Package 'tfestimators'

October 14, 2022

Type Package

Title Interface to 'TensorFlow' Estimators

Version 1.9.2

Description Interface to 'TensorFlow' Estimators <https://www.tensorflow.org/guide/estimator>, a high-level API that provides implementations of many different model types including linear models and deep neural networks.

License Apache License 2.0

URL https://github.com/rstudio/tfestimators

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

SystemRequirements TensorFlow (https://www.tensorflow.org/)

Encoding UTF-8

Depends R (>= 3.1)

Imports forge, magrittr, progress, reticulate (>= 1.10), rlang (>= 0.3), tensorflow (>= 1.9), tfruns (>= 1.1), tidyselect, utils, purrr, tibble, tidyr

RoxygenNote 7.1.1

Suggests ggplot2, modelr (>= 0.1.1), testthat, rmarkdown, knitr

VignetteBuilder knitr

NeedsCompilation no

Author JJ Allaire [aut],
Yuan Tang [aut] (<https://orcid.org/0000-0001-5243-233X>),
Kevin Ushey [aut],
Kevin Kuo [aut] (<https://orcid.org/0000-0001-7803-7901>),
Tomasz Kalinowski [cre],
Daniel Falbel [ctb, cph],
RStudio [cph, fnd],
Google Inc. [cph]

Maintainer Tomasz Kalinowski <tomasz.kalinowski@rstudio.com>

Repository CRAN

Date/Publication 2021-08-09 22:30:02 UTC

R topics documented:

`boosted_trees_estimators` *Boosted Trees Estimator*

Description

Construct a boosted trees estimator.

Usage

```
boosted_trees_regressor(
  feature_columns,
  n_batches_per_layer,
  model_dir = NULL,
  label_dimension = 1L,
  weight_column = NULL,
  n_trees = 100L,
  max_depth = 6L,
  learning_rate = 0.1,
  l1_regularization = 0,
  l2_regularization = 0,
  tree_complexity = 0,
  min_node_weight = 0,
  config = NULL
)

boosted_trees_classifier(
  feature_columns,
  n_batches_per_layer,
  model_dir = NULL,
  n_classes = 2L,
  weight_column = NULL,
  label_vocabulary = NULL,
  n_trees = 100L,
```

```
  max_depth = 6L,
  learning_rate = 0.1,
  l1_regularization = 0,
  l2_regularization = 0,
  tree_complexity = 0,
  min_node_weight = 0,
  config = NULL
)
```

Arguments

feature_columns	An R list containing all of the feature columns used by the model (typically, generated by feature_columns()).
n_batches_per_layer	The number of batches to collect statistics per layer.
model_dir	Directory to save the model parameters, graph, and so on. This can also be used to load checkpoints from the directory into a estimator to continue training a previously saved model.
label_dimension	Number of regression targets per example. This is the size of the last dimension of the labels and logits Tensor objects (typically, these have shape [batch_size, label_dimension]).
weight_column	A string, or a numeric column created by column_numeric() defining feature column representing weights. It is used to down weight or boost examples during training. It will be multiplied by the loss of the example. If it is a string, it is used as a key to fetch weight tensor from the features argument. If it is a numeric column, then the raw tensor is fetched by key weight_column$key, then weight_column$normalizer_fn is applied on it to get weight tensor.
n_trees	Number trees to be created.
max_depth	Maximum depth of the tree to grow.
learning_rate	Shrinkage parameter to be used when a tree added to the model.
l1_regularization	Regularization multiplier applied to the absolute weights of the tree leafs.
l2_regularization	Regularization multiplier applied to the square weights of the tree leafs.
tree_complexity	Regularization factor to penalize trees with more leaves.
min_node_weight	Minimum hessian a node must have for a split to be considered. The value will be compared with sum(leaf_hessian)/(batch_size * n_batches_per_layer).
config	A run configuration created by run_config(), used to configure the runtime settings.
n_classes	The number of label classes.
label_vocabulary	A list of strings represents possible label values. If given, labels must be string type and have any value in label_vocabulary. If it is not given, that means

labels are already encoded as integer or float within `[0, 1]` for `n_classes == 2` and encoded as integer values in `{0, 1,..., n_classes -1}` for `n_classes > 2`. Also there will be errors if vocabulary is not provided and labels are string.

See Also

Other canned estimators: `dnn_estimators`, `dnn_linear_combined_estimators`, `linear_estimators`

`classifier_parse_example_spec` *Generates Parsing Spec for TensorFlow Example to be Used with Classifiers*

Description

If users keep data in TensorFlow Example format, they need to call `tf$parse_example` with a proper feature spec. There are two main things that this utility helps:

- Users need to combine parsing spec of features with labels and weights (if any) since they are all parsed from same `tf$Example` instance. This utility combines these specs.
- It is difficult to map expected label by a classifier such as `dnn_classifier` to corresponding `tf$parse_example` spec. This utility encodes it by getting related information from users (key, dtype).

Usage

```
classifier_parse_example_spec(
  feature_columns,
  label_key,
  label_dtype = tf$int64,
  label_default = NULL,
  weight_column = NULL
)
```

Arguments

`feature_columns` An iterable containing all feature columns. All items should be instances of classes derived from `_FeatureColumn`.

`label_key` A string identifying the label. It means `tf$Example` stores labels with this key.

`label_dtype` A `tf$dtype` identifies the type of labels. By default it is `tf$int64`. If user defines a `label_vocabulary`, this should be set as `tf$string`. `tf$float32` labels are only supported for binary classification.

`label_default` used as label if label_key does not exist in given `tf$Example`. An example usage: let's say `label_key` is 'clicked' and `tf$Example` contains clicked data only for positive examples in following format `key:clicked, value:1`. This means that if there is no data with key 'clicked' it should count as negative example by setting `label_deafault=0`. Type of this value should be compatible with `label_dtype`.

`weight_column` A string or a numeric column created by `column_numeric()` defining feature column representing weights. It is used to down weight or boost examples during training. It will be multiplied by the loss of the example. If it is a string, it is used as a key to fetch weight tensor from the `features`. If it is a numeric column, raw tensor is fetched by key `weight_column$key`, then `weight_column$normalizer_fn` is applied on it to get weight tensor.

Value

A dict mapping each feature key to a `FixedLenFeature` or `VarLenFeature` value.

Raises

- ValueError: If label is used in `feature_columns`.
- ValueError: If weight_column is used in `feature_columns`.
- ValueError: If any of the given `feature_columns` is not a feature column instance.
- ValueError: If `weight_column` is not a numeric column instance.
- ValueError: if label_key is `NULL`.

See Also

Other parsing utilities: `regressor_parse_example_spec()`

`column-scope` *Establish a Feature Columns Selection Scope*

Description

This helper function provides a set of names to be used by `tidyselect` helpers in e.g. `feature_columns()`.

Usage

```
set_columns(columns)

with_columns(columns, expr)

scoped_columns(columns)
```

Arguments

columns	Either a named R object (whose names will be used to provide a selection context), or a character vector of such names.
expr	An R expression, to be evaluated with the selection context active.

---

column_base *Base Documentation for Feature Column Constructors*

Description

Base Documentation for Feature Column Constructors

Arguments

...	Expression(s) identifying input feature(s). Used as the column name and the dictionary key for feature parsing configs, feature tensors, and feature columns.

---

column_bucketized *Construct a Bucketized Column*

Description

Construct a bucketized column, representing discretized dense input. Buckets include the left boundary, and exclude the right boundary.

Usage

```
column_bucketized(source_column, boundaries)
```

Arguments

source_column	A one-dimensional dense column, as generated by column_numeric().
boundaries	A sorted list or list of floats specifying the boundaries.

Value

A bucketized column.

Raises

- ValueError: If source_column is not a numeric column, or if it is not one-dimensional.
- ValueError: If boundaries is not a sorted list or list.

See Also

Other feature column constructors: column_categorical_weighted(), column_categorical_with_hash_bucket(), column_categorical_with_identity(), column_categorical_with_vocabulary_file(), column_categorical_wit
column_crossed(), column_embedding(), column_numeric(), input_layer()

column_categorical_weighted *Construct a Weighted Categorical Column*

Description

Use this when each of your sparse inputs has both an ID and a value. For example, if you're representing text documents as a collection of word frequencies, you can provide 2 parallel sparse input features ('terms' and 'frequencies' below).

Usage

```
column_categorical_weighted(
  categorical_column,
  weight_feature_key,
  dtype = tf$float32
)
```

Arguments

categorical_column	A categorical column created by column_categorical_*() functions.
weight_feature_key	String key for weight values.
dtype	Type of weights, such as tf$float32. Only float and integer weights are supported.

Value

A categorical column composed of two sparse features: one represents id, the other represents weight (value) of the id feature in that example.

Raises

- ValueError: if dtype is not convertible to float.

See Also

Other feature column constructors: column_bucketized(), column_categorical_with_hash_bucket(), column_categorical_with_identity(), column_categorical_with_vocabulary_file(), column_categorical_wit column_crossed(), column_embedding(), column_numeric(), input_layer()

column_categorical_with_hash_bucket
Represents Sparse Feature where IDs are set by Hashing

Description

Use this when your sparse features are in string or integer format, and you want to distribute your inputs into a finite number of buckets by hashing. output_id = Hash(input_feature_string) % bucket_size For input dictionary `features`, `features$key$` is either tensor or sparse tensor object. If it's tensor object, missing values can be represented by `-1` for int and `''` for string. Note that these values are independent of the `default_value` argument.

Usage

```
column_categorical_with_hash_bucket(..., hash_bucket_size, dtype = tf$string)
```

Arguments

`...`	Expression(s) identifying input feature(s). Used as the column name and the dictionary key for feature parsing configs, feature tensors, and feature columns.
`hash_bucket_size`	An int > 1. The number of buckets.
`dtype`	The type of features. Only string and integer types are supported.

Value

A `_HashedCategoricalColumn`.

Raises

- ValueError: `hash_bucket_size` is not greater than 1.
- ValueError: `dtype` is neither string nor integer.

See Also

Other feature column constructors: `column_bucketized()`, `column_categorical_weighted()`, `column_categorical_with_identity()`, `column_categorical_with_vocabulary_file()`, `column_categorical_wit`
`column_crossed()`, `column_embedding()`, `column_numeric()`, `input_layer()`

column_categorical_with_identity *Construct a Categorical Column that Returns Identity Values*

Description

Use this when your inputs are integers in the range `[0, num_buckets)`, and you want to use the input value itself as the categorical ID. Values outside this range will result in `default_value` if specified, otherwise it will fail.

Usage

```
column_categorical_with_identity(..., num_buckets, default_value = NULL)
```

Arguments

`...`	Expression(s) identifying input feature(s). Used as the column name and the dictionary key for feature parsing configs, feature tensors, and feature columns.
`num_buckets`	Number of unique values.
`default_value`	If `NULL`, this column's graph operations will fail for out-of-range inputs. Otherwise, this value must be in the range `[0, num_buckets)`, and will replace inputs in that range.

Details

Typically, this is used for contiguous ranges of integer indexes, but it doesn't have to be. This might be inefficient, however, if many of IDs are unused. Consider `column_categorical_with_hash_bucket()` in that case.

For input dictionary `features`, `features$key` is either tensor or sparse tensor object. If it's tensor object, missing values can be represented by `-1` for int and `''` for string. Note that these values are independent of the `default_value` argument.

Value

A categorical column that returns identity values.

Raises

- ValueError: if `num_buckets` is less than one.
- ValueError: if `default_value` is not in range `[0, num_buckets)`.

See Also

Other feature column constructors: `column_bucketized()`, `column_categorical_weighted()`, `column_categorical_with_hash_bucket()`, `column_categorical_with_vocabulary_file()`, `column_categorical_with_vocabulary_list()`, `column_crossed()`, `column_embedding()`, `column_numeric()`, `input_layer()`

column_categorical_with_vocabulary_file
Construct a Categorical Column with a Vocabulary File

Description

Use this when your inputs are in string or integer format, and you have a vocabulary file that maps each value to an integer ID. By default, out-of-vocabulary values are ignored. Use either (but not both) of `num_oov_buckets` and `default_value` to specify how to include out-of-vocabulary values. For input dictionary `features`, `features[key]` is either tensor or sparse tensor object. If it's tensor object, missing values can be represented by `-1` for int and `''` for string. Note that these values are independent of the `default_value` argument.

Usage

```
column_categorical_with_vocabulary_file(
  ...,
  vocabulary_file,
  vocabulary_size,
  num_oov_buckets = 0L,
  default_value = NULL,
  dtype = tf$string
)
```

Arguments

`...`	Expression(s) identifying input feature(s). Used as the column name and the dictionary key for feature parsing configs, feature tensors, and feature columns.
`vocabulary_file`	The vocabulary file name.
`vocabulary_size`	Number of the elements in the vocabulary. This must be no greater than length of `vocabulary_file`, if less than length, later values are ignored.
`num_oov_buckets`	Non-negative integer, the number of out-of-vocabulary buckets. All out-of-vocabulary inputs will be assigned IDs in the range `[vocabulary_size, vocabulary_size+num_oov_bu` based on a hash of the input value. A positive `num_oov_buckets` can not be specified with `default_value`.
`default_value`	The integer ID value to return for out-of-vocabulary feature values, defaults to `-1`. This can not be specified with a positive `num_oov_buckets`.
`dtype`	The type of features. Only string and integer types are supported.

Value

A categorical column with a vocabulary file.

Raises

- ValueError: `vocabulary_file` is missing.
- ValueError: `vocabulary_size` is missing or < 1.
- ValueError: `num_oov_buckets` is not a non-negative integer.
- ValueError: `dtype` is neither string nor integer.

See Also

Other feature column constructors: `column_bucketized()`, `column_categorical_weighted()`, `column_categorical_with_hash_bucket()`, `column_categorical_with_identity()`, `column_categorical_with_vo`
`column_crossed()`, `column_embedding()`, `column_numeric()`, `input_layer()`

`column_categorical_with_vocabulary_list`
Construct a Categorical Column with In-Memory Vocabulary

Description

Use this when your inputs are in string or integer format, and you have an in-memory vocabulary mapping each value to an integer ID. By default, out-of-vocabulary values are ignored. Use `default_value` to specify how to include out-of-vocabulary values. For the input dictionary `features`, `features$key` is either tensor or sparse tensor object. If it's tensor object, missing values can be represented by `-1` for int and `''` for string.

Usage

```
column_categorical_with_vocabulary_list(
  ...,
  vocabulary_list,
  dtype = NULL,
  default_value = -1L,
  num_oov_buckets = 0L
)
```

Arguments

`...`	Expression(s) identifying input feature(s). Used as the column name and the dictionary key for feature parsing configs, feature tensors, and feature columns.
`vocabulary_list`	An ordered iterable defining the vocabulary. Each feature is mapped to the index of its value (if present) in `vocabulary_list`. Must be castable to `dtype`.
`dtype`	The type of features. Only string and integer types are supported. If `NULL`, it will be inferred from `vocabulary_list`.
`default_value`	The value to use for values not in `vocabulary_list`.

num_oov_buckets
: Non-negative integer, the number of out-of-vocabulary buckets. All out-of-vocabulary inputs will be assigned IDs in the range [vocabulary_size, vocabulary_size+num_oov_bu based on a hash of the input value. A positive num_oov_buckets can not be specified with default_value.

Details

Note that these values are independent of the default_value argument.

Value

A categorical column with in-memory vocabulary.

Raises

- ValueError: if vocabulary_list is empty, or contains duplicate keys.
- ValueError: if dtype is not integer or string.

See Also

Other feature column constructors: column_bucketized(), column_categorical_weighted(), column_categorical_with_hash_bucket(), column_categorical_with_identity(), column_categorical_with_vo column_crossed(), column_embedding(), column_numeric(), input_layer()

---

column_crossed *Construct a Crossed Column*

Description

Returns a column for performing crosses of categorical features. Crossed features will be hashed according to hash_bucket_size.

Usage

```
column_crossed(keys, hash_bucket_size, hash_key = NULL)
```

Arguments

keys
: An iterable identifying the features to be crossed. Each element can be either:
  - string: Will use the corresponding feature which must be of string type.
  - categorical column: Will use the transformed tensor produced by this column. Does not support hashed categorical columns.

hash_bucket_size
: The number of buckets (> 1).

hash_key
: Optional: specify the hash_key that will be used by the FingerprintCat64 function to combine the crosses fingerprints on SparseCrossOp.

Value

A crossed column.

Raises

- ValueError: If `len(keys) < 2`.
- ValueError: If any of the keys is neither a string nor categorical column.
- ValueError: If any of the keys is `_HashedCategoricalColumn`.
- ValueError: If `hash_bucket_size < 1`.

See Also

Other feature column constructors: `column_bucketized()`, `column_categorical_weighted()`, `column_categorical_with_hash_bucket()`, `column_categorical_with_identity()`, `column_categorical_with_vo` `column_categorical_with_vocabulary_list()`, `column_embedding()`, `column_numeric()`, `input_layer()`

`column_embedding` *Construct a Dense Column*

Description

Use this when your inputs are sparse, but you want to convert them to a dense representation (e.g., to feed to a DNN). Inputs must be a categorical column created by any of the `column_categorical_*()` functions.

Usage

```
column_embedding(
  categorical_column,
  dimension,
  combiner = "mean",
  initializer = NULL,
  ckpt_to_load_from = NULL,
  tensor_name_in_ckpt = NULL,
  max_norm = NULL,
  trainable = TRUE
)
```

Arguments

`categorical_column`	A categorical column created by a `column_categorical_*()` function. This column produces the sparse IDs that are inputs to the embedding lookup.
`dimension`	A positive integer, specifying dimension of the embedding.

combiner	A string specifying how to reduce if there are multiple entries in a single row. Currently "mean", "sqrtn" and "sum" are supported, with "mean" the default. "sqrtn"' often achieves good accuracy, in particular with bag-of-words columns. Each of this can be thought as example level normalizations on the column.
initializer	A variable initializer function to be used in embedding variable initialization. If not specified, defaults to tf$truncated_normal_initializer with mean 0.0 and standard deviation 1 / sqrt(dimension).
ckpt_to_load_from	String representing checkpoint name/pattern from which to restore column weights. Required if tensor_name_in_ckpt is not NULL.
tensor_name_in_ckpt	Name of the Tensor in ckpt_to_load_from from which to restore the column weights. Required if ckpt_to_load_from is not NULL.
max_norm	If not NULL, embedding values are l2-normalized to this value.
trainable	Whether or not the embedding is trainable. Default is TRUE.

Value

A dense column that converts from sparse input.

Raises

- ValueError: if dimension not > 0.
- ValueError: if exactly one of ckpt_to_load_from and tensor_name_in_ckpt is specified.
- ValueError: if initializer is specified and is not callable.

See Also

Other feature column constructors: column_bucketized(), column_categorical_weighted(), column_categorical_with_hash_bucket(), column_categorical_with_identity(), column_categorical_with_vo column_categorical_with_vocabulary_list(), column_crossed(), column_numeric(), input_layer()

---

column_indicator *Represents Multi-Hot Representation of Given Categorical Column*

Description

Used to wrap any column_categorical()* (e.g., to feed to DNN). Use column_embedding() if the inputs are sparse.

Usage

```
column_indicator(categorical_column)
```

Arguments

categorical_column
: A categorical column which is created by the `column_categorical_with_*()` or `column_crossed()` functions.

Value

An indicator column.

---

column_numeric *Construct a Real-Valued Column*

Description

Construct a Real-Valued Column

Usage

```
column_numeric(
  ...,
  shape = c(1L),
  default_value = NULL,
  dtype = tf$float32,
  normalizer_fn = NULL
)
```

Arguments

`...`
: Expression(s) identifying input feature(s). Used as the column name and the dictionary key for feature parsing configs, feature tensors, and feature columns.

`shape`
: An integer vector that specifies the shape of the tensor. An integer can be given which means a single dimension tensor with given width. The tensor representing the column will have the shape of `batch_size + shape`.

`default_value`
: A single value compatible with `dtype` or an iterable of values compatible with `dtype` which the column takes on during parsing if data is missing. A default value of `NULL` will cause `tf$parse_example` to fail if an example does not contain this column. If a single value is provided, the same value will be applied as the default value for every item. If an iterable of values is provided, the shape of the `default_value` should be equal to the given `shape`.

`dtype`
: The types for values contained in the column. The default value is `tf$float32`. Must be a non-quantized, real integer or floating point type.

`normalizer_fn`
: If not `NULL`, a function that can be used to normalize the value of the tensor after `default_value` is applied for parsing. Normalizer function takes the input `Tensor` as its argument, and returns the output tensor. (e.g. `function(x) {(x - 3.0) / 4.2)}`. Please note that even though the most common use case of this function is normalization, it can be used for any kind of Tensorflow transformations.

Value

A numeric column.

Raises

- TypeError: if any dimension in shape is not an int
- ValueError: if any dimension in shape is not a positive integer
- TypeError: if `default_value` is an iterable but not compatible with `shape`
- TypeError: if `default_value` is not compatible with `dtype`
- ValueError: if `dtype` is not convertible to `tf$float32`

See Also

Other feature column constructors: `column_bucketized()`, `column_categorical_weighted()`, `column_categorical_with_hash_bucket()`, `column_categorical_with_identity()`, `column_categorical_with_vo`
`column_categorical_with_vocabulary_list()`, `column_crossed()`, `column_embedding()`, `input_layer()`

---

dnn_estimators *Deep Neural Networks*

Description

Create a deep neural network (DNN) estimator.

Usage

```
dnn_regressor(
  hidden_units,
  feature_columns,
  model_dir = NULL,
  label_dimension = 1L,
  weight_column = NULL,
  optimizer = "Adagrad",
  activation_fn = "relu",
  dropout = NULL,
  input_layer_partitioner = NULL,
  config = NULL
)

dnn_classifier(
  hidden_units,
  feature_columns,
  model_dir = NULL,
  n_classes = 2L,
  weight_column = NULL,
  label_vocabulary = NULL,
```

```
  optimizer = "Adagrad",
  activation_fn = "relu",
  dropout = NULL,
  input_layer_partitioner = NULL,
  config = NULL
)
```

Arguments

hidden_units	An integer vector, indicating the number of hidden units in each layer. All layers are fully connected. For example, `c(64, 32)` means the first layer has 64 nodes, and the second layer has 32 nodes.
feature_columns	An R list containing all of the feature columns used by the model (typically, generated by `feature_columns()`).
model_dir	Directory to save the model parameters, graph, and so on. This can also be used to load checkpoints from the directory into a estimator to continue training a previously saved model.
label_dimension	Number of regression targets per example. This is the size of the last dimension of the labels and logits `Tensor` objects (typically, these have shape `[batch_size, label_dimension]`).
weight_column	A string, or a numeric column created by `column_numeric()` defining feature column representing weights. It is used to down weight or boost examples during training. It will be multiplied by the loss of the example. If it is a string, it is used as a key to fetch weight tensor from the `features` argument. If it is a numeric column, then the raw tensor is fetched by key `weight_column$key`, then `weight_column$normalizer_fn` is applied on it to get weight tensor.
optimizer	Either the name of the optimizer to be used when training the model, or a TensorFlow optimizer instance. Defaults to the Adagrad optimizer.
activation_fn	The activation function to apply to each layer. This can either be an actual activation function (e.g. `tf$nn$relu`), or the name of an activation function (e.g. `"relu"`). Defaults to the `"relu"` activation function. See https://www.tensorflow.org/versions/r1.15/api_docs/python/tf/nn for documentation related to the set of activation functions available in TensorFlow.
dropout	When not `NULL`, the probability we will drop out a given coordinate.
input_layer_partitioner	An optional partitioner for the input layer. Defaults to `min_max_variable_partitioner` with `min_slice_size` 64 « 20.
config	A run configuration created by `run_config()`, used to configure the runtime settings.
n_classes	The number of label classes.
label_vocabulary	A list of strings represents possible label values. If given, labels must be string type and have any value in `label_vocabulary`. If it is not given, that means labels are already encoded as integer or float within `[0, 1]` for `n_classes == 2` and encoded as integer values in `{0, 1,..., n_classes -1}` for `n_classes > 2`. Also there will be errors if vocabulary is not provided and labels are string.

See Also

Other canned estimators: boosted_trees_estimators, dnn_linear_combined_estimators, linear_estimators

---

dnn_linear_combined_estimators

Linear Combined Deep Neural Networks

Description

Also known as wide-n-deep estimators, these are estimators for TensorFlow Linear and DNN joined models for regression.

Usage

```
dnn_linear_combined_regressor(
  model_dir = NULL,
  linear_feature_columns = NULL,
  linear_optimizer = "Ftrl",
  dnn_feature_columns = NULL,
  dnn_optimizer = "Adagrad",
  dnn_hidden_units = NULL,
  dnn_activation_fn = "relu",
  dnn_dropout = NULL,
  label_dimension = 1L,
  weight_column = NULL,
  input_layer_partitioner = NULL,
  config = NULL
)

dnn_linear_combined_classifier(
  model_dir = NULL,
  linear_feature_columns = NULL,
  linear_optimizer = "Ftrl",
  dnn_feature_columns = NULL,
  dnn_optimizer = "Adagrad",
  dnn_hidden_units = NULL,
  dnn_activation_fn = "relu",
  dnn_dropout = NULL,
  n_classes = 2L,
  weight_column = NULL,
  label_vocabulary = NULL,
  input_layer_partitioner = NULL,
  config = NULL
)
```

Arguments

model_dir	Directory to save the model parameters, graph, and so on. This can also be used to load checkpoints from the directory into a estimator to continue training a previously saved model.
linear_feature_columns	The feature columns used by linear (wide) part of the model.
linear_optimizer	Either the name of the optimizer to be used when training the model, or a TensorFlow optimizer instance. Defaults to the FTRL optimizer.
dnn_feature_columns	The feature columns used by the neural network (deep) part in the model.
dnn_optimizer	Either the name of the optimizer to be used when training the model, or a TensorFlow optimizer instance. Defaults to the Adagrad optimizer.
dnn_hidden_units	An integer vector, indicating the number of hidden units in each layer. All layers are fully connected. For example, `c(64, 32)` means the first layer has 64 nodes, and the second layer has 32 nodes.
dnn_activation_fn	The activation function to apply to each layer. This can either be an actual activation function (e.g. `tf$nn$relu`), or the name of an activation function (e.g. `"relu"`). Defaults to the `"relu"` activation function. See https://www.tensorflow.org/versions/r1.15/api_docs/python/tf/nn for documentation related to the set of activation functions available in TensorFlow.
dnn_dropout	When not `NULL`, the probability we will drop out a given coordinate.
label_dimension	Number of regression targets per example. This is the size of the last dimension of the labels and logits `Tensor` objects (typically, these have shape `[batch_size, label_dimension]`).
weight_column	A string, or a numeric column created by `column_numeric()` defining feature column representing weights. It is used to down weight or boost examples during training. It will be multiplied by the loss of the example. If it is a string, it is used as a key to fetch weight tensor from the `features` argument. If it is a numeric column, then the raw tensor is fetched by key `weight_column$key`, then `weight_column$normalizer_fn` is applied on it to get weight tensor.
input_layer_partitioner	An optional partitioner for the input layer. Defaults to `min_max_variable_partitioner` with `min_slice_size` 64 « 20.
config	A run configuration created by `run_config()`, used to configure the runtime settings.
n_classes	The number of label classes.
label_vocabulary	A list of strings represents possible label values. If given, labels must be string type and have any value in `label_vocabulary`. If it is not given, that means labels are already encoded as integer or float within `[0, 1]` for `n_classes` == 2 and encoded as integer values in `{0, 1,..., n_classes -1}` for `n_classes` > 2. Also there will be errors if vocabulary is not provided and labels are string.

See Also

Other canned estimators: boosted_trees_estimators, dnn_estimators, linear_estimators

---

`estimator` *Construct a Custom Estimator*

Description

Construct a custom estimator, to be used to train and evaluate TensorFlow models.

Usage

```
estimator(
  model_fn,
  model_dir = NULL,
  config = NULL,
  params = NULL,
  class = NULL
)
```

Arguments

`model_fn`	The model function. See **Model Function** for details on the structure of a model function.
`model_dir`	Directory to save model parameters, graph and etc. This can also be used to load checkpoints from the directory into a estimator to continue training a previously saved model. If `NULL`, the `model_dir` in `config` will be used if set. If both are set, they must be same. If both are `NULL`, a temporary directory will be used.
`config`	Configuration object.
`params`	List of hyper parameters that will be passed into `model_fn`. Keys are names of parameters, values are basic python types.
`class`	An optional set of R classes to add to the generated object.

Details

The `Estimator` object wraps a model which is specified by a `model_fn`, which, given inputs and a number of other parameters, returns the operations necessary to perform training, evaluation, and prediction.

All outputs (checkpoints, event files, etc.) are written to `model_dir`, or a subdirectory thereof. If `model_dir` is not set, a temporary directory is used.

The `config` argument can be used to passed run configuration object containing information about the execution environment. It is passed on to the `model_fn`, if the `model_fn` has a parameter named "config" (and input functions in the same manner). If the `config` parameter is not passed, it is instantiated by `estimator()`. Not passing config means that defaults useful for local execution are used. `estimator()` makes config available to the model (for instance, to allow specialization based

on the number of workers available), and also uses some of its fields to control internals, especially regarding checkpointing.

The `params` argument contains hyperparameters. It is passed to the `model_fn`, if the `model_fn` has a parameter named "params", and to the input functions in the same manner. `estimator()` only passes `params` along, it does not inspect it. The structure of `params` is therefore entirely up to the developer.

None of estimator's methods can be overridden in subclasses (its constructor enforces this). Subclasses should use `model_fn` to configure the base class, and may add methods implementing specialized functionality.

Model Functions

The `model_fn` should be an R function of the form:

```
function(features, labels, mode, params) {
    # 1. Configure the model via TensorFlow operations.
    # 2. Define the loss function for training and evaluation.
    # 3. Define the training optimizer.
    # 4. Define how predictions should be produced.
    # 5. Return the result as an `estimator_spec()` object.
    estimator_spec(mode, predictions, loss, train_op, eval_metric_ops)
}
```

The model function's inputs are defined as follows:

`features`	The feature tensor(s).
`labels`	The label tensor(s).
`mode`	The current training mode ("train", "eval", "infer"). These can be accessed through the `mode_keys()` object.
`params`	An optional list of hyperparameters, as received through the `estimator()` constructor.

See `estimator_spec()` for more details as to how the estimator specification should be constructed, and https://www.tensorflow.org/versions/r1.15/api_docs/python/tf/estimator/Estimator for more information as to how the model function should be constructed.

See Also

Other custom estimator methods: `estimator_spec()`, `evaluate.tf_estimator()`, `export_savedmodel.tf_estimator()`, `predict.tf_estimator()`, `train.tf_estimator()`

`estimators` *Base Documentation for Canned Estimators*

Description

Base Documentation for Canned Estimators

Arguments

object	A TensorFlow estimator.
feature_columns	An R list containing all of the feature columns used by the model (typically, generated by feature_columns()).
model_dir	Directory to save the model parameters, graph, and so on. This can also be used to load checkpoints from the directory into a estimator to continue training a previously saved model.
label_dimension	Number of regression targets per example. This is the size of the last dimension of the labels and logits Tensor objects (typically, these have shape [batch_size, label_dimension]).
label_vocabulary	A list of strings represents possible label values. If given, labels must be string type and have any value in label_vocabulary. If it is not given, that means labels are already encoded as integer or float within [0, 1] for n_classes == 2 and encoded as integer values in {0, 1,..., n_classes -1} for n_classes > 2. Also there will be errors if vocabulary is not provided and labels are string.
weight_column	A string, or a numeric column created by column_numeric() defining feature column representing weights. It is used to down weight or boost examples during training. It will be multiplied by the loss of the example. If it is a string, it is used as a key to fetch weight tensor from the features argument. If it is a numeric column, then the raw tensor is fetched by key weight_column$key, then weight_column$normalizer_fn is applied on it to get weight tensor.
n_classes	The number of label classes.
config	A run configuration created by run_config(), used to configure the runtime settings.
input_layer_partitioner	An optional partitioner for the input layer. Defaults to min_max_variable_partitioner with min_slice_size 64 « 20.
partitioner	An optional partitioner for the input layer.

---

estimator_spec *Define an Estimator Specification*

Description

Define the estimator specification, used as part of the model_fn defined with custom estimators created by estimator(). See estimator() for more details.

Usage

```
estimator_spec(
  mode,
  predictions = NULL,
```

```
  loss = NULL,
  train_op = NULL,
  eval_metric_ops = NULL,
  training_hooks = NULL,
  evaluation_hooks = NULL,
  prediction_hooks = NULL,
  training_chief_hooks = NULL,
  ...
)
```

Arguments

mode	A key that specifies whether we are performing training (`"train"`), evaluation (`"eval"`), or prediction (`"infer"`). These values can also be accessed through the `mode_keys()` object.
predictions	The prediction tensor(s).
loss	The training loss tensor. Must be either scalar, or with shape `c(1)`.
train_op	The training operation – typically, a call to `optimizer$minimize(...)`, depending on the type of optimizer used during training.
eval_metric_ops	A list of metrics to be computed as part of evaluation. This should be a named list, mapping metric names (e.g. `"rmse"`) to the operation that computes the associated metric (e.g. `tf$metrics$root_mean_squared_error(...)`). These metric operations should be evaluated without any impact on state (typically is a pure computation results based on variables). For example, it should not trigger the update ops or requires any input fetching.
training_hooks	(Available since TensorFlow v1.4) A list of session run hooks to run on all workers during training.
evaluation_hooks	(Available since TensorFlow v1.4) A list of session run hooks to run during evaluation.
prediction_hooks	(Available since TensorFlow v1.7) A list of session run hooks to run during prediciton.
training_chief_hooks	(Available since TensorFlow v1.4) A list of session run hooks to run on chief worker during training.
...	Other optional (named) arguments, to be passed to the `EstimatorSpec` constructor.

See Also

Other custom estimator methods: `estimator()`, `evaluate.tf_estimator()`, `export_savedmodel.tf_estimator()`, `predict.tf_estimator()`, `train.tf_estimator()`

`evaluate.tf_estimator` *Evaluate an Estimator*

Description

Evaluate an estimator on input data provided by an `input_fn()`.

Usage

```
## S3 method for class 'tf_estimator'
evaluate(
  object,
  input_fn,
  steps = NULL,
  checkpoint_path = NULL,
  name = NULL,
  hooks = NULL,
  simplify = TRUE,
  ...
)
```

Arguments

`object`	A TensorFlow estimator.
`input_fn`	An input function, typically generated by the `input_fn()` helper function.
`steps`	The number of steps for which the model should be evaluated on this particular `evaluate()` invocation. If `NULL` (the default), this function will either evaluate forever, or until the supplied `input_fn()` has provided all available data.
`checkpoint_path`	The path to a specific model checkpoint to be used for prediction. If `NULL` (the default), the latest checkpoint in `model_dir` is used.
`name`	Name of the evaluation if user needs to run multiple evaluations on different data sets, such as on training data vs test data. Metrics for different evaluations are saved in separate folders, and appear separately in tensorboard.
`hooks`	A list of R functions, to be used as callbacks inside the training loop. By default, `hook_history_saver(every_n_step = 10)` and `hook_progress_bar()` will be attached if not provided to save the metrics history and create the progress bar.
`simplify`	Whether to simplify evaluation results into a `tibble`, as opposed to a list. Defaults to `TRUE`.
`...`	Optional arguments passed on to the estimator's `evaluate()` method.

Details

For each step, this method will call `input_fn()` to produce a single batch of data. Evaluation continues until:

- `steps` batches are processed, or
- The `input_fn()` is exhausted of data.

Value

An R list of evaluation metrics.

See Also

Other custom estimator methods: `estimator_spec()`, `estimator()`, `export_savedmodel.tf_estimator()`, `predict.tf_estimator()`, `train.tf_estimator()`

`eval_spec` *Configuration for the eval component of* `train_and_evaluate`

Description

`EvalSpec` combines details of evaluation of the trained model as well as its export. Evaluation consists of computing metrics to judge the performance of the trained model. Export writes out the trained model on to external storage.

Usage

```
eval_spec(
  input_fn,
  steps = 100,
  name = NULL,
  hooks = NULL,
  exporters = NULL,
  start_delay_secs = 120,
  throttle_secs = 600
)
```

Arguments

`input_fn`	Evaluation input function returning a tuple of: • features - `Tensor` or dictionary of string feature name to `Tensor`. • labels - `Tensor` or dictionary of `Tensor` with labels.
`steps`	Positive number of steps for which to evaluate model. If `NULL`, evaluates until `input_fn` raises an end-of-input exception.
`name`	Name of the evaluation if user needs to run multiple evaluations on different data sets. Metrics for different evaluations are saved in separate folders, and appear separately in tensorboard.

hooks	List of session run hooks to run during evaluation.
exporters	List of Exporters, or a single one, or NULL. exporters will be invoked after each evaluation.
start_delay_secs	Start evaluating after waiting for this many seconds.
throttle_secs	Do not re-evaluate unless the last evaluation was started at least this many seconds ago. Of course, evaluation does not occur if no new checkpoints are available, hence, this is the minimum.

See Also

Other training methods: train_and_evaluate.tf_estimator(), train_spec()

---

experiment *Construct an Experiment*

Description

Construct an experiment object.

Usage

```
experiment(object, ...)
```

Arguments

object	An R object.
...	Optional arguments passed on to implementing methods.

---

export_savedmodel.tf_estimator *Save an Estimator*

Description

Save an estimator (alongside its weights) to the directory export_dir_base.

Usage

```
## S3 method for class 'tf_estimator'
export_savedmodel(
  object,
  export_dir_base,
  serving_input_receiver_fn = NULL,
  assets_extra = NULL,
  as_text = FALSE,
  checkpoint_path = NULL,
  overwrite = TRUE,
  versioned = !overwrite,
  ...
)
```

Arguments

object	A TensorFlow estimator.
export_dir_base	A string containing a directory in which to export the SavedModel.
serving_input_receiver_fn	A function that takes no argument and returns a ServingInputReceiver. Required for custom models.
assets_extra	A dict specifying how to populate the assets.extra directory within the exported SavedModel, or NULL if no extra assets are needed.
as_text	whether to write the SavedModel proto in text format.
checkpoint_path	The checkpoint path to export. If NULL (the default), the most recent checkpoint found within the model directory is chosen.
overwrite	Should the export_dir directory be overwritten?
versioned	Should the model be exported under a versioned subdirectory?
...	Optional arguments passed on to the estimator's export_savedmodel() method.

Details

This method builds a new graph by first calling the serving_input_receiver_fn to obtain feature Tensors, and then calling this Estimator's model_fn to generate the model graph based on those features. It restores the given checkpoint (or, lacking that, the most recent checkpoint) into this graph in a fresh session. Finally it creates a timestamped export directory below the given export_dir_base, and writes a SavedModel into it containing a single MetaGraphDef saved from this session. The exported MetaGraphDef will provide one SignatureDef for each element of the export_outputs dict returned from the model_fn, named using the same keys. One of these keys is always signature_constants.DEFAULT_SERVING_SIGNATURE_DEF_KEY, indicating which signature will be served when a serving request does not specify one. For each signature, the outputs are provided by the corresponding ExportOutputs, and the inputs are always the input receivers provided by the serving_input_receiver_fn. Extra assets may be written into the SavedModel via the extra_assets argument. This should be a dict, where each key gives a destination path (including

the filename) relative to the assets.extra directory. The corresponding value gives the full path of the source file to be copied. For example, the simple case of copying a single file without renaming it is specified as `{'my_asset_file.txt': '/path/to/my_asset_file.txt'}`.

Value

The path to the exported directory, as a string.

Raises

ValueError: if no serving_input_receiver_fn is provided, no export_outputs are provided, or no checkpoint can be found.

See Also

Other custom estimator methods: `estimator_spec()`, `estimator()`, `evaluate.tf_estimator()`, `predict.tf_estimator()`, `train.tf_estimator()`

`feature_columns` *Feature Columns*

Description

Constructors for feature columns. A feature column defines the expected 'shape' of an input Tensor.

Usage

```
feature_columns(..., names = NULL)
```

Arguments

`...`	One or more feature column definitions. The tidyselect package is used to power generation of feature columns.
`names`	Available feature names (for selection / pattern matching) as a character vector (or R object that implements `names()` or `colnames()`).

graph_keys *Standard Names to Use for Graph Collections*

Description

The standard library uses various well-known names to collect and retrieve values associated with a graph.

Usage

```
graph_keys()
```

Details

For example, the `tf$Optimizer` subclasses default to optimizing the variables collected undergraph_keys()$TRAINABLE_VA if `NULL` is specified, but it is also possible to pass an explicit list of variables.

The following standard keys are defined:

- `GLOBAL_VARIABLES`: the default collection of `Variable` objects, shared across distributed environment (model variables are subset of these). See `tf$global_variables` for more details. Commonly, all `TRAINABLE_VARIABLES` variables will be in `MODEL_VARIABLES`, and all `MODEL_VARIABLES` variables will be in `GLOBAL_VARIABLES`.
- `LOCAL_VARIABLES`: the subset of `Variable` objects that are local to each machine. Usually used for temporarily variables, like counters. Note: use `tf$contrib$framework$local_variable` to add to this collection.
- `MODEL_VARIABLES`: the subset of `Variable` objects that are used in the model for inference (feed forward). Note: use `tf$contrib$framework$model_variable` to add to this collection.
- `TRAINABLE_VARIABLES`: the subset of `Variable` objects that will be trained by an optimizer. See `tf$trainable_variables` for more details.
- `SUMMARIES`: the summary `Tensor` objects that have been created in the graph. See `tf$summary$merge_all` for more details.
- `QUEUE_RUNNERS`: the `QueueRunner` objects that are used to produce input for a computation. See `tf$train$start_queue_runners` for more details.
- `MOVING_AVERAGE_VARIABLES`: the subset of `Variable` objects that will also keep moving averages. See `tf$moving_average_variables` for more details.
- `REGULARIZATION_LOSSES`: regularization losses collected during graph construction. The following standard keys are defined, but their collections are **not** automatically populated as many of the others are:
  - `WEIGHTS`
  - `BIASES`
  - `ACTIVATIONS`

See Also

Other utility functions: latest_checkpoint()

Examples

```
## Not run:
graph_keys()
graph_keys()$LOSSES

## End(Not run)
```

---

hook_checkpoint_saver *Saves Checkpoints Every N Steps or Seconds*

Description

Saves Checkpoints Every N Steps or Seconds

Usage

```
hook_checkpoint_saver(
  checkpoint_dir,
  save_secs = NULL,
  save_steps = NULL,
  saver = NULL,
  checkpoint_basename = "model.ckpt",
  scaffold = NULL,
  listeners = NULL
)
```

Arguments

checkpoint_dir	The base directory for the checkpoint files.
save_secs	An integer, indicating saving checkpoints every N secs.
save_steps	An integer, indicating saving checkpoints every N steps.
saver	A saver object, used for saving.
checkpoint_basename	The base name for the checkpoint files.
scaffold	A scaffold, used to get saver object.
listeners	List of checkpoint saver listener subclass instances, used for callbacks that run immediately after the corresponding hook_checkpoint_saver callbacks, only in steps where the hook_checkpoint_saver was triggered.

See Also

Other session_run_hook wrappers: hook_global_step_waiter(), hook_history_saver(), hook_logging_tensor(), hook_nan_tensor(), hook_progress_bar(), hook_step_counter(), hook_stop_at_step(), hook_summary_saver(), session_run_hook()

---

hook_global_step_waiter *Delay Execution until Global Step Reaches to* wait_until_step.

Description

This hook delays execution until global step reaches to wait_until_step. It is used to gradually start workers in distributed settings. One example usage would be setting wait_until_step=int(K*log(task_id+1)) assuming that task_id=0 is the chief.

Usage

```
hook_global_step_waiter(wait_until_step)
```

Arguments

wait_until_step	An integer indicating that until which global step should we wait.

See Also

Other session_run_hook wrappers: hook_checkpoint_saver(), hook_history_saver(), hook_logging_tensor(), hook_nan_tensor(), hook_progress_bar(), hook_step_counter(), hook_stop_at_step(), hook_summary_saver(), session_run_hook()

---

hook_history_saver *A Custom Run Hook for Saving Metrics History*

Description

This hook allows users to save the metrics history produced during training or evaluation in a specified frequency.

Usage

```
hook_history_saver(every_n_step = 10)
```

Arguments

every_n_step	Save the metrics every N steps

See Also

Other session_run_hook wrappers: hook_checkpoint_saver(), hook_global_step_waiter(), hook_logging_tensor(), hook_nan_tensor(), hook_progress_bar(), hook_step_counter(), hook_stop_at_step(), hook_summary_saver(), session_run_hook()

---

hook_logging_tensor *Prints Given Tensors Every N Local Steps, Every N Seconds, or at End*

Description

The tensors will be printed to the log, with INFO severity.

Usage

```
hook_logging_tensor(
  tensors,
  every_n_iter = NULL,
  every_n_secs = NULL,
  formatter = NULL,
  at_end = FALSE
)
```

Arguments

tensors	A list that maps string-valued tags to tensors/tensor names.
every_n_iter	An integer value, indicating the values of tensors will be printed once every N local steps taken on the current worker.
every_n_secs	An integer or float value, indicating the values of tensors will be printed once every N seconds. Exactly one of every_n_iter and every_n_secs should be provided.
formatter	A function that takes list(tag = tensor) and returns a string. If NULL uses default printing all tensors.
at_end	A boolean value specifying whether to print the values of tensors at the end of the run.

Details

Note that if at_end is TRUE, tensors should not include any tensor whose evaluation produces a side effect such as consuming additional inputs.

See Also

Other session_run_hook wrappers: hook_checkpoint_saver(), hook_global_step_waiter(), hook_history_saver(), hook_nan_tensor(), hook_progress_bar(), hook_step_counter(), hook_stop_at_step(), hook_summary_saver(), session_run_hook()

hook_nan_tensor *NaN Loss Monitor*

Description

Monitors loss and stops training if loss is NaN. Can either fail with exception or just stop training.

Usage

```
hook_nan_tensor(loss_tensor, fail_on_nan_loss = TRUE)
```

Arguments

loss_tensor	The loss tensor.
fail_on_nan_loss	A boolean indicating whether to raise exception when loss is NaN.

See Also

Other session_run_hook wrappers: hook_checkpoint_saver(), hook_global_step_waiter(), hook_history_saver(), hook_logging_tensor(), hook_progress_bar(), hook_step_counter(), hook_stop_at_step(), hook_summary_saver(), session_run_hook()

hook_progress_bar *A Custom Run Hook to Create and Update Progress Bar During Training or Evaluation*

Description

This hook creates a progress bar that creates and updates the progress bar during training or evaluation.

Usage

```
hook_progress_bar()
```

See Also

Other session_run_hook wrappers: hook_checkpoint_saver(), hook_global_step_waiter(), hook_history_saver(), hook_logging_tensor(), hook_nan_tensor(), hook_step_counter(), hook_stop_at_step(), hook_summary_saver(), session_run_hook()

hook_step_counter *Steps per Second Monitor*

Description

Steps per Second Monitor

Usage

```
hook_step_counter(
  every_n_steps = 100,
  every_n_secs = NULL,
  output_dir = NULL,
  summary_writer = NULL
)
```

Arguments

every_n_steps	Run this counter every N steps
every_n_secs	Run this counter every N seconds
output_dir	The output directory
summary_writer	The summary writer

See Also

Other session_run_hook wrappers: hook_checkpoint_saver(), hook_global_step_waiter(), hook_history_saver(), hook_logging_tensor(), hook_nan_tensor(), hook_progress_bar(), hook_stop_at_step(), hook_summary_saver(), session_run_hook()

hook_stop_at_step *Monitor to Request Stop at a Specified Step*

Description

Monitor to Request Stop at a Specified Step

Usage

```
hook_stop_at_step(num_steps = NULL, last_step = NULL)
```

Arguments

num_steps	Number of steps to execute.
last_step	Step after which to stop.

See Also

Other session_run_hook wrappers: hook_checkpoint_saver(), hook_global_step_waiter(), hook_history_saver(), hook_logging_tensor(), hook_nan_tensor(), hook_progress_bar(), hook_step_counter(), hook_summary_saver(), session_run_hook()

`hook_summary_saver` *Saves Summaries Every N Steps*

Description

Saves Summaries Every N Steps

Usage

```
hook_summary_saver(
  save_steps = NULL,
  save_secs = NULL,
  output_dir = NULL,
  summary_writer = NULL,
  scaffold = NULL,
  summary_op = NULL
)
```

Arguments

`save_steps`	An integer indicating saving summaries every N steps. Exactly one of `save_secs` and `save_steps` should be set.
`save_secs`	An integer indicating saving summaries every N seconds.
`output_dir`	The directory to save the summaries to. Only used if no `summary_writer` is supplied.
`summary_writer`	The summary writer. If `NULL` and an `output_dir` was passed, one will be created accordingly.
`scaffold`	A scaffold to get summary_op if it's not provided.
`summary_op`	A tensor of type `tf$string` containing the serialized summary protocol buffer or a list of tensors. They are most likely an output by TensorFlow summary methods like `tf$summary$scalar` or `tf$summary$merge_all`. It can be passed in as one tensor; if more than one, they must be passed in as a list.

See Also

Other session_run_hook wrappers: hook_checkpoint_saver(), hook_global_step_waiter(), hook_history_saver(), hook_logging_tensor(), hook_nan_tensor(), hook_progress_bar(), hook_step_counter(), hook_stop_at_step(), session_run_hook()

input_fn *Construct an Input Function*

Description

This function constructs input function from various types of input used to feed different TensorFlow estimators.

Usage

```
input_fn(object, ...)

## Default S3 method:
input_fn(object, ...)

## S3 method for class 'formula'
input_fn(object, data, ...)

## S3 method for class 'data.frame'
input_fn(
  object,
  features,
  response = NULL,
  batch_size = 128,
  shuffle = "auto",
  num_epochs = 1,
  queue_capacity = 1000,
  num_threads = 1,
  ...
)

## S3 method for class 'list'
input_fn(
  object,
  features,
  response = NULL,
  batch_size = 128,
  shuffle = "auto",
  num_epochs = 1,
  queue_capacity = 1000,
  num_threads = 1,
  ...
)

## S3 method for class 'matrix'
input_fn(object, ...)
```

Arguments

object, data	An 'input source' – either a data set (e.g. an R data.frame), or another kind of object that can provide the data required for training.
...	Optional arguments passed on to implementing submethods.
features	The names of feature variables to be used.
response	The name of the response variable.
batch_size	The batch size.
shuffle	Whether to shuffle the queue. When "auto" (the default), shuffling will be performed except when this input function is called by a predict() method.
num_epochs	The number of epochs to iterate over data.
queue_capacity	The size of queue to accumulate.
num_threads	The number of threads used for reading and enqueueing. In order to have predictable and repeatable order of reading and enqueueing, such as in prediction and evaluation mode, num_threads should be 1.

Details

For list objects, this method is particularly useful when constructing dynamic length of inputs for models like recurrent neural networks. Note that some arguments are not available yet for input_fn applied to list objects. See S3 method signatures below for more details.

See Also

Other input functions: numpy_input_fn()

Examples

```
## Not run:
# Construct the input function through formula interface
input_fn1 <- input_fn(mpg ~ drat + cyl, mtcars)

## End(Not run)

## Not run:
# Construct the input function from a data.frame object
input_fn1 <- input_fn(mtcars, response = mpg, features = c(drat, cyl))

## End(Not run)

## Not run:
# Construct the input function from a list object
input_fn1 <- input_fn(
   object = list(
     feature1 = list(
       list(list(1), list(2), list(3)),
       list(list(4), list(5), list(6))),
     feature2 = list(
       list(list(7), list(8), list(9)),
```

```
      list(list(10), list(11), list(12))),
    response = list(
      list(1, 2, 3), list(4, 5, 6))),
  features = c("feature1", "feature2"),
  response = "response",
  batch_size = 10L)

## End(Not run)
```

`input_layer` *Construct an Input Layer*

Description

Returns a dense tensor as input layer based on given `feature_columns`. At the first layer of the model, this column oriented data should be converted to a single tensor.

Usage

```
input_layer(
  features,
  feature_columns,
  weight_collections = NULL,
  trainable = TRUE
)
```

Arguments

`features`	A mapping from key to tensors. Feature columns look up via these keys. For example `column_numeric('price')` will look at 'price' key in this dict. Values can be a sparse tensor or tensor depends on corresponding feature column.
`feature_columns`	An iterable containing the FeatureColumns to use as inputs to your model. All items should be instances of classes derived from a dense column such as `column_numeric()`, `column_embedding()`, `column_bucketized()`, `column_indicator()`. If you have categorical features, you can wrap them with an `column_embedding()` or `column_indicator()`.
`weight_collections`	A list of collection names to which the Variable will be added. Note that, variables will also be added to collections `graph_keys()$GLOBAL_VARIABLES` and `graph_keys()$MODEL_VARIABLES`.
`trainable`	If `TRUE` also add the variable to the graph collection `graph_keys()$TRAINABLE_VARIABLES` (see `tf$Variable`).

Value

A tensor which represents input layer of a model. Its shape is (batch_size, first_layer_dimension) and its dtype is `float32`. first_layer_dimension is determined based on given `feature_columns`.

Raises

- ValueError: if an item in `feature_columns` is not a dense column.

See Also

Other feature column constructors: `column_bucketized()`, `column_categorical_weighted()`, `column_categorical_with_hash_bucket()`, `column_categorical_with_identity()`, `column_categorical_with_vo`
`column_categorical_with_vocabulary_list()`, `column_crossed()`, `column_embedding()`, `column_numeric()`

---

keras_model_to_estimator *Keras Estimators*

Description

Create an Estimator from a compiled Keras model

Usage

```
keras_model_to_estimator(
  keras_model = NULL,
  keras_model_path = NULL,
  custom_objects = NULL,
  model_dir = NULL,
  config = NULL
)
```

Arguments

`keras_model`	A keras model.
`keras_model_path`	Directory to a keras model on disk.
`custom_objects`	Dictionary for custom objects.
`model_dir`	Directory to save Estimator model parameters, graph and etc.
`config`	Configuration object.

latest_checkpoint *Get the Latest Checkpoint in a Checkpoint Directory*

Description

Get the Latest Checkpoint in a Checkpoint Directory

Usage

```
latest_checkpoint(checkpoint_dir, ...)
```

Arguments

checkpoint_dir	The path to the checkpoint directory.
...	Optional arguments passed on to latest_checkpoint().

See Also

Other utility functions: graph_keys()

linear_estimators *Construct a Linear Estimator*

Description

Construct a linear model, which can be used to predict a continuous outcome (in the case of linear_regressor()) or a categorical outcome (in the case of linear_classifier()).

Usage

```
linear_regressor(
  feature_columns,
  model_dir = NULL,
  label_dimension = 1L,
  weight_column = NULL,
  optimizer = "Ftrl",
  config = NULL,
  partitioner = NULL
)

linear_classifier(
  feature_columns,
  model_dir = NULL,
  n_classes = 2L,
  weight_column = NULL,
```

```
  label_vocabulary = NULL,
  optimizer = "Ftrl",
  config = NULL,
  partitioner = NULL
)
```

Arguments

feature_columns	An R list containing all of the feature columns used by the model (typically, generated by feature_columns()).
model_dir	Directory to save the model parameters, graph, and so on. This can also be used to load checkpoints from the directory into a estimator to continue training a previously saved model.
label_dimension	Number of regression targets per example. This is the size of the last dimension of the labels and logits Tensor objects (typically, these have shape [batch_size, label_dimension]).
weight_column	A string, or a numeric column created by column_numeric() defining feature column representing weights. It is used to down weight or boost examples during training. It will be multiplied by the loss of the example. If it is a string, it is used as a key to fetch weight tensor from the features argument. If it is a numeric column, then the raw tensor is fetched by key weight_column$key, then weight_column$normalizer_fn is applied on it to get weight tensor.
optimizer	Either the name of the optimizer to be used when training the model, or a TensorFlow optimizer instance. Defaults to the FTRL optimizer.
config	A run configuration created by run_config(), used to configure the runtime settings.
partitioner	An optional partitioner for the input layer.
n_classes	The number of label classes.
label_vocabulary	A list of strings represents possible label values. If given, labels must be string type and have any value in label_vocabulary. If it is not given, that means labels are already encoded as integer or float within [0, 1] for n_classes == 2 and encoded as integer values in {0, 1,..., n_classes - 1} for n_classes > 2. Also there will be errors if vocabulary is not provided and labels are string.

See Also

Other canned estimators: boosted_trees_estimators, dnn_estimators, dnn_linear_combined_estimators

metric_keys *Canonical Metric Keys*

Description

The canonical set of keys that can be used to access metrics from canned estimators.

Usage

```
metric_keys()
```

See Also

Other estimator keys: `mode_keys()`, `prediction_keys()`

Examples

```
## Not run:
metrics <- metric_keys()

# Get the available keys
metrics

metrics$ACCURACY

## End(Not run)
```

model_dir *Model directory*

Description

Get the directory where a model's artifacts are stored.

Usage

```
model_dir(object, ...)
```

Arguments

`object`	Model object
`...`	Unused

mode_keys *Canonical Mode Keys*

Description

The names for different possible modes for an estimator. The following standard keys are defined:

Usage

```
mode_keys()
```

Details

TRAIN	Training mode.
EVAL	Evaluation mode.
PREDICT	Prediction / inference mode.

See Also

Other estimator keys: metric_keys(), prediction_keys()

Examples

```
## Not run:
modes <- mode_keys()
modes$TRAIN

## End(Not run)
```

numpy_input_fn *Construct Input Function Containing Python Dictionaries of Numpy Arrays*

Description

This returns a function outputting features and target based on the dict of numpy arrays. The dict features has the same keys as the x.

Usage

```
numpy_input_fn(
  x,
  y = NULL,
  batch_size = 128,
  num_epochs = 1,
  shuffle = NULL,
  queue_capacity = 1000,
  num_threads = 1
)
```

Arguments

x	dict of numpy array object.
y	numpy array object. NULL if absent.
batch_size	Integer, size of batches to return.
num_epochs	Integer, number of epochs to iterate over data. If NULL will run forever.
shuffle	Boolean, if TRUE shuffles the queue. Avoid shuffle at prediction time.
queue_capacity	Integer, size of queue to accumulate.
num_threads	Integer, number of threads used for reading and enqueueing. In order to have predicted and repeatable order of reading and enqueueing, such as in prediction and evaluation mode, num_threads should be 1. #'

Details

Note that this function is still experimental and should only be used if necessary, e.g. feed in data that's dictionary of numpy arrays.

Raises

ValueError: if the shape of y mismatches the shape of values in x (i.e., values in x have same shape).
TypeError: x is not a dict or shuffle is not bool.

See Also

Other input functions: input_fn()

---

plot.tf_estimator_history

Plot training history

Description

Plots metrics recorded during training.

Usage

```
## S3 method for class 'tf_estimator_history'
plot(
  x,
  y,
  metrics = NULL,
  method = c("auto", "ggplot2", "base"),
  smooth = getOption("tf.estimator.plot.history.smooth", TRUE),
  theme_bw = getOption("tf.estimator.plot.history.theme_bw", FALSE),
  ...
)
```

Arguments

x	Training history object returned from `train()`.
y	Unused.
metrics	One or more metrics to plot (e.g. `c('total_losses', 'mean_losses')`). 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 data points 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.

`predict.tf_estimator` *Generate Predictions with an Estimator*

Description

Generate predicted labels / values for input data provided by `input_fn()`.

Usage

```
## S3 method for class 'tf_estimator'
predict(
  object,
  input_fn,
  checkpoint_path = NULL,
  predict_keys = c("predictions", "classes", "class_ids", "logistic", "logits",
    "probabilities"),
  hooks = NULL,
  as_iterable = FALSE,
  simplify = TRUE,
```

```
  yield_single_examples = TRUE,
  ...
)
```

Arguments

object	A TensorFlow estimator.
input_fn	An input function, typically generated by the input_fn() helper function.
checkpoint_path	The path to a specific model checkpoint to be used for prediction. If NULL (the default), the latest checkpoint in model_dir is used.
predict_keys	The types of predictions that should be produced, as an R list. When this argument is not specified (the default), all possible predicted values will be returned.
hooks	A list of R functions, to be used as callbacks inside the training loop. By default, hook_history_saver(every_n_step = 10) and hook_progress_bar() will be attached if not provided to save the metrics history and create the progress bar.
as_iterable	Boolean; should a raw Python generator be returned? When FALSE (the default), the predicted values will be consumed from the generator and returned as an R object.
simplify	Whether to simplify prediction results into a tibble, as opposed to a list. Defaults to TRUE.
yield_single_examples	(Available since TensorFlow v1.7) If FALSE, yields the whole batch as returned by the model_fn instead of decomposing the batch into individual elements. This is useful if model_fn returns some tensors with first dimension not equal to the batch size.
...	Optional arguments passed on to the estimator's predict() method.

Yields

Evaluated values of predictions tensors.

Raises

ValueError: Could not find a trained model in model_dir. ValueError: if batch length of predictions are not same. ValueError: If there is a conflict between predict_keys and predictions. For example if predict_keys is not NULL but EstimatorSpec.predictions is not a dict.

See Also

Other custom estimator methods: estimator_spec(), estimator(), evaluate.tf_estimator(), export_savedmodel.tf_estimator(), train.tf_estimator()

prediction_keys *Canonical Model Prediction Keys*

Description

The canonical set of keys used for models and estimators that provide different types of predicted values through their `predict()` method.

Usage

```
prediction_keys()
```

See Also

Other estimator keys: `metric_keys()`, `mode_keys()`

Examples

```
 ## Not run:
keys <- prediction_keys()

# Get the available keys
keys

# Key for retrieving probabilities from prediction values
keys$PROBABILITIES

## End(Not run)
```

regressor_parse_example_spec *Generates Parsing Spec for TensorFlow Example to be Used with Regressors*

Description

If users keep data in `tf$Example` format, they need to call `tf$parse_example` with a proper feature spec. There are two main things that this utility helps:

- Users need to combine parsing spec of features with labels and weights (if any) since they are all parsed from same `tf$Example` instance. This utility combines these specs.
- It is difficult to map expected label by a regressor such as `dnn_regressor` to corresponding `tf$parse_example` spec. This utility encodes it by getting related information from users (key, dtype).

Usage

```
regressor_parse_example_spec(
  feature_columns,
  label_key,
  label_dtype = tf$float32,
  label_default = NULL,
  label_dimension = 1L,
  weight_column = NULL
)
```

Arguments

feature_columns	An iterable containing all feature columns. All items should be instances of classes derived from _FeatureColumn.
label_key	A string identifying the label. It means tf$Example stores labels with this key.
label_dtype	A tf$dtype identifies the type of labels. By default it is tf$float32.
label_default	used as label if label_key does not exist in given tf$Example. By default default_value is none, which means tf$parse_example will error out if there is any missing label.
label_dimension	Number of regression targets per example. This is the size of the last dimension of the labels and logits Tensor objects (typically, these have shape [batch_size, label_dimension]).
weight_column	A string or a _NumericColumn created by column_numeric defining feature column representing weights. It is used to down weight or boost examples during training. It will be multiplied by the loss of the example. If it is a string, it is used as a key to fetch weight tensor from the features. If it is a _NumericColumn, raw tensor is fetched by key weight_column$key, then weight_column$normalizer_fn is applied on it to get weight tensor.

Value

A dict mapping each feature key to a FixedLenFeature or VarLenFeature value.

Raises

- ValueError: If label is used in feature_columns.
- ValueError: If weight_column is used in feature_columns.
- ValueError: If any of the given feature_columns is not a _FeatureColumn instance.
- ValueError: If weight_column is not a _NumericColumn instance.
- ValueError: if label_key is NULL.

See Also

Other parsing utilities: classifier_parse_example_spec()

run_config *Run Configuration*

Description

This class specifies the configurations for an `Estimator` run.

Usage

```
run_config()
```

See Also

Other run_config methods: `task_type()`

Examples

```
## Not run:
config <- run_config()

# Get the properties of the config
names(config)

# Change the mutable properties of the config
config <- config$replace(tf_random_seed = 11L, save_summary_steps = 12L)

# Print config as key value pairs
print(config)

## End(Not run)
```

session_run_args *Create Session Run Arguments*

Description

Create a set of session run arguments. These are used as the return values in the `before_run(context)` callback of a `session_run_hook()`, for requesting the values of specific tensor in the `after_run(context, values)` callback.

Usage

```
session_run_args(...)
```

Arguments

...	A set of tensors or operations.

See Also

session_run_hook()

---

session_run_hook *Create Custom Session Run Hooks*

Description

Create a set of session run hooks, used to record information during training of an estimator. See **Details** for more information on the various hooks that can be defined.

Usage

```
session_run_hook(
  begin = function() { },
  after_create_session = function(session, coord) { },
  before_run = function(context) { },
  after_run = function(context, values) { },
  end = function(session) { }
)
```

Arguments

begin	function(): An R function, to be called once before using the session.
after_create_session	function(session, coord): An R function, to be called once the new TensorFlow session has been created.
before_run	function(run_context): An R function to be called before a run.
after_run	function(run_context, run_values): An R function to be called after a run.
end	function(session): An R function to be called at the end of the session.
	Typically, you'll want to define a before_run() hook that defines the set of tensors you're interested in for a particular run, and then you'll use the resulting values of those tensors in your after_run() hook. The tensors requested in your before_run() hook will be made available as part of the second argument in the after_run() hook (the values argument).

See Also

session_run_args()

Other session_run_hook wrappers: hook_checkpoint_saver(), hook_global_step_waiter(), hook_history_saver(), hook_logging_tensor(), hook_nan_tensor(), hook_progress_bar(), hook_step_counter(), hook_stop_at_step(), hook_summary_saver()

task_type *Task Types*

Description

This constant class gives the constant strings for available task types used in `run_config`.

Usage

```
task_type()
```

See Also

Other run_config methods: `run_config()`

Examples

```
## Not run:
task_type()$MASTER

## End(Not run)
```

tfestimators *High-level Estimator API in TensorFlow for R*

Description

This library provides an R interface to the Estimator API inside TensorFlow that's designed to streamline the process of creating, evaluating, and deploying general machine learning and deep learning models.

Details

TensorFlow is an open source software library for numerical computation using data flow graphs. Nodes in the graph represent mathematical operations, while the graph edges represent the multidimensional data arrays (tensors) communicated between them. The flexible architecture allows you to deploy computation to one or more CPUs or GPUs in a desktop, server, or mobile device with a single API.

The TensorFlow API is composed of a set of Python modules that enable constructing and executing TensorFlow graphs. The tensorflow package provides access to the complete TensorFlow API from within R.

For additional documentation on the tensorflow package see https://tensorflow.rstudio.com

train-evaluate-predict *Base Documentation for train, evaluate, and predict.*

Description

Base Documentation for train, evaluate, and predict.

Arguments

input_fn	An input function, typically generated by the input_fn() helper function.
hooks	A list of R functions, to be used as callbacks inside the training loop. By default, hook_history_saver(every_n_step = 10) and hook_progress_bar() will be attached if not provided to save the metrics history and create the progress bar.
checkpoint_path	The path to a specific model checkpoint to be used for prediction. If NULL (the default), the latest checkpoint in model_dir is used.

train.tf_estimator *Train an Estimator*

Description

Train an estimator on a set of input data provides by the input_fn().

Usage

```
## S3 method for class 'tf_estimator'
train(
  object,
  input_fn,
  steps = NULL,
  hooks = NULL,
  max_steps = NULL,
  saving_listeners = NULL,
  ...
)
```

Arguments

object	A TensorFlow estimator.
input_fn	An input function, typically generated by the input_fn() helper function.

steps — The number of steps for which the model should be trained on this particular `train()` invocation. If `NULL` (the default), this function will either train forever, or until the supplied `input_fn()` has provided all available data.

hooks — A list of R functions, to be used as callbacks inside the training loop. By default, `hook_history_saver(every_n_step = 10)` and `hook_progress_bar()` will be attached if not provided to save the metrics history and create the progress bar.

max_steps — The total number of steps for which the model should be trained. If set, `steps` must be `NULL`. If the estimator has already been trained a total of `max_steps` times, then no training will be performed.

saving_listeners — (Available since TensorFlow v1.4) A list of `CheckpointSaverListener` objects used for callbacks that run immediately before or after checkpoint savings.

... — Optional arguments, passed on to the estimator's `train()` method.

Value

A data.frame of the training loss history.

See Also

Other custom estimator methods: `estimator_spec()`, `estimator()`, `evaluate.tf_estimator()`, `export_savedmodel.tf_estimator()`, `predict.tf_estimator()`

---

train_and_evaluate.tf_estimator

Train and evaluate the estimator.

Description

(Available since TensorFlow v1.4)

Usage

```
## S3 method for class 'tf_estimator'
train_and_evaluate(object, train_spec, eval_spec, ...)
```

Arguments

object — An estimator object to train and evaluate.

train_spec — A `TrainSpec` instance to specify the training specification.

eval_spec — A `EvalSpec` instance to specify the evaluation and export specification.

... — Not used.

Details

This utility function trains, evaluates, and (optionally) exports the model by using the given `estimator`. All training related specification is held in `train_spec`, including training `input_fn` and training max steps, etc. All evaluation and export related specification is held in `eval_spec`, including evaluation `input_fn`, steps, etc.

This utility function provides consistent behavior for both local (non-distributed) and distributed configurations. Currently, the only supported distributed training configuration is between-graph replication.

Overfitting: In order to avoid overfitting, it is recommended to set up the training `input_fn` to shuffle the training data properly. It is also recommended to train the model a little longer, say multiple epochs, before performing evaluation, as the input pipeline starts from scratch for each training. It is particularly important for local training and evaluation.

Stop condition: In order to support both distributed and non-distributed configuration reliably, the only supported stop condition for model training is `train_spec.max_steps`. If `train_spec.max_steps` is `NULL`, the model is trained forever. *Use with care* if model stop condition is different. For example, assume that the model is expected to be trained with one epoch of training data, and the training `input_fn` is configured to throw `OutOfRangeError` after going through one epoch, which stops the `Estimator.train`. For a three-training-worker distributed configuration, each training worker is likely to go through the whole epoch independently. So, the model will be trained with three epochs of training data instead of one epoch.

Raises

- ValueError: if environment variable `TF_CONFIG` is incorrectly set.

See Also

Other training methods: `eval_spec()`, `train_spec()`

`train_spec` *Configuration for the train component of* `train_and_evaluate`

Description

`TrainSpec` determines the input data for the training, as well as the duration. Optional hooks run at various stages of training.

Usage

```
train_spec(input_fn, max_steps = NULL, hooks = NULL)
```

Arguments

input_fn	Training input function returning a tuple of: • features - Tensor or dictionary of string feature name to Tensor. • labels - Tensor or dictionary of Tensor with labels.
max_steps	Positive number of total steps for which to train model. If NULL, train forever. The training input_fn is not expected to generate OutOfRangeError or StopIteration exceptions.
hooks	List of session run hooks to run on all workers (including chief) during training.

See Also

Other training methods: eval_spec(), train_and_evaluate.tf_estimator()

---

variable_names_values *Get variable names and values associated with an estimator*

Description

These helper functions extract the names and values of variables in the graphs associated with trained estimator models.

Usage

```
variable_names(object)

variable_value(object, variable = NULL)
```

Arguments

object	A trained estimator model.
variable	(Optional) Names of variables to extract as a character vector. If not specified, values for all variables are returned.

Value

For variable_names(), a vector of variable names. For variable_values(), a named list of variable values.

Index