# Module: tf.compat.v1.feature\_column

* [**Contents**](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/compat/v1/feature_column#top_of_page)
* [Functions](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/compat/v1/feature_column#functions)

Public API for tf.feature\_column namespace.

## Functions

[bucketized\_column(...)](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/feature_column/bucketized_column): Represents discretized dense input.

[categorical\_column\_with\_hash\_bucket(...)](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/feature_column/categorical_column_with_hash_bucket): Represents sparse feature where ids are set by hashing.

[categorical\_column\_with\_identity(...)](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/feature_column/categorical_column_with_identity): A CategoricalColumn that returns identity values.

[categorical\_column\_with\_vocabulary\_file(...)](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/compat/v1/feature_column/categorical_column_with_vocabulary_file): A CategoricalColumn with a vocabulary file.

[categorical\_column\_with\_vocabulary\_list(...)](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/feature_column/categorical_column_with_vocabulary_list): A CategoricalColumn with in-memory vocabulary.

[crossed\_column(...)](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/feature_column/crossed_column): Returns a column for performing crosses of categorical features.

[embedding\_column(...)](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/feature_column/embedding_column): DenseColumn that converts from sparse, categorical input.

[indicator\_column(...)](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/feature_column/indicator_column): Represents multi-hot representation of given categorical column.

[input\_layer(...)](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/compat/v1/feature_column/input_layer): Returns a dense Tensor as input layer based on given feature\_columns.

[linear\_model(...)](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/compat/v1/feature_column/linear_model): Returns a linear prediction Tensor based on given feature\_columns.

[make\_parse\_example\_spec(...)](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/compat/v1/feature_column/make_parse_example_spec): Creates parsing spec dictionary from input feature\_columns.

[numeric\_column(...)](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/feature_column/numeric_column): Represents real valued or numerical features.

[sequence\_categorical\_column\_with\_hash\_bucket(...)](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/feature_column/sequence_categorical_column_with_hash_bucket): A sequence of categorical terms where ids are set by hashing.

[sequence\_categorical\_column\_with\_identity(...)](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/feature_column/sequence_categorical_column_with_identity): Returns a feature column that represents sequences of integers.

[sequence\_categorical\_column\_with\_vocabulary\_file(...)](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/feature_column/sequence_categorical_column_with_vocabulary_file): A sequence of categorical terms where ids use a vocabulary file.

[sequence\_categorical\_column\_with\_vocabulary\_list(...)](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/feature_column/sequence_categorical_column_with_vocabulary_list): A sequence of categorical terms where ids use an in-memory list.

[sequence\_numeric\_column(...)](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/feature_column/sequence_numeric_column): Returns a feature column that represents sequences of numeric data.

[shared\_embedding\_columns(...)](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/compat/v1/feature_column/shared_embedding_columns): List of dense columns that convert from sparse, categorical input.

[weighted\_categorical\_column(...)](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/feature_column/weighted_categorical_column): Applies weight values to a CategoricalColumn.

# tf.compat.v1.feature\_column.input\_layer

Returns a dense Tensor as input layer based on given feature\_columns.

tf.compat.v1.feature\_column.input\_layer(  
    features,  
    feature\_columns,  
    weight\_collections=None,  
    trainable=True,  
    cols\_to\_vars=None,  
    cols\_to\_output\_tensors=None  
)

Defined in [python/feature\_column/feature\_column.py](https://github.com/tensorflow/tensorflow/tree/r2.0/tensorflow/python/feature_column/feature_column.py).

Generally a single example in training data is described with FeatureColumns. At the first layer of the model, this column oriented data should be converted to a single Tensor.

#### Example:

price = numeric\_column('price')  
keywords\_embedded = embedding\_column(  
    categorical\_column\_with\_hash\_bucket("keywords", 10K), dimensions=16)  
columns = [price, keywords\_embedded, ...]  
features = tf.io.parse\_example(..., features=make\_parse\_example\_spec(columns))  
dense\_tensor = input\_layer(features, columns)  
for units in [128, 64, 32]:  
  dense\_tensor = tf.compat.v1.layers.dense(dense\_tensor, units, tf.nn.relu)  
prediction = tf.compat.v1.layers.dense(dense\_tensor, 1)

#### Args:

* **features**: A mapping from key to tensors. \_FeatureColumns look up via these keys. For example numeric\_column('price') will look at 'price' key in this dict. Values can be a SparseTensor or a Tensor depends on corresponding \_FeatureColumn.
* **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.
* **weight\_collections**: A list of collection names to which the Variable will be added. Note that variables will also be added to collections tf.GraphKeys.GLOBAL\_VARIABLES and ops.GraphKeys.MODEL\_VARIABLES.
* **trainable**: If True also add the variable to the graph collectionGraphKeys.TRAINABLE\_VARIABLES (see [tf.Variable](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/Variable)).
* **cols\_to\_vars**: If not None, must be a dictionary that will be filled with a mapping from \_FeatureColumn to list of Variables. For example, after the call, we might have cols\_to\_vars = {\_EmbeddingColumn( categorical\_column=\_HashedCategoricalColumn( key='sparse\_feature', hash\_bucket\_size=5, dtype=tf.string), dimension=10): [<tf.Variable 'some\_variable:0' shape=(5, 10), <tf.Variable 'some\_variable:1' shape=(5, 10)]} If a column creates no variables, its value will be an empty list.
* **cols\_to\_output\_tensors**: If not None, must be a dictionary that will be filled with a mapping from '\_FeatureColumn' to the associated output Tensors.

#### Returns:

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 \_DenseColumn.

# tf.compat.v1.feature\_column.linear\_model

Returns a linear prediction Tensor based on given feature\_columns.

tf.compat.v1.feature\_column.linear\_model(  
    features,  
    feature\_columns,  
    units=1,  
    sparse\_combiner='sum',  
    weight\_collections=None,  
    trainable=True,  
    cols\_to\_vars=None  
)

Defined in [python/feature\_column/feature\_column.py](https://github.com/tensorflow/tensorflow/tree/r2.0/tensorflow/python/feature_column/feature_column.py).

This function generates a weighted sum based on output dimension units. Weighted sum refers to logits in classification problems. It refers to the prediction itself for linear regression problems.

Note on supported columns: linear\_model treats categorical columns as indicator\_columns. To be specific, assume the input as SparseTensor looks like:

  shape = [2, 2]  
  {  
      [0, 0]: "a"  
      [1, 0]: "b"  
      [1, 1]: "c"  
  }

linear\_model assigns weights for the presence of "a", "b", "c' implicitly, just like indicator\_column, while input\_layer explicitly requires wrapping each of categorical columns with an embedding\_column or an indicator\_column.

#### Example of usage:

price = numeric\_column('price')  
price\_buckets = bucketized\_column(price, boundaries=[0., 10., 100., 1000.])  
keywords = categorical\_column\_with\_hash\_bucket("keywords", 10K)  
keywords\_price = crossed\_column('keywords', price\_buckets, ...)  
columns = [price\_buckets, keywords, keywords\_price ...]  
features = tf.io.parse\_example(..., features=make\_parse\_example\_spec(columns))  
prediction = linear\_model(features, columns)

The sparse\_combiner argument works as follows For example, for two features represented as the categorical columns:

  # Feature 1  
  
  shape = [2, 2]  
  {  
      [0, 0]: "a"  
      [0, 1]: "b"  
      [1, 0]: "c"  
  }  
  
  # Feature 2  
  
  shape = [2, 3]  
  {  
      [0, 0]: "d"  
      [1, 0]: "e"  
      [1, 1]: "f"  
      [1, 2]: "f"  
  }

with sparse\_combiner as "mean", the linear model outputs consequently are:

  y\_0 = 1.0 / 2.0 \* ( w\_a + w\_b ) + w\_d + b  
  y\_1 = w\_c + 1.0 / 3.0 \* ( w\_e + 2.0 \* w\_f ) + b

where y\_i is the output, b is the bias, and w\_x is the weight assigned to the presence of x in the input features.

#### Args:

* **features**: A mapping from key to tensors. \_FeatureColumns look up via these keys. For example numeric\_column('price') will look at 'price' key in this dict. Values are Tensor or SparseTensor depending on corresponding \_FeatureColumn.
* **feature\_columns**: An iterable containing the FeatureColumns to use as inputs to your model. All items should be instances of classes derived from \_FeatureColumns.
* **units**: An integer, dimensionality of the output space. Default value is 1.
* **sparse\_combiner**: A string specifying how to reduce if a categorical column is multivalent. Except numeric\_column, almost all columns passed to linear\_model are considered as categorical columns. It combines each categorical column independently. Currently "mean", "sqrtn" and "sum" are supported, with "sum" the default for linear model. "sqrtn" often achieves good accuracy, in particular with bag-of-words columns.
  + "sum": do not normalize features in the column
  + "mean": do l1 normalization on features in the column
  + "sqrtn": do l2 normalization on features in the column
* **weight\_collections**: A list of collection names to which the Variable will be added. Note that, variables will also be added to collections tf.GraphKeys.GLOBAL\_VARIABLES and ops.GraphKeys.MODEL\_VARIABLES.
* **trainable**: If True also add the variable to the graph collectionGraphKeys.TRAINABLE\_VARIABLES (see [tf.Variable](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/Variable)).
* **cols\_to\_vars**: If not None, must be a dictionary that will be filled with a mapping from \_FeatureColumn to associated list of Variables. For example, after the call, we might have cols\_to\_vars = { \_NumericColumn( key='numeric\_feature1', shape=(1,): [], 'bias': [], \_NumericColumn( key='numeric\_feature2', shape=(2,)): []} If a column creates no variables, its value will be an empty list. Note that cols\_to\_vars will also contain a string key 'bias' that maps to a list of Variables.

#### Returns:

A Tensor which represents predictions/logits of a linear model. Its shape is (batch\_size, units) and its dtype is float32.

#### Raises:

* **ValueError**: if an item in feature\_columns is neither a \_DenseColumn nor \_CategoricalColumn.

# tf.compat.v1.feature\_column.shared\_embedding\_columns

List of dense columns that convert from sparse, categorical input.

tf.compat.v1.feature\_column.shared\_embedding\_columns(  
    categorical\_columns,  
    dimension,  
    combiner='mean',  
    initializer=None,  
    shared\_embedding\_collection\_name=None,  
    ckpt\_to\_load\_from=None,  
    tensor\_name\_in\_ckpt=None,  
    max\_norm=None,  
    trainable=True  
)

Defined in [python/feature\_column/feature\_column\_v2.py](https://github.com/tensorflow/tensorflow/tree/r2.0/tensorflow/python/feature_column/feature_column_v2.py).

This is similar to embedding\_column, except that it produces a list of embedding columns that share the same embedding weights.

Use this when your inputs are sparse and of the same type (e.g. watched and impression video IDs that share the same vocabulary), and you want to convert them to a dense representation (e.g., to feed to a DNN).

Inputs must be a list of categorical columns created by any of the categorical\_column\_\* function. They must all be of the same type and have the same arguments except key. E.g. they can be categorical\_column\_with\_vocabulary\_file with the same vocabulary\_file. Some or all columns could also be weighted\_categorical\_column.

Here is an example embedding of two features for a DNNClassifier model:

watched\_video\_id = categorical\_column\_with\_vocabulary\_file(  
    'watched\_video\_id', video\_vocabulary\_file, video\_vocabulary\_size)  
impression\_video\_id = categorical\_column\_with\_vocabulary\_file(  
    'impression\_video\_id', video\_vocabulary\_file, video\_vocabulary\_size)  
columns = shared\_embedding\_columns(  
    [watched\_video\_id, impression\_video\_id], dimension=10)  
  
estimator = tf.estimator.DNNClassifier(feature\_columns=columns, ...)  
  
label\_column = ...  
def input\_fn():  
  features = tf.io.parse\_example(  
      ..., features=make\_parse\_example\_spec(columns + [label\_column]))  
  labels = features.pop(label\_column.name)  
  return features, labels  
  
estimator.train(input\_fn=input\_fn, steps=100)

Here is an example using shared\_embedding\_columns with model\_fn:

def model\_fn(features, ...):  
  watched\_video\_id = categorical\_column\_with\_vocabulary\_file(  
      'watched\_video\_id', video\_vocabulary\_file, video\_vocabulary\_size)  
  impression\_video\_id = categorical\_column\_with\_vocabulary\_file(  
      'impression\_video\_id', video\_vocabulary\_file, video\_vocabulary\_size)  
  columns = shared\_embedding\_columns(  
      [watched\_video\_id, impression\_video\_id], dimension=10)  
  dense\_tensor = input\_layer(features, columns)  
  # Form DNN layers, calculate loss, and return EstimatorSpec.  
  ...

#### Args:

* **categorical\_columns**: List of categorical columns created by acategorical\_column\_with\_\* function. These columns produce the sparse IDs that are inputs to the embedding lookup. All columns must be of the same type and have the same arguments except key. E.g. they can be categorical\_column\_with\_vocabulary\_file with the same vocabulary\_file. Some or all columns could also be weighted\_categorical\_column.
* **dimension**: An integer specifying dimension of the embedding, must be > 0.
* **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. For more information, see tf.embedding\_lookup\_sparse.
* **initializer**: A variable initializer function to be used in embedding variable initialization. If not specified, defaults to [tf.compat.v1.truncated\_normal\_initializer](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/compat/v1/truncated_normal_initializer) with mean 0.0 and standard deviation 1/sqrt(dimension).
* **shared\_embedding\_collection\_name**: Optional name of the collection where shared embedding weights are added. If not given, a reasonable name will be chosen based on the names of categorical\_columns. This is also used in variable\_scope when creating shared embedding weights.
* **ckpt\_to\_load\_from**: String representing checkpoint name/pattern from which to restore column weights. Required if tensor\_name\_in\_ckpt is not None.
* **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 None.
* **max\_norm**: If not None, each embedding is clipped if its l2-norm is larger than this value, before combining.
* **trainable**: Whether or not the embedding is trainable. Default is True.

#### Returns:

A list of dense columns that converts from sparse input. The order of results follows the ordering of categorical\_columns.

#### Raises:

* **ValueError**: if dimension not > 0.
* **ValueError**: if any of the given categorical\_columns is of different type or has different arguments than the others.
* **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.
* **RuntimeError**: if eager execution is enabled.

# tf.feature\_column.bucketized\_column

* [**Contents**](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/feature_column/bucketized_column#top_of_page)
* [Aliases:](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/feature_column/bucketized_column#aliases)
* [Used in the tutorials:](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/feature_column/bucketized_column#used_in_the_tutorials)

Represents discretized dense input.

### Aliases:

* tf.compat.v1.feature\_column.bucketized\_column
* tf.compat.v2.feature\_column.bucketized\_column
* tf.feature\_column.bucketized\_column

tf.feature\_column.bucketized\_column(  
    source\_column,  
    boundaries  
)

Defined in [python/feature\_column/feature\_column\_v2.py](https://github.com/tensorflow/tensorflow/tree/r2.0/tensorflow/python/feature_column/feature_column_v2.py).

### Used in the tutorials:

* [Classify structured data](https://www.tensorflow.org/beta/tutorials/keras/feature_columns)

Buckets include the left boundary, and exclude the right boundary. Namely, boundaries=[0., 1., 2.] generates buckets (-inf, 0.), [0., 1.), [1., 2.), and [2., +inf).

For example, if the inputs are

boundaries = [0, 10, 100]  
input tensor = [[-5, 10000]  
                [150,   10]  
                [5,    100]]

then the output will be

output = [[0, 3]  
          [3, 2]  
          [1, 3]]

#### Example:

price = numeric\_column('price')  
bucketized\_price = bucketized\_column(price, boundaries=[...])  
columns = [bucketized\_price, ...]  
features = tf.io.parse\_example(..., features=make\_parse\_example\_spec(columns))  
linear\_prediction = linear\_model(features, columns)  
  
# or  
columns = [bucketized\_price, ...]  
features = tf.io.parse\_example(..., features=make\_parse\_example\_spec(columns))  
dense\_tensor = input\_layer(features, columns)

bucketized\_column can also be crossed with another categorical column using crossed\_column:

price = numeric\_column('price')  
# bucketized\_column converts numerical feature to a categorical one.  
bucketized\_price = bucketized\_column(price, boundaries=[...])  
# 'keywords' is a string feature.  
price\_x\_keywords = crossed\_column([bucketized\_price, 'keywords'], 50K)  
columns = [price\_x\_keywords, ...]  
features = tf.io.parse\_example(..., features=make\_parse\_example\_spec(columns))  
linear\_prediction = linear\_model(features, columns)

#### Args:

* **source\_column**: A one-dimensional dense column which is generated with numeric\_column.
* **boundaries**: A sorted list or tuple of floats specifying the boundaries.

#### Returns:

A BucketizedColumn.

#### 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 tuple.

# tf.feature\_column.categorical\_column\_with\_hash\_bucket

* [**Contents**](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/feature_column/categorical_column_with_hash_bucket#top_of_page)
* [Aliases:](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/feature_column/categorical_column_with_hash_bucket#aliases)
* [Used in the tutorials:](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/feature_column/categorical_column_with_hash_bucket#used_in_the_tutorials)

Represents sparse feature where ids are set by hashing.

### Aliases:

* tf.compat.v1.feature\_column.categorical\_column\_with\_hash\_bucket
* tf.compat.v2.feature\_column.categorical\_column\_with\_hash\_bucket
* tf.feature\_column.categorical\_column\_with\_hash\_bucket

tf.feature\_column.categorical\_column\_with\_hash\_bucket(  
    key,  
    hash\_bucket\_size,  
    dtype=tf.dtypes.string  
)

Defined in [python/feature\_column/feature\_column\_v2.py](https://github.com/tensorflow/tensorflow/tree/r2.0/tensorflow/python/feature_column/feature_column_v2.py).

### Used in the tutorials:

* [Classify structured data](https://www.tensorflow.org/beta/tutorials/keras/feature_columns)

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 string type input. For int type input, the value is converted to its string representation first and then hashed by the same formula.

For input dictionary features, features[key] is either Tensor or SparseTensor. If Tensor, missing values can be represented by -1 for int and '' for string, which will be dropped by this feature column.

#### Example:

keywords = categorical\_column\_with\_hash\_bucket("keywords", 10K)  
columns = [keywords, ...]  
features = tf.io.parse\_example(..., features=make\_parse\_example\_spec(columns))  
linear\_prediction = linear\_model(features, columns)  
  
# or  
keywords\_embedded = embedding\_column(keywords, 16)  
columns = [keywords\_embedded, ...]  
features = tf.io.parse\_example(..., features=make\_parse\_example\_spec(columns))  
dense\_tensor = input\_layer(features, columns)

#### Args:

* **key**: A unique string identifying the input feature. It is used as the column name and the dictionary key for feature parsing configs, feature Tensor objects, 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.

#### Returns:

A HashedCategoricalColumn.

#### Raises:

* **ValueError**: hash\_bucket\_size is not greater than 1.
* **ValueError**: dtype is neither string nor integer.

# tf.feature\_column.categorical\_column\_with\_identity

* [**Contents**](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/feature_column/categorical_column_with_identity#top_of_page)
* [Aliases:](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/feature_column/categorical_column_with_identity#aliases)

A CategoricalColumn that returns identity values.

### Aliases:

* tf.compat.v1.feature\_column.categorical\_column\_with\_identity
* tf.compat.v2.feature\_column.categorical\_column\_with\_identity
* tf.feature\_column.categorical\_column\_with\_identity

tf.feature\_column.categorical\_column\_with\_identity(  
    key,  
    num\_buckets,  
    default\_value=None  
)

Defined in [python/feature\_column/feature\_column\_v2.py](https://github.com/tensorflow/tensorflow/tree/r2.0/tensorflow/python/feature_column/feature_column_v2.py).

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.

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 categorical\_column\_with\_hash\_bucket in that case.

For input dictionary features, features[key] is either Tensor or SparseTensor. If Tensor, missing values can be represented by -1 for int and '' for string, which will be dropped by this feature column.

In the following examples, each input in the range [0, 1000000) is assigned the same value. All other inputs are assigned default\_value 0. Note that a literal 0 in inputs will result in the same default ID.

#### Linear model:

video\_id = categorical\_column\_with\_identity(  
    key='video\_id', num\_buckets=1000000, default\_value=0)  
columns = [video\_id, ...]  
features = tf.io.parse\_example(..., features=make\_parse\_example\_spec(columns))  
linear\_prediction, \_, \_ = linear\_model(features, columns)

Embedding for a DNN model:

columns = [embedding\_column(video\_id, 9),...]  
features = tf.io.parse\_example(..., features=make\_parse\_example\_spec(columns))  
dense\_tensor = input\_layer(features, columns)

#### Args:

* **key**: A unique string identifying the input feature. It is used as the column name and the dictionary key for feature parsing configs, feature Tensor objects, and feature columns.
* **num\_buckets**: Range of inputs and outputs is [0, num\_buckets).
* **default\_value**: If None, 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.

#### Returns:

A CategoricalColumn that returns identity values.

#### Raises:

* **ValueError**: if num\_buckets is less than one.
* **ValueError**: if default\_value is not in range [0, num\_buckets).

# tf.feature\_column.categorical\_column\_with\_vocabulary\_file

* [**Contents**](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/feature_column/categorical_column_with_vocabulary_file#top_of_page)
* [Aliases:](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/feature_column/categorical_column_with_vocabulary_file#aliases)

A CategoricalColumn with a vocabulary file.

### Aliases:

* tf.compat.v2.feature\_column.categorical\_column\_with\_vocabulary\_file
* tf.feature\_column.categorical\_column\_with\_vocabulary\_file

tf.feature\_column.categorical\_column\_with\_vocabulary\_file(  
    key,  
    vocabulary\_file,  
    vocabulary\_size=None,  
    dtype=tf.dtypes.string,  
    default\_value=None,  
    num\_oov\_buckets=0  
)

Defined in [python/feature\_column/feature\_column\_v2.py](https://github.com/tensorflow/tensorflow/tree/r2.0/tensorflow/python/feature_column/feature_column_v2.py).

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) ofnum\_oov\_buckets and default\_value to specify how to include out-of-vocabulary values.

For input dictionary features, features[key] is either Tensor or SparseTensor. If Tensor, missing values can be represented by -1 for int and '' for string, which will be dropped by this feature column.

Example with num\_oov\_buckets: File '/us/states.txt' contains 50 lines, each with a 2-character U.S. state abbreviation. All inputs with values in that file are assigned an ID 0-49, corresponding to its line number. All other values are hashed and assigned an ID 50-54.

states = categorical\_column\_with\_vocabulary\_file(  
    key='states', vocabulary\_file='/us/states.txt', vocabulary\_size=50,  
    num\_oov\_buckets=5)  
columns = [states, ...]  
features = tf.io.parse\_example(..., features=make\_parse\_example\_spec(columns))  
linear\_prediction = linear\_model(features, columns)

Example with default\_value: File '/us/states.txt' contains 51 lines - the first line is 'XX', and the other 50 each have a 2-character U.S. state abbreviation. Both a literal 'XX' in input, and other values missing from the file, will be assigned ID 0. All others are assigned the corresponding line number 1-50.

states = categorical\_column\_with\_vocabulary\_file(  
    key='states', vocabulary\_file='/us/states.txt', vocabulary\_size=51,  
    default\_value=0)  
columns = [states, ...]  
features = tf.io.parse\_example(..., features=make\_parse\_example\_spec(columns))  
linear\_prediction, \_, \_ = linear\_model(features, columns)

And to make an embedding with either:

columns = [embedding\_column(states, 3),...]  
features = tf.io.parse\_example(..., features=make\_parse\_example\_spec(columns))  
dense\_tensor = input\_layer(features, columns)

#### Args:

* **key**: A unique string identifying the input feature. It is used as the column name and the dictionary key for feature parsing configs, feature Tensor objects, 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. If None, it is set to the length of vocabulary\_file.
* **dtype**: The type of features. Only string and integer types are supported.
* **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.
* **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\_buckets) based on a hash of the input value. A positive num\_oov\_buckets can not be specified with default\_value.

#### Returns:

A CategoricalColumn with a vocabulary file.

#### Raises:

* **ValueError**: vocabulary\_file is missing or cannot be opened.
* **ValueError**: vocabulary\_size is missing or < 1.
* **ValueError**: num\_oov\_buckets is a negative integer.
* **ValueError**: num\_oov\_buckets and default\_value are both specified.
* **ValueError**: dtype is neither string nor integer.

# tf.feature\_column.categorical\_column\_with\_vocabulary\_list

* [**Contents**](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/feature_column/categorical_column_with_vocabulary_list#top_of_page)
* [Aliases:](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/feature_column/categorical_column_with_vocabulary_list#aliases)
* [Used in the tutorials:](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/feature_column/categorical_column_with_vocabulary_list#used_in_the_tutorials)

A CategoricalColumn with in-memory vocabulary.

### Aliases:

* tf.compat.v1.feature\_column.categorical\_column\_with\_vocabulary\_list
* tf.compat.v2.feature\_column.categorical\_column\_with\_vocabulary\_list
* tf.feature\_column.categorical\_column\_with\_vocabulary\_list

tf.feature\_column.categorical\_column\_with\_vocabulary\_list(  
    key,  
    vocabulary\_list,  
    dtype=None,  
    default\_value=-1,  
    num\_oov\_buckets=0  
)

Defined in [python/feature\_column/feature\_column\_v2.py](https://github.com/tensorflow/tensorflow/tree/r2.0/tensorflow/python/feature_column/feature_column_v2.py).

### Used in the tutorials:

* [Build a linear model with Estimators](https://www.tensorflow.org/beta/tutorials/estimators/linear)
* [Classify structured data](https://www.tensorflow.org/beta/tutorials/keras/feature_columns)

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 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 SparseTensor. If Tensor, missing values can be represented by -1 for int and '' for string, which will be dropped by this feature column.

Example with num\_oov\_buckets: In the following example, each input in vocabulary\_list is assigned an ID 0-3 corresponding to its index (e.g., input 'B' produces output 2). All other inputs are hashed and assigned an ID 4-5.

colors = categorical\_column\_with\_vocabulary\_list(  
    key='colors', vocabulary\_list=('R', 'G', 'B', 'Y'),  
    num\_oov\_buckets=2)  
columns = [colors, ...]  
features = tf.io.parse\_example(..., features=make\_parse\_example\_spec(columns))  
linear\_prediction, \_, \_ = linear\_model(features, columns)

Example with default\_value: In the following example, each input in vocabulary\_list is assigned an ID 0-4 corresponding to its index (e.g., input 'B' produces output 3). All other inputs are assigned default\_value 0.

colors = categorical\_column\_with\_vocabulary\_list(  
    key='colors', vocabulary\_list=('X', 'R', 'G', 'B', 'Y'), default\_value=0)  
columns = [colors, ...]  
features = tf.io.parse\_example(..., features=make\_parse\_example\_spec(columns))  
linear\_prediction, \_, \_ = linear\_model(features, columns)

And to make an embedding with either:

columns = [embedding\_column(colors, 3),...]  
features = tf.io.parse\_example(..., features=make\_parse\_example\_spec(columns))  
dense\_tensor = input\_layer(features, columns)

#### Args:

* **key**: A unique string identifying the input feature. It is used as the column name and the dictionary key for feature parsing configs, feature Tensor objects, 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 None, it will be inferred from vocabulary\_list.
* **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.
* **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 [len(vocabulary\_list), len(vocabulary\_list)+num\_oov\_buckets) based on a hash of the input value. A positive num\_oov\_buckets can not be specified with default\_value.

#### Returns:

A CategoricalColumn with in-memory vocabulary.

#### Raises:

* **ValueError**: if vocabulary\_list is empty, or contains duplicate keys.
* **ValueError**: num\_oov\_buckets is a negative integer.
* **ValueError**: num\_oov\_buckets and default\_value are both specified.
* **ValueError**: if dtype is not integer or string.

# tf.feature\_column.crossed\_column

* [**Contents**](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/feature_column/crossed_column#top_of_page)
* [Aliases:](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/feature_column/crossed_column#aliases)
* [Used in the tutorials:](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/feature_column/crossed_column#used_in_the_tutorials)

Returns a column for performing crosses of categorical features.

### Aliases:

* tf.compat.v1.feature\_column.crossed\_column
* tf.compat.v2.feature\_column.crossed\_column
* tf.feature\_column.crossed\_column

tf.feature\_column.crossed\_column(  
    keys,  
    hash\_bucket\_size,  
    hash\_key=None  
)

Defined in [python/feature\_column/feature\_column\_v2.py](https://github.com/tensorflow/tensorflow/tree/r2.0/tensorflow/python/feature_column/feature_column_v2.py).

### Used in the tutorials:

* [Build a linear model with Estimators](https://www.tensorflow.org/beta/tutorials/estimators/linear)
* [Classify structured data](https://www.tensorflow.org/beta/tutorials/keras/feature_columns)

Crossed features will be hashed according to hash\_bucket\_size. Conceptually, the transformation can be thought of as: Hash(cartesian product of features) % hash\_bucket\_size

For example, if the input features are:

* SparseTensor referred by first key:

shape = [2, 2]  
{  
    [0, 0]: "a"  
    [1, 0]: "b"  
    [1, 1]: "c"  
}

* SparseTensor referred by second key:

shape = [2, 1]  
{  
    [0, 0]: "d"  
    [1, 0]: "e"  
}

then crossed feature will look like:

 shape = [2, 2]  
{  
    [0, 0]: Hash64("d", Hash64("a")) % hash\_bucket\_size  
    [1, 0]: Hash64("e", Hash64("b")) % hash\_bucket\_size  
    [1, 1]: Hash64("e", Hash64("c")) % hash\_bucket\_size  
}

Here is an example to create a linear model with crosses of string features:

keywords\_x\_doc\_terms = crossed\_column(['keywords', 'doc\_terms'], 50K)  
columns = [keywords\_x\_doc\_terms, ...]  
features = tf.io.parse\_example(..., features=make\_parse\_example\_spec(columns))  
linear\_prediction = linear\_model(features, columns)

You could also use vocabulary lookup before crossing:

keywords = categorical\_column\_with\_vocabulary\_file(  
    'keywords', '/path/to/vocabulary/file', vocabulary\_size=1K)  
keywords\_x\_doc\_terms = crossed\_column([keywords, 'doc\_terms'], 50K)  
columns = [keywords\_x\_doc\_terms, ...]  
features = tf.io.parse\_example(..., features=make\_parse\_example\_spec(columns))  
linear\_prediction = linear\_model(features, columns)

If an input feature is of numeric type, you can use categorical\_column\_with\_identity, or bucketized\_column, as in the example:

# vertical\_id is an integer categorical feature.  
vertical\_id = categorical\_column\_with\_identity('vertical\_id', 10K)  
price = numeric\_column('price')  
# bucketized\_column converts numerical feature to a categorical one.  
bucketized\_price = bucketized\_column(price, boundaries=[...])  
vertical\_id\_x\_price = crossed\_column([vertical\_id, bucketized\_price], 50K)  
columns = [vertical\_id\_x\_price, ...]  
features = tf.io.parse\_example(..., features=make\_parse\_example\_spec(columns))  
linear\_prediction = linear\_model(features, columns)

To use crossed column in DNN model, you need to add it in an embedding column as in this example:

vertical\_id\_x\_price = crossed\_column([vertical\_id, bucketized\_price], 50K)  
vertical\_id\_x\_price\_embedded = embedding\_column(vertical\_id\_x\_price, 10)  
dense\_tensor = input\_layer(features, [vertical\_id\_x\_price\_embedded, ...])

#### Args:

* **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.
  + CategoricalColumn: Will use the transformed tensor produced by this column. Does not support hashed categorical column.
* **hash\_bucket\_size**: An int > 1. The number of buckets.
* **hash\_key**: Specify the hash\_key that will be used by the FingerprintCat64 function to combine the crosses fingerprints on SparseCrossOp (optional).

#### Returns:

A CrossedColumn.

#### Raises:

* **ValueError**: If len(keys) < 2.
* **ValueError**: If any of the keys is neither a string nor CategoricalColumn.
* **ValueError**: If any of the keys is HashedCategoricalColumn.
* **ValueError**: If hash\_bucket\_size < 1.

# tf.feature\_column.embedding\_column

* [**Contents**](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/feature_column/embedding_column#top_of_page)
* [Aliases:](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/feature_column/embedding_column#aliases)
* [Used in the tutorials:](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/feature_column/embedding_column#used_in_the_tutorials)

DenseColumn that converts from sparse, categorical input.

### Aliases:

* tf.compat.v1.feature\_column.embedding\_column
* tf.compat.v2.feature\_column.embedding\_column
* tf.feature\_column.embedding\_column

tf.feature\_column.embedding\_column(  
    categorical\_column,  
    dimension,  
    combiner='mean',  
    initializer=None,  
    ckpt\_to\_load\_from=None,  
    tensor\_name\_in\_ckpt=None,  
    max\_norm=None,  
    trainable=True  
)

Defined in [python/feature\_column/feature\_column\_v2.py](https://github.com/tensorflow/tensorflow/tree/r2.0/tensorflow/python/feature_column/feature_column_v2.py).

### Used in the tutorials:

* [Classify structured data](https://www.tensorflow.org/beta/tutorials/keras/feature_columns)

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 CategoricalColumn created by any of the categorical\_column\_\* function. Here is an example of using embedding\_column with DNNClassifier:

video\_id = categorical\_column\_with\_identity(  
    key='video\_id', num\_buckets=1000000, default\_value=0)  
columns = [embedding\_column(video\_id, 9),...]  
  
estimator = tf.estimator.DNNClassifier(feature\_columns=columns, ...)  
  
label\_column = ...  
def input\_fn():  
  features = tf.io.parse\_example(  
      ..., features=make\_parse\_example\_spec(columns + [label\_column]))  
  labels = features.pop(label\_column.name)  
  return features, labels  
  
estimator.train(input\_fn=input\_fn, steps=100)

Here is an example using embedding\_column with model\_fn:

def model\_fn(features, ...):  
  video\_id = categorical\_column\_with\_identity(  
      key='video\_id', num\_buckets=1000000, default\_value=0)  
  columns = [embedding\_column(video\_id, 9),...]  
  dense\_tensor = input\_layer(features, columns)  
  # Form DNN layers, calculate loss, and return EstimatorSpec.  
  ...

#### Args:

* **categorical\_column**: A CategoricalColumn created by a categorical\_column\_with\_\*function. This column produces the sparse IDs that are inputs to the embedding lookup.
* **dimension**: An integer specifying dimension of the embedding, must be > 0.
* **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. For more information, see tf.embedding\_lookup\_sparse.
* **initializer**: A variable initializer function to be used in embedding variable initialization. If not specified, defaults to [tf.compat.v1.truncated\_normal\_initializer](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/compat/v1/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 None.
* **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 None.
* **max\_norm**: If not None, embedding values are l2-normalized to this value.
* **trainable**: Whether or not the embedding is trainable. Default is True.

#### Returns:

DenseColumn 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.
* **RuntimeError**: If eager execution is enabled.

# tf.feature\_column.indicator\_column

* [**Contents**](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/feature_column/indicator_column#top_of_page)
* [Aliases:](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/feature_column/indicator_column#aliases)
* [Used in the tutorials:](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/feature_column/indicator_column#used_in_the_tutorials)

Represents multi-hot representation of given categorical column.

### Aliases:

* tf.compat.v1.feature\_column.indicator\_column
* tf.compat.v2.feature\_column.indicator\_column
* tf.feature\_column.indicator\_column

tf.feature\_column.indicator\_column(categorical\_column)

Defined in [python/feature\_column/feature\_column\_v2.py](https://github.com/tensorflow/tensorflow/tree/r2.0/tensorflow/python/feature_column/feature_column_v2.py).

### Used in the tutorials:

* [Build a linear model with Estimators](https://www.tensorflow.org/beta/tutorials/estimators/linear)
* [Classify structured data](https://www.tensorflow.org/beta/tutorials/keras/feature_columns)
* For DNN model, indicator\_column can be used to wrap any categorical\_column\_\* (e.g., to feed to DNN). Consider to Use embedding\_column if the number of buckets/unique(values) are large.
* For Wide (aka linear) model, indicator\_column is the internal representation for categorical column when passing categorical column directly (as any element in feature\_columns) to linear\_model. See linear\_model for details.

name = indicator\_column(categorical\_column\_with\_vocabulary\_list(  
    'name', ['bob', 'george', 'wanda'])  
columns = [name, ...]  
features = tf.io.parse\_example(..., features=make\_parse\_example\_spec(columns))  
dense\_tensor = input\_layer(features, columns)  
  
dense\_tensor == [[1, 0, 0]]  # If "name" bytes\_list is ["bob"]  
dense\_tensor == [[1, 0, 1]]  # If "name" bytes\_list is ["bob", "wanda"]  
dense\_tensor == [[2, 0, 0]]  # If "name" bytes\_list is ["bob", "bob"]

#### Args:

* **categorical\_column**: A CategoricalColumn which is created bycategorical\_column\_with\_\* or crossed\_column functions.

#### Returns:

An IndicatorColumn.

# tf.feature\_column.make\_parse\_example\_spec

* [**Contents**](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/feature_column/make_parse_example_spec#top_of_page)
* [Aliases:](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/feature_column/make_parse_example_spec#aliases)
* [Used in the guide:](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/feature_column/make_parse_example_spec#used_in_the_guide)

Creates parsing spec dictionary from input feature\_columns.

### Aliases:

* tf.compat.v2.feature\_column.make\_parse\_example\_spec
* tf.feature\_column.make\_parse\_example\_spec

tf.feature\_column.make\_parse\_example\_spec(feature\_columns)

Defined in [python/feature\_column/feature\_column\_v2.py](https://github.com/tensorflow/tensorflow/tree/r2.0/tensorflow/python/feature_column/feature_column_v2.py).

### Used in the guide:

* [Using the SavedModel format](https://www.tensorflow.org/beta/guide/saved_model)

The returned dictionary can be used as arg 'features' in [tf.io.parse\_example](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/io/parse_example).

#### Typical usage example:

# Define features and transformations  
feature\_a = categorical\_column\_with\_vocabulary\_file(...)  
feature\_b = numeric\_column(...)  
feature\_c\_bucketized = bucketized\_column(numeric\_column("feature\_c"), ...)  
feature\_a\_x\_feature\_c = crossed\_column(  
    columns=["feature\_a", feature\_c\_bucketized], ...)  
  
feature\_columns = set(  
    [feature\_b, feature\_c\_bucketized, feature\_a\_x\_feature\_c])  
features = tf.io.parse\_example(  
    serialized=serialized\_examples,  
    features=make\_parse\_example\_spec(feature\_columns))

For the above example, make\_parse\_example\_spec would return the dict:

{  
    "feature\_a": parsing\_ops.VarLenFeature(tf.string),  
    "feature\_b": parsing\_ops.FixedLenFeature([1], dtype=tf.float32),  
    "feature\_c": parsing\_ops.FixedLenFeature([1], dtype=tf.float32)  
}

#### Args:

* **feature\_columns**: An iterable containing all feature columns. All items should be instances of classes derived from FeatureColumn.

#### Returns:

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

#### Raises:

* **ValueError**: If any of the given feature\_columns is not a FeatureColumn instance.

# tf.feature\_column.numeric\_column

* [**Contents**](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/feature_column/numeric_column#top_of_page)
* [Aliases:](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/feature_column/numeric_column#aliases)
* [Used in the guide:](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/feature_column/numeric_column#used_in_the_guide)
* [Used in the tutorials:](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/feature_column/numeric_column#used_in_the_tutorials)

Represents real valued or numerical features.

### Aliases:

* tf.compat.v1.feature\_column.numeric\_column
* tf.compat.v2.feature\_column.numeric\_column
* tf.feature\_column.numeric\_column

tf.feature\_column.numeric\_column(  
    key,  
    shape=(1,),  
    default\_value=None,  
    dtype=tf.dtypes.float32,  
    normalizer\_fn=None  
)

Defined in [python/feature\_column/feature\_column\_v2.py](https://github.com/tensorflow/tensorflow/tree/r2.0/tensorflow/python/feature_column/feature_column_v2.py).

### Used in the guide:

* [Distributed training in TensorFlow](https://www.tensorflow.org/beta/guide/distribute_strategy)
* [Using the SavedModel format](https://www.tensorflow.org/beta/guide/saved_model)

### Used in the tutorials:

* [Build a linear model with Estimators](https://www.tensorflow.org/beta/tutorials/estimators/linear)
* [Classify structured data](https://www.tensorflow.org/beta/tutorials/keras/feature_columns)
* [Premade Estimators](https://www.tensorflow.org/beta/tutorials/estimators/premade_estimators)

#### Example:

price = numeric\_column('price')  
columns = [price, ...]  
features = tf.io.parse\_example(..., features=make\_parse\_example\_spec(columns))  
dense\_tensor = input\_layer(features, columns)  
  
# or  
bucketized\_price = bucketized\_column(price, boundaries=[...])  
columns = [bucketized\_price, ...]  
features = tf.io.parse\_example(..., features=make\_parse\_example\_spec(columns))  
linear\_prediction = linear\_model(features, columns)

#### Args:

* **key**: A unique string identifying the input feature. It is used as the column name and the dictionary key for feature parsing configs, feature Tensor objects, and feature columns.
* **shape**: An iterable of integers 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 tf.Example parsing if data is missing. A default value of None will cause [tf.io.parse\_example](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/io/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**: defines the type of values. Default value is [tf.float32](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf#float32). Must be a non-quantized, real integer or floating point type.
* **normalizer\_fn**: If not None, 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. lambda 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.

#### Returns:

A NumericColumn.

#### 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](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf#float32).

# tf.feature\_column.sequence\_categorical\_column\_with\_hash\_bucket

* [**Contents**](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/feature_column/sequence_categorical_column_with_hash_bucket#top_of_page)
* [Aliases:](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/feature_column/sequence_categorical_column_with_hash_bucket#aliases)

A sequence of categorical terms where ids are set by hashing.

### Aliases:

* tf.compat.v1.feature\_column.sequence\_categorical\_column\_with\_hash\_bucket
* tf.compat.v2.feature\_column.sequence\_categorical\_column\_with\_hash\_bucket
* tf.feature\_column.sequence\_categorical\_column\_with\_hash\_bucket

tf.feature\_column.sequence\_categorical\_column\_with\_hash\_bucket(  
    key,  
    hash\_bucket\_size,  
    dtype=tf.dtypes.string  
)

Defined in [python/feature\_column/sequence\_feature\_column.py](https://github.com/tensorflow/tensorflow/tree/r2.0/tensorflow/python/feature_column/sequence_feature_column.py).

Pass this to embedding\_column or indicator\_column to convert sequence categorical data into dense representation for input to sequence NN, such as RNN.

#### Example:

tokens = sequence\_categorical\_column\_with\_hash\_bucket(  
    'tokens', hash\_bucket\_size=1000)  
tokens\_embedding = embedding\_column(tokens, dimension=10)  
columns = [tokens\_embedding]  
  
features = tf.io.parse\_example(..., features=make\_parse\_example\_spec(columns))  
sequence\_feature\_layer = SequenceFeatures(columns)  
sequence\_input, sequence\_length = sequence\_feature\_layer(features)  
sequence\_length\_mask = tf.sequence\_mask(sequence\_length)  
  
rnn\_cell = tf.keras.layers.SimpleRNNCell(hidden\_size)  
rnn\_layer = tf.keras.layers.RNN(rnn\_cell)  
outputs, state = rnn\_layer(sequence\_input, mask=sequence\_length\_mask)

#### Args:

* **key**: A unique string identifying the input feature.
* **hash\_bucket\_size**: An int > 1. The number of buckets.
* **dtype**: The type of features. Only string and integer types are supported.

#### Returns:

A SequenceCategoricalColumn.

#### Raises:

* **ValueError**: hash\_bucket\_size is not greater than 1.
* **ValueError**: dtype is neither string nor integer.

# tf.feature\_column.sequence\_categorical\_column\_with\_identity

* [**Contents**](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/feature_column/sequence_categorical_column_with_identity#top_of_page)
* [Aliases:](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/feature_column/sequence_categorical_column_with_identity#aliases)

Returns a feature column that represents sequences of integers.

### Aliases:

* tf.compat.v1.feature\_column.sequence\_categorical\_column\_with\_identity
* tf.compat.v2.feature\_column.sequence\_categorical\_column\_with\_identity
* tf.feature\_column.sequence\_categorical\_column\_with\_identity

tf.feature\_column.sequence\_categorical\_column\_with\_identity(  
    key,  
    num\_buckets,  
    default\_value=None  
)

Defined in [python/feature\_column/sequence\_feature\_column.py](https://github.com/tensorflow/tensorflow/tree/r2.0/tensorflow/python/feature_column/sequence_feature_column.py).

Pass this to embedding\_column or indicator\_column to convert sequence categorical data into dense representation for input to sequence NN, such as RNN.

#### Example:

watches = sequence\_categorical\_column\_with\_identity(  
    'watches', num\_buckets=1000)  
watches\_embedding = embedding\_column(watches, dimension=10)  
columns = [watches\_embedding]  
  
features = tf.io.parse\_example(..., features=make\_parse\_example\_spec(columns))  
sequence\_feature\_layer = SequenceFeatures(columns)  
sequence\_input, sequence\_length = sequence\_feature\_layer(features)  
sequence\_length\_mask = tf.sequence\_mask(sequence\_length)  
  
rnn\_cell = tf.keras.layers.SimpleRNNCell(hidden\_size)  
rnn\_layer = tf.keras.layers.RNN(rnn\_cell)  
outputs, state = rnn\_layer(sequence\_input, mask=sequence\_length\_mask)

#### Args:

* **key**: A unique string identifying the input feature.
* **num\_buckets**: Range of inputs. Namely, inputs are expected to be in the range [0, num\_buckets).
* **default\_value**: If None, 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 out-of-range inputs.

#### Returns:

A SequenceCategoricalColumn.

#### Raises:

* **ValueError**: if num\_buckets is less than one.
* **ValueError**: if default\_value is not in range [0, num\_buckets).

# tf.feature\_column.sequence\_categorical\_column\_with\_vocabulary\_file

* [**Contents**](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/feature_column/sequence_categorical_column_with_vocabulary_file#top_of_page)
* [Aliases:](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/feature_column/sequence_categorical_column_with_vocabulary_file#aliases)

A sequence of categorical terms where ids use a vocabulary file.

### Aliases:

* tf.compat.v1.feature\_column.sequence\_categorical\_column\_with\_vocabulary\_file
* tf.compat.v2.feature\_column.sequence\_categorical\_column\_with\_vocabulary\_file
* tf.feature\_column.sequence\_categorical\_column\_with\_vocabulary\_file

tf.feature\_column.sequence\_categorical\_column\_with\_vocabulary\_file(  
    key,  
    vocabulary\_file,  
    vocabulary\_size=None,  
    num\_oov\_buckets=0,  
    default\_value=None,  
    dtype=tf.dtypes.string  
)

Defined in [python/feature\_column/sequence\_feature\_column.py](https://github.com/tensorflow/tensorflow/tree/r2.0/tensorflow/python/feature_column/sequence_feature_column.py).

Pass this to embedding\_column or indicator\_column to convert sequence categorical data into dense representation for input to sequence NN, such as RNN.

#### Example:

states = sequence\_categorical\_column\_with\_vocabulary\_file(  
    key='states', vocabulary\_file='/us/states.txt', vocabulary\_size=50,  
    num\_oov\_buckets=5)  
states\_embedding = embedding\_column(states, dimension=10)  
columns = [states\_embedding]  
  
features = tf.io.parse\_example(..., features=make\_parse\_example\_spec(columns))  
sequence\_feature\_layer = SequenceFeatures(columns)  
sequence\_input, sequence\_length = sequence\_feature\_layer(features)  
sequence\_length\_mask = tf.sequence\_mask(sequence\_length)  
  
rnn\_cell = tf.keras.layers.SimpleRNNCell(hidden\_size)  
rnn\_layer = tf.keras.layers.RNN(rnn\_cell)  
outputs, state = rnn\_layer(sequence\_input, mask=sequence\_length\_mask)

#### Args:

* **key**: A unique string identifying the input feature.
* **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. If None, it is set to the length of vocabulary\_file.
* **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\_buckets) 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.

#### Returns:

A SequenceCategoricalColumn.

#### Raises:

* **ValueError**: vocabulary\_file is missing or cannot be opened.
* **ValueError**: vocabulary\_size is missing or < 1.
* **ValueError**: num\_oov\_buckets is a negative integer.
* **ValueError**: num\_oov\_buckets and default\_value are both specified.
* **ValueError**: dtype is neither string nor integer.

# tf.feature\_column.sequence\_categorical\_column\_with\_vocabulary\_list

* [**Contents**](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/feature_column/sequence_categorical_column_with_vocabulary_list#top_of_page)
* [Aliases:](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/feature_column/sequence_categorical_column_with_vocabulary_list#aliases)

A sequence of categorical terms where ids use an in-memory list.

### Aliases:

* tf.compat.v1.feature\_column.sequence\_categorical\_column\_with\_vocabulary\_list
* tf.compat.v2.feature\_column.sequence\_categorical\_column\_with\_vocabulary\_list
* tf.feature\_column.sequence\_categorical\_column\_with\_vocabulary\_list

tf.feature\_column.sequence\_categorical\_column\_with\_vocabulary\_list(  
    key,  
    vocabulary\_list,  
    dtype=None,  
    default\_value=-1,  
    num\_oov\_buckets=0  
)

Defined in [python/feature\_column/sequence\_feature\_column.py](https://github.com/tensorflow/tensorflow/tree/r2.0/tensorflow/python/feature_column/sequence_feature_column.py).

Pass this to embedding\_column or indicator\_column to convert sequence categorical data into dense representation for input to sequence NN, such as RNN.

#### Example:

colors = sequence\_categorical\_column\_with\_vocabulary\_list(  
    key='colors', vocabulary\_list=('R', 'G', 'B', 'Y'),  
    num\_oov\_buckets=2)  
colors\_embedding = embedding\_column(colors, dimension=3)  
columns = [colors\_embedding]  
  
features = tf.io.parse\_example(..., features=make\_parse\_example\_spec(columns))  
sequence\_feature\_layer = SequenceFeatures(columns)  
sequence\_input, sequence\_length = sequence\_feature\_layer(features)  
sequence\_length\_mask = tf.sequence\_mask(sequence\_length)  
  
rnn\_cell = tf.keras.layers.SimpleRNNCell(hidden\_size)  
rnn\_layer = tf.keras.layers.RNN(rnn\_cell)  
outputs, state = rnn\_layer(sequence\_input, mask=sequence\_length\_mask)

#### Args:

* **key**: A unique string identifying the input feature.
* **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 None, it will be inferred from vocabulary\_list.
* **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.
* **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 [len(vocabulary\_list), len(vocabulary\_list)+num\_oov\_buckets) based on a hash of the input value. A positive num\_oov\_buckets can not be specified with default\_value.

#### Returns:

A SequenceCategoricalColumn.

#### Raises:

* **ValueError**: if vocabulary\_list is empty, or contains duplicate keys.
* **ValueError**: num\_oov\_buckets is a negative integer.
* **ValueError**: num\_oov\_buckets and default\_value are both specified.
* **ValueError**: if dtype is not integer or string.

# tf.feature\_column.sequence\_numeric\_column

* [**Contents**](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/feature_column/sequence_numeric_column#top_of_page)
* [Aliases:](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/feature_column/sequence_numeric_column#aliases)

Returns a feature column that represents sequences of numeric data.

### Aliases:

* tf.compat.v1.feature\_column.sequence\_numeric\_column
* tf.compat.v2.feature\_column.sequence\_numeric\_column
* tf.feature\_column.sequence\_numeric\_column

tf.feature\_column.sequence\_numeric\_column(  
    key,  
    shape=(1,),  
    default\_value=0.0,  
    dtype=tf.dtypes.float32,  
    normalizer\_fn=None  
)

Defined in [python/feature\_column/sequence\_feature\_column.py](https://github.com/tensorflow/tensorflow/tree/r2.0/tensorflow/python/feature_column/sequence_feature_column.py).

#### Example:

temperature = sequence\_numeric\_column('temperature')  
columns = [temperature]  
  
features = tf.io.parse\_example(..., features=make\_parse\_example\_spec(columns))  
sequence\_feature\_layer = SequenceFeatures(columns)  
sequence\_input, sequence\_length = sequence\_feature\_layer(features)  
sequence\_length\_mask = tf.sequence\_mask(sequence\_length)  
  
rnn\_cell = tf.keras.layers.SimpleRNNCell(hidden\_size)  
rnn\_layer = tf.keras.layers.RNN(rnn\_cell)  
outputs, state = rnn\_layer(sequence\_input, mask=sequence\_length\_mask)

#### Args:

* **key**: A unique string identifying the input features.
* **shape**: The shape of the input data per sequence id. E.g. if shape=(2,), each example must contain 2 \* sequence\_length values.
* **default\_value**: A single value compatible with dtype that is used for padding the sparse data into a dense Tensor.
* **dtype**: The type of values.
* **normalizer\_fn**: If not None, 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. lambda 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.

#### Returns:

A SequenceNumericColumn.

#### Raises:

* **TypeError**: if any dimension in shape is not an int.
* **ValueError**: if any dimension in shape is not a positive integer.
* **ValueError**: if dtype is not convertible to [tf.float32](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf#float32).

# tf.feature\_column.shared\_embeddings

* [**Contents**](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/feature_column/shared_embeddings#top_of_page)
* [Aliases:](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/feature_column/shared_embeddings#aliases)

List of dense columns that convert from sparse, categorical input.

### Aliases:

* tf.compat.v2.feature\_column.shared\_embeddings
* tf.feature\_column.shared\_embeddings

tf.feature\_column.shared\_embeddings(  
    categorical\_columns,  
    dimension,  
    combiner='mean',  
    initializer=None,  
    shared\_embedding\_collection\_name=None,  
    ckpt\_to\_load\_from=None,  
    tensor\_name\_in\_ckpt=None,  
    max\_norm=None,  
    trainable=True  
)

Defined in [python/feature\_column/feature\_column\_v2.py](https://github.com/tensorflow/tensorflow/tree/r2.0/tensorflow/python/feature_column/feature_column_v2.py).

This is similar to embedding\_column, except that it produces a list of embedding columns that share the same embedding weights.

Use this when your inputs are sparse and of the same type (e.g. watched and impression video IDs that share the same vocabulary), and you want to convert them to a dense representation (e.g., to feed to a DNN).

Inputs must be a list of categorical columns created by any of the categorical\_column\_\* function. They must all be of the same type and have the same arguments except key. E.g. they can be categorical\_column\_with\_vocabulary\_file with the same vocabulary\_file. Some or all columns could also be weighted\_categorical\_column.

Here is an example embedding of two features for a DNNClassifier model:

watched\_video\_id = categorical\_column\_with\_vocabulary\_file(  
    'watched\_video\_id', video\_vocabulary\_file, video\_vocabulary\_size)  
impression\_video\_id = categorical\_column\_with\_vocabulary\_file(  
    'impression\_video\_id', video\_vocabulary\_file, video\_vocabulary\_size)  
columns = shared\_embedding\_columns(  
    [watched\_video\_id, impression\_video\_id], dimension=10)  
  
estimator = tf.estimator.DNNClassifier(feature\_columns=columns, ...)  
  
label\_column = ...  
def input\_fn():  
  features = tf.io.parse\_example(  
      ..., features=make\_parse\_example\_spec(columns + [label\_column]))  
  labels = features.pop(label\_column.name)  
  return features, labels  
  
estimator.train(input\_fn=input\_fn, steps=100)

Here is an example using shared\_embedding\_columns with model\_fn:

def model\_fn(features, ...):  
  watched\_video\_id = categorical\_column\_with\_vocabulary\_file(  
      'watched\_video\_id', video\_vocabulary\_file, video\_vocabulary\_size)  
  impression\_video\_id = categorical\_column\_with\_vocabulary\_file(  
      'impression\_video\_id', video\_vocabulary\_file, video\_vocabulary\_size)  
  columns = shared\_embedding\_columns(  
      [watched\_video\_id, impression\_video\_id], dimension=10)  
  dense\_tensor = input\_layer(features, columns)  
  # Form DNN layers, calculate loss, and return EstimatorSpec.  
  ...

#### Args:

* **categorical\_columns**: List of categorical columns created by acategorical\_column\_with\_\* function. These columns produce the sparse IDs that are inputs to the embedding lookup. All columns must be of the same type and have the same arguments except key. E.g. they can be categorical\_column\_with\_vocabulary\_file with the same vocabulary\_file. Some or all columns could also be weighted\_categorical\_column.
* **dimension**: An integer specifying dimension of the embedding, must be > 0.
* **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. For more information, see tf.embedding\_lookup\_sparse.
* **initializer**: A variable initializer function to be used in embedding variable initialization. If not specified, defaults to [tf.compat.v1.truncated\_normal\_initializer](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/compat/v1/truncated_normal_initializer) with mean 0.0 and standard deviation 1/sqrt(dimension).
* **shared\_embedding\_collection\_name**: Optional collective name of these columns. If not given, a reasonable name will be chosen based on the names of categorical\_columns.
* **ckpt\_to\_load\_from**: String representing checkpoint name/pattern from which to restore column weights. Required if tensor\_name\_in\_ckpt is not None.
* **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 None.
* **max\_norm**: If not None, each embedding is clipped if its l2-norm is larger than this value, before combining.
* **trainable**: Whether or not the embedding is trainable. Default is True.

#### Returns:

A list of dense columns that converts from sparse input. The order of results follows the ordering of categorical\_columns.

#### Raises:

* **ValueError**: if dimension not > 0.
* **ValueError**: if any of the given categorical\_columns is of different type or has different arguments than the others.
* **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.
* **RuntimeError**: if eager execution is enabled.

# tf.feature\_column.weighted\_categorical\_column

* [**Contents**](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/feature_column/weighted_categorical_column#top_of_page)
* [Aliases:](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/feature_column/weighted_categorical_column#aliases)

Applies weight values to a CategoricalColumn.

### Aliases:

* tf.compat.v1.feature\_column.weighted\_categorical\_column
* tf.compat.v2.feature\_column.weighted\_categorical\_column
* tf.feature\_column.weighted\_categorical\_column

tf.feature\_column.weighted\_categorical\_column(  
    categorical\_column,  
    weight\_feature\_key,  
    dtype=tf.dtypes.float32  
)

Defined in [python/feature\_column/feature\_column\_v2.py](https://github.com/tensorflow/tensorflow/tree/r2.0/tensorflow/python/feature_column/feature_column_v2.py).

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).

#### Example:

Input tf.Example objects:

[  
  features {  
    feature {  
      key: "terms"  
      value {bytes\_list {value: "very" value: "model"}}  
    }  
    feature {  
      key: "frequencies"  
      value {float\_list {value: 0.3 value: 0.1}}  
    }  
  },  
  features {  
    feature {  
      key: "terms"  
      value {bytes\_list {value: "when" value: "course" value: "human"}}  
    }  
    feature {  
      key: "frequencies"  
      value {float\_list {value: 0.4 value: 0.1 value: 0.2}}  
    }  
  }  
]

categorical\_column = categorical\_column\_with\_hash\_bucket(  
    column\_name='terms', hash\_bucket\_size=1000)  
weighted\_column = weighted\_categorical\_column(  
    categorical\_column=categorical\_column, weight\_feature\_key='frequencies')  
columns = [weighted\_column, ...]  
features = tf.io.parse\_example(..., features=make\_parse\_example\_spec(columns))  
linear\_prediction, \_, \_ = linear\_model(features, columns)

This assumes the input dictionary contains a SparseTensor for key 'terms', and a SparseTensor for key 'frequencies'. These 2 tensors must have the same indices and dense shape.

#### Args:

* **categorical\_column**: A CategoricalColumn created by categorical\_column\_with\_\*functions.
* **weight\_feature\_key**: String key for weight values.
* **dtype**: Type of weights, such as [tf.float32](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf#float32). Only float and integer weights are supported.

#### Returns:

A CategoricalColumn 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.