
  amsgrad = FALSE,        # Whether to apply the AMSGrad variant of this algorithm from the paper "On the Convergence of Adam and Beyond".
  clipnorm = NULL,        # Gradients will be clipped when their L2 norm exceeds this value.
  clipvalue = NULL,       # Gradients will be clipped when their absolute value exceeds this value.
  ...                     # Unused, present only for backwards compatibility
)
```

**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 \( k_{\text{epsilon}} \).
- `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.
optimizer_adamax

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`  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().
- `schedule_decay`  Schedule deacy.
- `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.

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()
optimizer_sgd

---

Stochastic gradient descent optimizer

Description

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.engine.training.Model

**Plot a Keras model**

**Description**

Plot a Keras model

**Usage**

```r
## S3 method for class 'keras.engine.training.Model'
plot(
  x,
  show_shapes = FALSE,
  show_dtype = FALSE,
  show_layer_names = TRUE,
  ..., 
  rankdir = "TB",
  expand_nested = FALSE,
  dpi = 96,
  layer_range = NULL,
  show_layer_activations = FALSE,
  to_file = NULL
)
```

**Arguments**

- `x` A Keras model instance
- `show_shapes` whether to display shape information.
- `show_dtype` whether to display layer dtypes.
- `show_layer_names` whether to display layer names.
- `...` passed on to `keras$utils$plot_model()`. Used for forward and backward compatibility.
- `rankdir` a string specifying the format of the plot: 'TB' creates a vertical plot; 'LR' creates a horizontal plot. (argument passed to PyDot)
- `expand_nested` Whether to expand nested models into clusters.
- `dpi` Dots per inch. Increase this value if the image text appears excessively pixelated.

**See Also**

Other text preprocessing: `make_sampling_table()`, `skipgrams()`, `text_hashing_trick()`, `text_one_hot()`, `text_to_word_sequence()`
**layer_range**

A list containing two character strings, which is the starting layer name and ending layer name (both inclusive) indicating the range of layers for which the plot will be generated. It also accepts regex patterns instead of exact name. In such case, start predicate will be the first element it matches to `layer_range[1]` and the end predicate will be the last element it matches to `layer_range[2]`. By default NULL which considers all layers of model. Note that you must pass range such that the resultant subgraph must be complete.

**show_layer_activations**

Display layer activations (only for layers that have an activation property).

**to_file**

File name of the plot image. If NULL (the default), the model is drawn on the default graphics device. Otherwise, a file is saved.

**Value**

Nothing, called for it’s side effects.

**Raises**

ValueError: if `plot_model` is called before the model is built, unless a `input_shape` argument was supplied to `keras_model_sequential()`.

**Requirements**

This function requires pydot and graphviz. pydot is by default installed by `install_keras()`, but if you installed tensorflow by other means, you can install pydot directly with:

```r
reticulate::py_install("pydot", pip = TRUE)
```

In a conda environment, you can install graphviz with:

```r
reticulate::conda_install(packages = "graphviz")
# Restart the R session after install.
```

Otherwise you can install graphviz from here: [https://graphviz.gitlab.io/download/](https://graphviz.gitlab.io/download/)

---

**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.

---

### 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
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 = "auto",
  steps = NULL,
  callbacks = NULL,
  ...
)
```

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, 1, 2, or "auto". "auto" defaults to 1 for most cases and defaults to verbose=2 when used with ParameterServerStrategy or with interactive logging disabled.
- **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

predict_on_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_generator(), predict_on_batch(), predict_proba(), summary.keras.engine.training.Model(), train_on_batch()

---

**predict_on_batch**

Returns predictions for a single batch of samples.

Description

Returns predictions for a single batch of samples.

Usage

```python
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

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_proba(), summary.keras.engine.training.Model(), train_on_batch()
**regularizer_l1**  
*L1 and L2 regularization*

**Description**

L1 and L2 regularization

**Usage**

```python
regularizer_l1(l = 0.01)
regularizer_l2(l = 0.01)
regularizer_l1_l2(l1 = 0.01, l2 = 0.01)
```

**Arguments**

- `l`  
  Regularization factor.
- `l1`  
  L1 regularization factor.
- `l2`  
  L2 regularization factor.

**regularizer_orthogonal**

*A regularizer that encourages input vectors to be orthogonal to each other*

**Description**

A regularizer that encourages input vectors to be orthogonal to each other

**Usage**

```python
regularizer_orthogonal(factor = 0.01, mode = "rows", ...)
```

**Arguments**

- `factor`  
  Float. The regularization factor. The regularization penalty will be proportional to `factor` times the mean of the dot products between the L2-normalized rows (if `mode="rows"`, or columns if `mode="columns"`) of the inputs, excluding the product of each row/column with itself. Defaults to 0.01.
- `mode`  
  String, one of "rows", "columns". Defaults to "rows". In rows mode, the regularization effect seeks to make the rows of the input orthogonal to each other. In columns mode, it seeks to make the columns of the input orthogonal to each other.
For backwards and forwards compatibility

```r
layer <- layer_dense(
  units = 4,
  kernel_regularizer = regularizer_orthogonal(factor = 0.01))
```

Details

It can be applied to either the rows of a matrix (mode="rows") or its columns (mode="columns"). When applied to a Dense kernel of shape (input_dim, units), rows mode will seek to make the feature vectors (i.e. the basis of the output space) orthogonal to each other.

See Also

- [https://www.tensorflow.org/api_docs/python/tf/keras/regularizers/OrthogonalRegularizer](https://www.tensorflow.org/api_docs/python/tf/keras/regularizers/OrthogonalRegularizer)

---

**reset_states**

*Reset the states for a layer*

Description

Reset the states for a layer

Usage

```r
reset_states(object)
```

Arguments

- `object` Model or layer object

See Also

Other layer methods: `count_params()`, `get_config()`, `get_input_at()`, `get_weights()`
save_model_hdf5  

Save/Load models using HDF5 files

Description

Save/Load models using HDF5 files

Usage

```r
save_model_hdf5(object, filepath, overwrite = TRUE, include_optimizer = TRUE)

load_model_hdf5(filepath, custom_objects = NULL, compile = TRUE)
```

Arguments

- `object`: Model object to save
- `filepath`: File path
- `overwrite`: Overwrite existing file if necessary
- `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.

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 re-instantiated 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.
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)
```

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_tf()`, `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

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()`
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)
```

**Arguments**

- **object**: Text tokenizer fit with `fit_text_tokenizer()`
- **filename**: File to save/load

**Details**

You should always use the same text tokenizer for training and prediction. In many cases however prediction will occur in another session with a version of the model loaded via `load_model_hdf5()`.

In this case you need to save the text tokenizer object after training and then reload it prior to prediction.

**See Also**

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

**Examples**

```r
## Not run:
# vectorize texts then save for use in prediction
tokenizer <- text_tokenizer(num_words = 10000) #>
fit_text_tokenizer(tokenizer, texts)
save_text_tokenizer(tokenizer, "tokenizer")

# (train model, etc.)

# ...later in another session
tokenizer <- load_text_tokenizer("tokenizer")
```
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  
  Tokenizer

sequences  
  List of sequences (a sequence is a list of integer word indices).

mode  
  one of "binary", "count", "tfidf", "freq".

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

```r
sequential_model_input_layer(
  input_shape = NULL,
  batch_size = NULL,
  dtype = NULL,
  input_tensor = NULL,
  sparse = NULL,
  name = NULL,
  ragged = NULL,
  type_spec = NULL,
  ...
)
```

Arguments

- **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.
- **...**: 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**

```r
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.

**See Also**

Other model persistence: `get_weights()`, `model_to_json()`, `model_to_yaml()`, `save_model_hdf5()`, `save_model_tf()`, `save_model_weights_hdf5()`
**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 $[i-window_size, i+window_size+1]$
- `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: [word_index, other_word_index].
• labels is an integer vector of 0 and 1, where 1 indicates that other_word_index was found in the same window as word_index, and 0 indicates that 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

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".
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).

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
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_matrix(), texts_to_sequences_generator()
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_dataset_from_directory

*Generate a tf.data.Dataset from text files in a directory*

Description

Generate a tf.data.Dataset from text files in a directory

Usage

text_dataset_from_directory(
  directory,
  labels = "inferred",
  label_mode = "int",
  class_names = NULL,
  batch_size = 32L,
  max_length = NULL,
  shuffle = TRUE,
  seed = NULL,
  validation_split = NULL,
  subset = NULL,
  follow_links = FALSE,
  ...
)

Arguments

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

labels       Either "inferred" (labels are generated from the directory structure), NULL (no labels), or a list of integer labels of the same size as the number of text files found in the directory. Labels should be sorted according to the alphanumeric order of the text file paths (obtained via `os.walk(directory)` in Python).
label_mode  

- 'int': means that the labels are encoded as integers (e.g. for `sparse_categorical_crossentropy` loss).
- 'categorical' means that the labels are encoded as a categorical vector (e.g. for `categorical_crossentropy` loss).
- 'binary' means that the labels (there can be only 2) are encoded as float32 scalars with values 0 or 1 (e.g. for `binary_crossentropy`).
- NULL (no labels).

class_names  Only valid if labels is "inferred". This is the explicit list of class names (must match names of subdirectories). Used to control the order of the classes (otherwise alphanumerical order is used).

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

max_length  Maximum size of a text string. Texts longer than this will be truncated to max_length.

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.

follow_links  Whether to visits subdirectories pointed to by symlinks. Defaults to FALSE.

...  For future compatibility (unused presently).

Details

If your directory structure is:

```
main_directory/
  ...class_a/
  ......a_text_1.txt
  ......a_text_2.txt
  ...class_b/
  ......b_text_1.txt
  ......b_text_2.txt
```

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

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

See Also

- https://www.tensorflow.org/api_docs/python/tf/keras/utils/text_dataset_from_directory
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

```r
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 the Python hash() function. Otherwise 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()`
text_one_hot  

One-hot encode a text into a list of word indexes in a vocabulary of size n.

Description

One-hot encode a text into a list of word indexes in a vocabulary of size n.

Usage

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).
- **text**: for compatibility purpose. use input_text instead.

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

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.
- **split**: character or string to use for token splitting.
- **char_level**: if TRUE, every character will be treated as a token
- **oov_token**: NULL or string. If given, it will be added to `word_index` and used to replace out-of-vocabulary words during text_to_sequence calls.

Details

By default, all punctuation is removed, turning the texts into space-separated sequences of words (words maybe include the ' character). These sequences are then split into lists of tokens. They will then be indexed or vectorized. 0 is a reserved index that won’t be assigned to any word.

Attributes

The tokenizer object has the following attributes:

- **word_counts**: named list mapping words to the number of times they appeared on during fit. Only set after `fit_text_tokenizer()` is called on the tokenizer.
- **word_docs**: named list mapping words to the number of documents/texts they appeared on during fit. Only set after `fit_text_tokenizer()` is called on the tokenizer.
text_to_word_sequence

Convert text to a sequence of words (or tokens).

Description

Convert text to a sequence of words (or tokens).

Usage

text_to_word_sequence(
  text,
  filters = "!"#$%&()*+,-./:;<=>?@\^[\]\\{\|}~\t\n",
  lower = TRUE,
  split = " "
)

Arguments

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_dataset_from_array

Creates a dataset of sliding windows over a timeseries provided as array

Description

Creates a dataset of sliding windows over a timeseries provided as array

Usage

timeseries_dataset_from_array(
  data,
  targets,
  sequence_length,
  sequence_stride = 1L,
  sampling_rate = 1L,
  batch_size = 128L,
  shuffle = FALSE,
  ...,
  seed = NULL,
  start_index = NULL,
  end_index = NULL
)

Arguments

data array or eager tensor containing consecutive data points (timesteps). The first axis is expected to be the time dimension.

targets Targets corresponding to timesteps in data. targets[i] should be the target corresponding to the window that starts at index i (see example 2 below). Pass NULL if you don’t have target data (in this case the dataset will only yield the input data).

sequence_length Length of the output sequences (in number of timesteps).

sequence_stride Period between successive output sequences. For stride s, output samples would start at index data[i], data[i + s], data[i + (2 * s)], etc.

sampling_rate Period between successive individual timesteps within sequences. For rate r, timesteps data[i], data[i + r], ... data[i + sequence_length] are used for create a sample sequence.

batch_size Number of timeseries samples in each batch (except maybe the last one).

shuffle Whether to shuffle output samples, or instead draw them in chronological order.

... For backwards and forwards compatibility, ignored presently.

seed Optional int; random seed for shuffling.
timeseries_dataset_from_array

\texttt{start\_index, end\_index}

Optional int (1 based); data points earlier than \texttt{start\_index} or later than \texttt{end\_index} will not be used in the output sequences. This is useful to reserve part of the data for test or validation.

Details

This function takes in a sequence of data-points gathered at equal intervals, along with time series parameters such as length of the sequences/windows, spacing between two sequence/windows, etc., to produce batches of timeseries inputs and targets.

Value

A \texttt{tf.data.Dataset} instance. If \texttt{targets} was passed, the dataset yields batches of two items: (\texttt{batch\_of\_sequences, batch\_of\_targets}). If not, the dataset yields only \texttt{batch\_of\_sequences}.

Example 1

Consider indices 0:99. With \texttt{sequence\_length=10, sampling\_rate=2, sequence\_stride=3, shuffle=FALSE}, the dataset will yield batches of sequences composed of the following indices:

First sequence: 0 2 4 6 8 10 12 14 16 18
Second sequence: 3 5 7 9 11 13 15 17 19 21
Third sequence: 6 8 10 12 14 16 18 20 22 24
... Last sequence: 78 80 82 84 86 88 90 92 94 96

In this case the last 3 data points are discarded since no full sequence can be generated to include them (the next sequence would have started at index 81, and thus its last step would have gone over 99).

Example 2

Temporal regression.

Consider an array \texttt{data} of scalar values, of shape (\texttt{steps}). To generate a dataset that uses the past 10 timesteps to predict the next timestep, you would use:

```r
steps <- 100
# data is integer seq with some noise
data <- array(1:steps + abs(rnorm(steps, sd = .25)))
inputs_data <- head(data, -10)  # drop last 10
targets <- tail(data, -10)  # drop first 10
dataset <- timeseries_dataset_from_array(
  inputs_data, targets, sequence_length=10)
library(tfdatasets)
dataset_iterator <- as_iterator(dataset)
repeat {
  batch <- iter_next(dataset_iterator)
  if(is.null(batch)) break
```
Example 3

Temporal regression for many-to-many architectures.

Consider two arrays of scalar values \(X\) and \(Y\), both of shape \((100)\). The resulting dataset should consist of samples with 20 timestamps each. The samples should not overlap. To generate a dataset that uses the current timestamp to predict the corresponding target timestep, you would use:

```r
X <- seq(100)
Y <- X*2
sample_length <- 20
input_dataset <- timeseries_dataset_from_array(
  X, NULL, sequence_length=sample_length, sequence_stride=sample_length)
target_dataset <- timeseries_dataset_from_array(
  Y, NULL, sequence_length=sample_length, sequence_stride=sample_length)
```

```
library(tfdatasets)
dataset_iterator <-
  zip_datasets(input_dataset, target_dataset) %>%
  as_array_iterator()
while(!is.null(batch <- iter_next(dataset_iterator))) {
  c(inputs, targets) %<-% batch
  stopifnot(
    all.equal(inputs[1,], X[1:sample_length]),
    all.equal(targets[1,], Y[1:sample_length]),
    # second sample equals output timestamps 20-40
    all.equal(inputs[2,], X[(1:sample_length) + sample_length]),
    all.equal(targets[2,], Y[(1:sample_length) + sample_length])
  )
}
```

Example

```r
int_sequence <- seq(20)
```
timeseries_generator <- timeseries_dataset_from_array(
  data = head(int_sequence, -3), # drop last 3
  targets = tail(int_sequence, -3), # drop first 3
  sequence_length = 3,
  start_index = 3,
  end_index = 9,
  batch_size = 2
)

library(tfdatasets)
dummy_dataset_iterator <- as_array_iterator(dummy_dataset)

repeat {
  batch <- iter_next(dummy_dataset_iterator)
  if (is.null(batch)) # iterator exhausted
    break
  c(inputs, targets) %<-% batch
  for (r in 1:nrow(inputs))
    cat(sprintf("input: [ %s ] target: %s\n", 
      paste(inputs[r,], collapse = " "), targets[r]))
  cat("---------------------------\n") # demark batches
}

Will give output like:

input: [ 3 4 5 ] target: 6
input: [ 4 5 6 ] target: 7
---------------------------
input: [ 5 6 7 ] target: 8
input: [ 6 7 8 ] target: 9
---------------------------
input: [ 7 8 9 ] target: 10

See Also

- https://www.tensorflow.org/api_docs/python/tf/keras/utils/timeseries_dataset_from_array

---

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 create 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()`).

Description

This layer wrapper allows to apply a layer to every temporal slice of an input.
Usage

time_distributed(object, layer, ...)

Arguments

object What to compose the new Layer instance with. Typically a Sequential model or a Tensor (e.g., as returned by layer_input()). The return value depends on object. If object is:

• missing or NULL, the Layer instance is returned.
• a Sequential model, the model with an additional layer is returned.
• a Tensor, the output tensor from layer_instance(object) is returned.

layer a tf.keras.layers.Layer instance.
... standard layer arguments.

Details

Every input should be at least 3D, and the dimension of index one of the first input will be considered to be the temporal dimension.

Consider a batch of 32 video samples, where each sample is a 128x128 RGB image with channels_last data format, across 10 timesteps. The batch input shape is (32, 10, 128, 128, 3).

You can then use TimeDistributed to apply the same Conv2D layer to each of the 10 timesteps, independently:

input <- layer_input(c(10, 128, 128, 3))
conv_layer <- layer_conv_2d(filters = 64, kernel_size = c(3, 3))
output <- input %>% time_distributed(conv_layer)
output$shape # TensorShape([None, 10, 126, 126, 64])

Because TimeDistributed applies the same instance of Conv2D to each of the timestamps, the same set of weights are used at each timestamp.

See Also

• https://www.tensorflow.org/api_docs/python/tf/keras/layers/TimeDistributed

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.
train_on_batch

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.

train_on_batch

Single gradient update or model evaluation over one batch of samples.

Description

Single gradient update or model evaluation over one batch of samples.

Usage

train_on_batch(object, x, y, class_weight = NULL, sample_weight = NULL)
test_on_batch(object, x, y, sample_weight = 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.
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")
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

## Not run:
# define custom metric
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)

### zip_lists

#### zip lists

**Description**

This is conceptually similar to `zip()` in Python, or R functions `purrr::transpose()` and `data.table::transpose()` (albeit, accepting elements in ... instead of a single list), with one crucial difference: if the provided objects are named, then matching is done by names, not positions.

**Usage**

```r
zip_lists(...)  
```

**Arguments**

```r
...  
```

R lists or atomic vectors, optionally named.

**Details**

All arguments supplied must be of the same length. If positional matching is required, then all arguments provided must be unnamed. If matching by names, then all arguments must have the same set of names, but they can be in different orders.

**Value**

A inverted list

**Examples**

```r
gradients <- list("grad_for_wt_1", "grad_for_wt_2", "grad_for_wt_3")
weights <- list("weight_1", "weight_2", "weight_3")
str(zip_lists(gradients, weights))
str(zip_lists(gradient = gradients, weight = weights))

names(gradients) <- names(weights) <- paste0("layer_", 1:3)
str(zip_lists(gradients, weights[c(3, 1, 2)]))
```
names(gradients) <- paste0("gradient_", 1:3)
try(zip_lists(gradients, weights)) # error, names don't match
# call unname directly for positional matching
zip_lists(unname(gradients), unname(weights))

---

**Description**

Make a python class constructor

**Usage**

```r
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.

**See Also**

[https://keras.rstudio.com/articles/new-guides/python_subclasses.html](https://keras.rstudio.com/articles/new-guides/python_subclasses.html)

**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!

# In addition to 'self', there is also 'private' available.
# This is an R environment unique to each class instance, where you can
# store objects that you don't want converted to Python, but still want
# available from methods. You can also assign methods to private, and
# 'self' and 'private' will be available in private methods.

MyClass %py_class% {
  initialize <- function(x) {
    print("Hi from MyClass$initialize()!")
    private$y <- paste("A Private field:", x)
  }

  get_private_field <- function() {
    private$y
  }

  private$a_private_method <- function() {
    cat("a_private_method() was called.\n")
    cat("private$y is ", sQuote(private$y), "\n")
  }

  call_private_method <- function()
    private$a_private_method()
}

inst1 <- MyClass(1)
inst2 <- MyClass(2)
inst1$get_private_field()
inst2$get_private_field()
inst1$call_private_method()
inst2$call_private_method()

## End(Not run)
Description

Make an Active Binding

Usage

sym %<-active% value

Arguments

sym symbol to bind
value A function to call when the value of sym is accessed.

Details

Active bindings defined in a %py_class% are converted to @property decorated methods.

Value

value, invisibly

See Also

makeActiveBinding()

Examples

set.seed(1234)
x %<-active% function(value) {
  message("Evaluating function of active binding")
  if(missing(value))
    runif(1)
  else
    message("Received: ", value)
}
x
x <- "foo"
x <- "foo"
x
rm(x) # cleanup
Index

* RNN cell layers
  layer_gru_cell, 284
  layer_lstm_cell, 303
  layer_simple_rnn_cell, 347
  layer_stacked_rnn_cells, 352

* activation layers
  layer_activation, 202
  layer_activation_elu, 203
  layer_activation_leaky_relu, 205
  layer_activation_parametric_relu, 206
  layer_activation_relu, 207
  layer_activation_selu, 209
  layer_activation_softermmax, 210
  layer_activation_thresholded_relu, 211

* attention layers
  layer_attention, 216

* callbacks
  callback_csv_logger, 40
  callback_early_stopping, 41
  callback_lambda, 42
  callback_learning_rate_scheduler, 43
  callback_model_checkpoint, 44
  callback_progbar_logger, 45
  callback_reduce_lr_on_plateau, 46
  callback_remote_monitor, 47
  callback_tensorboard, 48
  callback_terminate_on_naan, 49

* categorical features preprocessing layers
  layer_category_encoding, 225
  layer_hashing, 286
  layer_integer_lookup, 289
  layer_string_lookup, 353

* convolutional layers
  layer_conv_1d, 228
  layer_conv_1d_transpose, 231
  layer_conv_2d, 233
  layer_conv_2d_transpose, 236
  layer_conv_3d, 239
  layer_conv_3d_transpose, 242
  layer_conv_lstm_2d, 247
  layer_cropping_1d, 253
  layer_cropping_2d, 254
  layer_cropping_3d, 255
  layer_depthwise_conv_1d, 260
  layer_depthwise_conv_2d, 262
  layer_separable_conv_1d, 338
  layer_separable_conv_2d, 341
  layer_upsampling_1d, 360
  layer_upsampling_2d, 361
  layer_upsampling_3d, 362
  layer_zero_padding_1d, 364
  layer_zero_padding_2d, 365
  layer_zero_padding_3d, 366

* core layers
  layer_activation, 202
  layer_activity_regularization, 212
  layer_attention, 216
  layer_dense, 257
  layer_dense_features, 259
  layer_dropout, 267
  layer_flatten, 270
  layer_input, 288
  layer_lambda, 291
  layer_masking, 305
  layer_permute, 316
  layer_repeat_vector, 329
  layer_reshape, 331

* datasets
  dataset_boston_housing, 57
  dataset_cifar10, 58
  dataset_cifar100, 58
  dataset_fashion_mnist, 59
  dataset_imdb, 60
  dataset_mnist, 61
  dataset_reuters, 62
keras, 100

* dropout layers
  - layer_dropout, 267
  - layer.spatial_dropout_1d, 349
  - layer.spatial_dropout_2d, 350
  - layer.spatial_dropout_3d, 351

* image augmentation layers
  - layer.random_brightness, 317
  - layer.random_contrast, 318
  - layer.random_crop, 319
  - layer.random_flip, 321
  - layer.random_height, 322
  - layer.random_rotation, 323
  - layer.random_translation, 325
  - layer.random_width, 326
  - layer.random_zoom, 328

* image preprocessing layers
  - layer.center_crop, 226
  - layer.rescaling, 330
  - layer.resizing, 333

* image preprocessing
  - fit_image_data_generator, 67
  - flow_images_from_data, 68
  - flow_images_from_dataframe, 70
  - flow_images_from_directory, 72
  - image_load, 87
  - image_to_array, 88

* initializers
  - initializer.constant, 89
  - initializer.glorot_normal, 90
  - initializer.glorot_uniform, 90
  - initializer.he_normal, 91
  - initializer.he_uniform, 92
  - initializer.identity, 92
  - initializer.lecun_normal, 93
  - initializer.lecun_uniform, 94
  - initializer.ones, 94
  - initializer.orthogonal, 95
  - initializer.random_normal, 95
  - initializer.random_uniform, 96
  - initializer.truncated_normal, 96
  - initializer.variance_scaling, 97
  - initializer.zeros, 98

* layer methods
  - count_params, 54
  - get_config, 76
  - get_input_at, 78
  - get_weights, 80

reset_states, 448

* layer wrappers
  - bidirectional, 39
  - time_distributed, 471

* locally connected layers
  - layer_locally_connected_1d, 294
  - layer_locally_connected_2d, 297

* merge layers
  - layer_average, 217
  - layer_concatenate, 227
  - layer_dot, 266
  - layer_maximum, 306
  - layer_minimum, 311
  - layer_multiply, 312
  - layer_subtract, 355

* merging layers
  - layer_add, 214

* metrics
  - custom_metric, 56
  - metric_accuracy, 385
  - metric_auc, 386
  - metric_binary_accuracy, 388
  - metric_binary_crossentropy, 390
  - metric_categorical_accuracy, 391
  - metric_categorical_crossentropy, 392
  - metric_categorical_hinge, 394
  - metric_cosine_similarity, 395
  - metric_false_negatives, 396
  - metric_false_positives, 397
  - metric_hinge, 398
  - metric_kullback_leibler_divergence, 399
  - metric_logcosh_error, 400
  - metric_mean, 401
  - metric_mean_absolute_error, 402
  - metric_mean_absolute_percentage_error, 403
  - metric_mean_iou, 405
  - metric_mean_relative_error, 406
  - metric_mean_squared_error, 407
  - metric_mean_squared_logarithmic_error, 408
  - metric_mean_tensor, 409
  - metric_mean_wrapper, 410
  - metric_poisson, 411
  - metric_precision, 412
  - metric_precision_at_recall, 414
metric_recall, 415
metric_recall_at_precision, 416
metric_root_mean_squared_error, 418
metric_sensitivity_at_specificity, 419
metric_sparse_categorical_accuracy, 420
metric_sparse_categorical_crossentropy, 421
metric_sparse_top_k_categorical_accuracy, 423
metric_specificity_at_sensitivity, 424
metric_squared_hinge, 425
metric_sum, 426
metric_top_k_categorical_accuracy, 427
metric_true_negatives, 428
metric_true_positives, 429

* model functions
  compile.keras.engine.training.Model, 444
  evaluate.keras.engine.training.Model, 445
  fit.keras.engine.training.Model, 446
  get_config, 447
  get_layer, 448
  keras_model, 449
  keras_model_sequential, 450
  pop_layer, 451
  predict.keras.engine.training.Model, 452
  predict_on_batch, 453
  summary.keras.engine.training.Model, 454
  train_on_batch, 455

* model persistence
  get_weights, 456
  model_to_json, 457
  model_to_yaml, 458
  save_model_hdf5, 459
  save_model_tf, 460
  save_model_weights_hdf5, 461
  serialize_model, 462

* noise layers
  layer_alpha_dropout, 463

layer_gaussian_dropout, 271
layer_gaussian_noise, 272

* numerical features preprocessing layers
  layer_discretization, 265
  layer_normalization, 314

* optimizers
  optimizer_adadelta, 434
  optimizer_adagrad, 435
  optimizer_adam, 436
  optimizer_adamax, 437
  optimizer_nadam, 438
  optimizer_rmsprop, 439
  optimizer_sgd, 440

* pooling layers
  layer_average_pooling_1d, 218
  layer_average_pooling_2d, 219
  layer_average_pooling_3d, 220
  layer_global_average_pooling_1d, 221
  layer_global_average_pooling_2d, 222
  layer_global_average_pooling_3d, 223
  layer_global_max_pooling_1d, 224
  layer_global_max_pooling_2d, 225
  layer_global_max_pooling_3d, 226

* preprocessing layer methods
  adapt, 427

* preprocessing layers
  layer_category_encoding, 228
  layer_center_crop, 229
  layer_discretization, 230
  layer_hashing, 231
  layer_integer_lookup, 232
  layer_random_brightness, 233
  layer_random_contrast, 234
  layer_random_crop, 235
  layer_random_flip, 236
  layer_random_height, 237
  layer_random_rotation, 238
  layer_random_translation, 239
  layer_random_width, 240
  layer_random_zoom, 241
  layer_rescaling, 242
*recurrent layers*
- layer_gru, 280
- layer_lstm, 299
- layer_rnn, 334
- layer_simple_rnn, 344

*saved model*
- model_from_saved_model, 430

*text preprocessing layers*
- layer_text_vectorization, 356

*text preprocessing*
- make_sampling_table, 381
- pad_sequences, 441
- skipgrams, 457
- text_hashing_trick, 463
- text_one_hot, 464
- text_to_word_sequence, 466

*text tokenization*
- fit_text_tokenizer, 68
- save_text_tokenizer, 453
- sequences_to_matrix, 454
- text_tokenizer, 465
- texts_to_matrix, 459
- texts_to_sequences, 459
- texts_to_sequences_generator, 460

adapt(), 266, 291, 315, 355, 359
application_densenet, 15
application_densenet121
(application_densenet), 15
application_densenet121_69
(application_densenet), 15
application_densenet201
(application_densenet), 15
application_efficientnet, 17
application_efficientnet_b0
(application_efficientnet), 17
application_efficientnet_b1
(application_efficientnet), 17
application_efficientnet_b2
(application_efficientnet), 17
application_efficientnet_b3
(application_efficientnet), 17
application_efficientnet_b4
(application_efficientnet), 17
application_efficientnet_b5
(application_efficientnet), 17
application_efficientnet_b6
(application_efficientnet), 17
application_efficientnet_b7
(application_efficientnet), 17
application_inception_resnet_v2, 20
application_inception_v3, 22
application_mobilenet, 23
application_mobilenet_v2, 25
application_mobilenet_v3, 27
application_mobilenet_v3_large
(application.mobilenet_v3), 27
application_mobilenet_v3_small
(application.mobilenet_v3), 27
application_nasnet, 30
application_nasnetlarge
(application.nasnet), 30
application_nasnetmobile
(application.nasnet), 30
application_resnet, 31
application_resnet101
(application.resnet), 31
application_resnet101_v2
(application.resnet), 31
application_resnet101_v2_large
(application.resnet), 31
application_resnet50
(application.resnet), 31
application_resnet152
(application.resnet), 31
application_resnet152_v2
(application.resnet), 31
application_resnet152_v2_large
(application.resnet), 31
application_resnet50 (application_resnet), 31
application_resnet50_v2 (application_resnet), 31
application_vgg, 35
application_vgg16 (application_vgg), 35
application_vgg19 (application_vgg), 35
application_xception, 37

backend, 38
backend(), 53
bidirectional, 39, 472
binary_crossentropy, (loss-functions), 377

callback_csv_logger, 40, 42–47, 49
callback_early_stopping, 41, 41–43–47, 49
callback_lambda, 41, 42, 44–47, 49
callback_learning_rate_scheduler, 41–43, 43, 45–47, 49
callback_model_checkpoint, 41–44, 44, 45–47, 49
callback_progbar_logger, 41–45, 45, 46, 47, 49
callback_reduce_lr_on_plateau, 41–45, 46, 47, 49
callback_remote_monitor, 41–46, 47, 49
callback_tensorboard, 41–47, 48, 49
callback_terminate_on_naan, 41–47, 49,

close_model, 50
compile(), 63, 66
compile.keras.engine.training.Model, 50, 64, 67, 77, 79, 102, 103, 440, 445, 446, 459, 474
compile.keras.engine.training.Model(), 380, 381
conda_binary(), 99
constraint_maxnorm (constraints), 52
constraint_maxnorm(), 261
constraint_minmaxnorm (constraints), 52
constraint_nonneg (constraints), 52
constraint_uniitnorm (constraints), 52
constraints, 52
count_params, 54, 77, 79, 80, 448
create_layer, 54
create_layer_wrapper, 55
dataset_boston_housing, 57, 58, 59, 61, 63
dataset_cifar10, 57, 58, 59, 61, 63
dataset_cifar100, 57, 58, 59, 61, 63
dataset_fashion_mnist, 57, 58, 59, 61, 63
dataset_imdb, 57–59, 60, 61, 63
dataset_imdb(), 62
dataset_imdb_word_index (dataset_imdb), 60
dataset_mnist, 57–59, 61, 63
dataset_reuters, 57–59, 61, 62
dataset_reuters_word_index (dataset_reuters), 62
densenet_preprocess_input
(data_processing), 15
evaluate.keras.engine.training.Model, 52, 63, 67, 77, 79, 102, 103, 445, 446, 459, 474
evaluate_generator, 52, 64, 67, 77, 79, 102, 103, 445, 446, 459, 474
evaluate_generator(), 73
export_savedmodel.keras.engine.training.Model, 64

fit.keras.engine.training.Model, 52, 64, 65, 77, 79, 102, 103, 445, 446, 459, 474
fit_generator, 52, 64, 67, 77, 79, 102, 103, 445, 446, 459, 474
fit_generator(), 471
fit_image_data_generator, 67, 69, 72, 74, 87, 88
fit_text_tokenizer, 68, 453, 454, 460, 461, 466
fit_text_tokenizer(), 453
flow_images_from_data, 67, 68, 72, 74, 87, 88
flow_images_from_dataframe, 67, 69, 70, 74, 87, 88
flow_images_from_directory, 67, 69, 72, 72, 87, 88
format.keras.engine.training.Model
(summary.keras.engine.training.Model), 458
freeze_weights, 74
from_config (get_config), 76

generator_next, 76
generate, 52, 54, 64, 67, 76, 79, 80, 102, 103, 445, 446, 448, 459, 474
get_file, 77
get_input_at, 54, 77, 78, 80, 448
get_input_mask_at (get_input_at), 78
get_input_shape_at (get_input_at), 78
get_output_at (get_input_at), 78
get_output_mask_at (get_input_at), 78
get_output_shape_at (get_input_at), 78
get_vocabulary
  (layer_text_vectorization), 356
generate_weights, 54, 77, 79, 80, 431, 432, 448, 450, 452, 456

h5_matrix, 80

image_array_resize (image_to_array), 88
image_array_save (image_to_array), 88
image_data_generator, 85
image_data_generator(), 67, 69, 71, 73, 76
image_dataset_from_directory, 82
image_load, 67, 69, 72, 74, 87, 88
image_to_array, 67, 69, 72, 74, 87, 88
imagenet_decode_predictions, 81
imagenet_preprocess_input, 82
implementation, 89
inception_resnet_v2_preprocess_input
  (application_inception_resnet_v2), 20
inception_v3_preprocess_input
  (application_inception_v3), 22
initializer_constant, 89, 90–98
initializer_glorot_normal, 89, 90, 91–98
initializer_glorot_uniform, 89, 90, 91–98, 261
initializer_he_normal, 89–91, 91, 92–98
initializer_he_uniform, 89–91, 92, 93–98
initializer_identity, 89–92, 92, 93–98
initializer_lecun_normal, 89–93, 93–98, 210
initializer_lecun_uniform, 89–94, 94–98
initializer_ones, 89–94, 94, 95–98
initializer_orthogonal, 89–94, 95, 96–98
initializer_random_normal, 89–95, 95, 96–98
initializer_random_normal(), 96
initializer_random_uniform, 89–96, 96, 97, 98
initializer_truncated_normal, 89–96, 96, 98
initializer_variance_normal, 89–97, 97, 98
initializer_zeros, 89–98, 98
install_keras, 98
is_keras_available, 99
k_abs, 104
k_all, 105
k_any, 105
k_arange, 106
k_argmax, 107
k_argmin, 107
k_backend, 108
k_batch_dot, 108
k_batch_flatten, 109
k_batch_get_value, 110
k_batch_get_value(), 111
k_batch_normalization, 110
k_batch_set_value, 111
k_batch_set_value(), 110
k_bias_add, 112
k_binary_crossentropy, 112
k_batch, 113
k_cast_to_floatx, 114
k_categorical_crossentropy, 114
k_clear_session, 115
k_clip, 115
k_concatenate, 116
k_constant, 117
k_conv1d, 117
k_conv2d, 118
k_conv2d_transpose, 119
k_conv3d, 120
k_conv3d_transpose, 121
k_cos, 122
k_count_params, 122
k_ctc_batch_cost, 123
k_ctc_decode, 124
k_ctc_label_dense_to_sparse, 125
k_cumprod, 125
k_cumsum, 126
k_depthwise_conv2d, 127
<table>
<thead>
<tr>
<th>Function/Method</th>
<th>Page(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>layer_activation.relu</td>
<td>203, 204, 206–208, 210, 210, 212</td>
</tr>
<tr>
<td>layer_activation.softmax</td>
<td>203, 204, 206–208, 210, 210, 212</td>
</tr>
<tr>
<td>layer_activation.thresholdered.relu</td>
<td>203, 204, 206–208, 210, 211, 212</td>
</tr>
<tr>
<td>layer_add</td>
<td>214</td>
</tr>
<tr>
<td>layer_additive_attention</td>
<td>214</td>
</tr>
<tr>
<td>layer_alpha_dropout</td>
<td>210, 215, 272, 273</td>
</tr>
<tr>
<td>layer_average</td>
<td>217, 228, 266, 307, 311, 312, 356</td>
</tr>
<tr>
<td>layer_average_pooling_1d</td>
<td>218, 220, 222, 274, 276–280, 308, 309, 311</td>
</tr>
<tr>
<td>layer_average_pooling_2d</td>
<td>219, 219, 222, 274, 276–280, 308, 309, 311</td>
</tr>
<tr>
<td>layer_average_pooling_3d</td>
<td>219, 220, 221, 274, 276–280, 308, 309, 311</td>
</tr>
<tr>
<td>layer_batch_normalization</td>
<td>222</td>
</tr>
<tr>
<td>layer_concatenate</td>
<td>218, 227, 266, 307, 311, 312, 356</td>
</tr>
<tr>
<td>layer_conv_1d</td>
<td>228, 233, 236, 239, 241, 244, 250, 254–256, 262, 264, 340, 343, 361, 362, 364–366, 368</td>
</tr>
<tr>
<td>layer_conv_1dTranspose</td>
<td>230, 231, 236, 239, 241, 244, 250, 254–256, 262, 264, 340, 343, 361, 362, 364–366, 368</td>
</tr>
<tr>
<td>layer_conv_2dTranspose</td>
<td>230, 233, 236, 239, 241, 244, 250, 254–256, 262, 264, 340, 343, 361, 362, 364–366, 368</td>
</tr>
<tr>
<td>layer_conv_3d</td>
<td>230, 233, 236, 239, 241, 244, 250, 254–256, 262, 264, 340, 343, 361, 362, 364–366, 368</td>
</tr>
</tbody>
</table>
INDEX

layer_multiply, 218, 228, 266, 307, 311, 312, 356
layer_random_zoom, 226, 227, 266, 288, 291, 315, 318–324, 326, 327, 328, 331, 334, 355, 359
layer_rn, 284, 303, 334, 347
layer_separable_conv_2d, 230, 233, 236, 239, 241, 244, 250, 254–256, 262, 264, 340, 341, 361, 362, 364–366, 368
layer_simple_rnn, 284, 303, 337, 344
layer_simple_rnn_cell, 286, 305, 347, 353
layer.spatial_dropout_1d, 268, 249, 351, 352
layer.spatial_dropout_2d, 268, 350, 350, 352
layer.spatial_dropout_3d, 268, 350, 351, 351
layer_stacked_rnn_cells, 286, 305, 348, 352
layer_subtract, 218, 228, 266, 307, 311, 312, 355
layer_unit_normalization, 359
layer_upsampling_1d, 230, 233, 236, 239, 241, 244, 250, 254–256, 262, 264, 340, 343, 360, 362, 364–366, 368
layer_upsampling_2d, 230, 233, 236, 239, 241, 244, 250, 254–256, 262, 264, 340, 343, 361, 364–366, 368
layer_zero_padding_1d, 230, 233, 236, 239, 241, 244, 250, 254–256, 262, 264, 340, 343, 361, 362, 364, 366, 368
layer_zero_padding_2d, 230, 233, 236, 239, 241, 244, 250, 254–256, 262, 264, 340, 343, 361, 362, 364, 365, 368
learning_rate_schedule_cosine_decay,
learning_rate_schedule_inverse_time_decay, 369
learning_rate_schedule_exponential_decay, 370
learning_rate_schedule_inverse_time_decay, 372
learning_rate_schedule_piecewise_constant_decay, 374
learning_rate_schedule_polynomial_decay, 375
load_model_hdf5 (save_model_hdf5), 449
load_model_hdf5(), 56, 453
load_model_tf (save_model_tf), 450
load_model_weights_hdf5
(save_model_weights_hdf5), 451
load_model_weights_tf
(save_model_weights_tf), 452
load_text_tokenizer
(save_text_tokenizer), 453
loss_functions, 377
loss_binary_crossentropy
(loss-functions), 377
loss_binary_crossentropy(), 381
loss_categorical_crossentropy
(loss-functions), 377
loss_categorical_crossentropy(), 473
loss_categorical_hinge
(loss-functions), 377
loss_cosine_similarity
(loss-functions), 377
loss_hinge (loss-functions), 377
loss_huber (loss-functions), 377
loss_kl_divergence
(loss-functions), 377
loss_kullback_leibler_divergence
(loss-functions), 377
loss_logcosh (loss-functions), 377
loss_mean_absolute_error
(loss-functions), 377
loss_mean_absolute_percentage_error
(loss-functions), 377
loss_mean_squared_error
(loss-functions), 377
loss_mean_squared_logarithmic_error
(loss-functions), 377
loss_poisson (loss-functions), 377
loss_sparse_categorical_crossentropy
(loss-functions), 377
loss_squared_hinge (loss-functions), 377
make_sampling_table, 381, 442, 458, 463, 464, 466
makeActiveBinding(), 479
mark_active, 382
Metric, 383
metric_kullback_leibler_divergence, 56, 386, 388, 389, 391, 392, 394,
metric_mean_squared_error

metric_mean

INDEX

491


metric_mean_wrapper()


metric_true_negatives, 57, 386, 388, 390–392, 394, 396–404, 406–413,
serialize_model(), 449
set_vocabulary
  (layer_text_vectorization), 356
set_weights (get_weights), 80
skipgrams, 382, 442, 457, 463, 464, 466
skipgrams(), 381
summary.keras.engine.training.Model,
  52, 64, 67, 77, 79, 102, 103, 445, 446, 458, 474
tensorflow::install_tensorflow(), 98, 99
test_on_batch (train_on_batch), 473
text_dataset_from_directory, 461
text_hashing_trick, 382, 442, 458, 463, 464, 466
text_one_hot, 382, 442, 458, 463, 464, 466
text_to_word_sequence, 382, 442, 458, 463, 464, 466
text_tokenizer, 68, 453, 454, 460, 461, 465
text_tokenizer(), 68
texts_to_matrix, 68, 453, 454, 459, 460, 461, 466
texts_to_matrix(), 68
texts_to_sequences, 68, 453, 454, 460, 460, 461, 466
texts_to_sequences(), 68
texts_to_sequences_generator, 68, 453, 454, 460, 466
time_distributed, 40, 471
timeseries_dataset_from_array, 467
timeseries_generator, 470
to_categorical, 472
to_categorical(), 380
train_on_batch, 52, 64, 67, 77, 79, 102, 103, 445, 446, 459, 473
unfreeze_weights (freeze_weights), 74
unserialize_model (serialize_model), 456
use_backend (use_implementation), 474
use_implementation, 474
with_custom_object_scope, 475
with_custom_object_scope(), 56, 449
xception_preprocess_input
  (application_xception), 37
zip_lists, 476