Google Developers

Blog of our latest news, updates, and stories for developers

Introducing TensorFlow Feature Columns

Monday, November 20, 2017

Posted by the TensorFlow Team

Welcome to Part 2 of a blog series that introduces TensorFlow Datasets and Estimators. We're devoting this article to **feature columns**—a data structure describing the features that an Estimator requires for training and inference. As you'll see, feature columns are very rich, enabling you to represent a diverse range of data.

In Part 1, we used the pre-made Estimator `DNNClassifier` to train a model to predict different types of Iris flowers from four input features. That example created only numerical feature columns (of type `tf.feature_column.numeric_column)`. Although those feature columns were sufficient to model the lengths of petals and sepals, real world data sets contain all kinds of non-numerical features. For example:

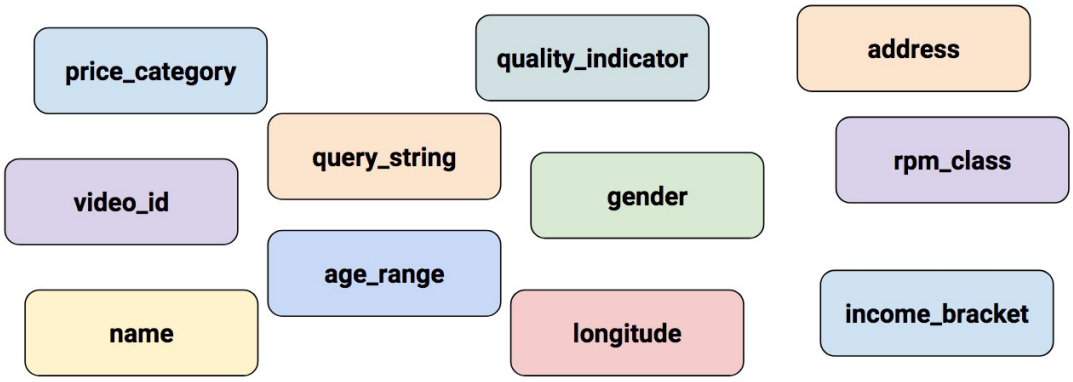

Figure 1. Non-numerical features.

How can we represent non-numerical feature types? That's exactly what this blogpost is all about.

Input to a Deep Neural Network

Let's start by asking what kind of data can we actually feed into a deep neural network? The answer is, of course, numbers (for example, `tf.float32`). After all, every neuron in a neural network performs multiplication and addition operations on weights and input data. Real-life input data, however, often contains non-numerical (categorical) data. For example, consider a `product_class` feature that can contain the following three non-numerical values:

- `kitchenware`
- `electronics`
- `sports`

ML models generally represent categorical values as simple vectors in which a 1 represents the presence of a value and a 0 represents the absence of a value. For example, when `product_class` is set to

`sports`, an ML model would usually represent `product_class` as [0, 0, 1], meaning:

- 0: `kitchenware` is absent
- 0: `electronics` is absent
- 1: `sports`: is present

So, although raw data can be numerical or categorical, an ML model represents all features as either a number or a vector of numbers.

Introducing Feature Columns

As Figure 2 suggests, you specify the input to a model through the `feature_columns` argument of an Estimator (`DNNClassifier` for Iris). Feature Columns bridge input data (as returned by `input_fn`) with your model.

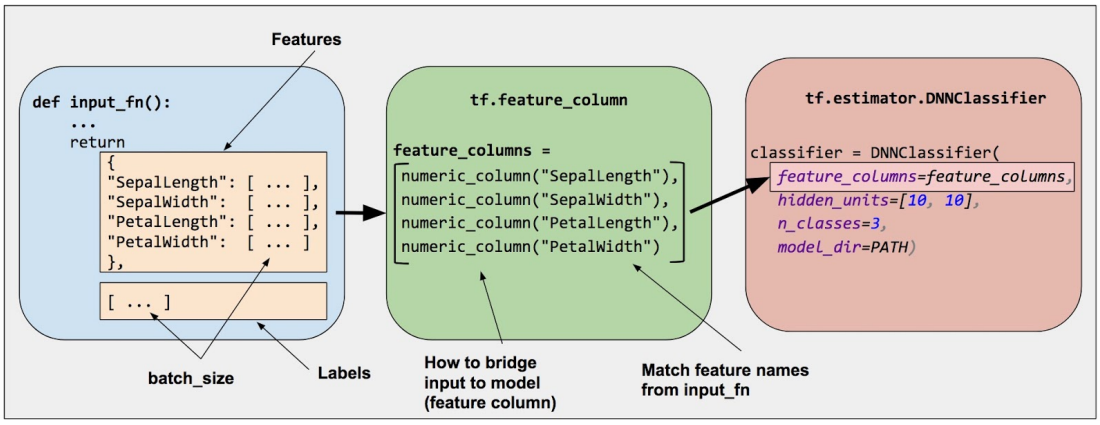

Figure 2. Feature columns bridge raw data with the data your model needs.

To represent features as a feature column, call functions of the `tf.feature_column` package. This blogpost explains nine of the functions in this package. As Figure 3 shows, all nine functions return

either a Categorical-Column or a Dense-Column object, except `bucketized_column` which inherits from both classes:

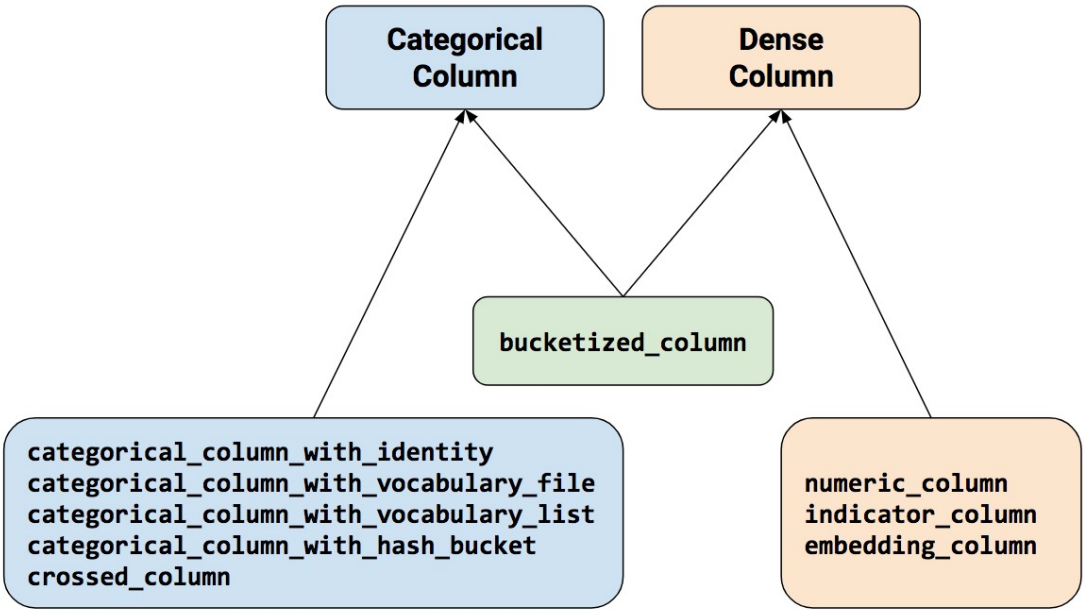

Figure 3. Feature column methods fall into two main categories and one hybrid category.

Let's look at these functions in more detail.

Numeric Column

The Iris classifier called `tf.numeric_column()` for all input features: SepalLength, SepalWidth, PetalLength, PetalWidth. Although `tf.numeric_column()` provides optional arguments, calling the function without any arguments is a perfectly easy way to specify a numerical value with the default data type (`tf.float32`) as input to your model. For example:

```
# Defaults to a tf.float32 scalar.
numeric_feature_column = tf.feature_column.numeric_column(key="SepalLer
```

Use the `dtype` argument to specify a non-default numerical data type. For example:

```
# Represent a tf.float64 scalar.
numeric_feature_column = tf.feature_column.numeric_column(key="SepalLer
                                                          dtype=tf.floa
```

By default, a numeric column creates a single value (scalar). Use the `shape` argument to specify another shape. For example:

```
# Represent a 10-element vector in which each cell contains a tf.float3
vector_feature_column = tf.feature_column.numeric_column(key="Bowling",
                                                         shape=10)

# Represent a 10x5 matrix in which each cell contains a tf.float32.
matrix_feature_column = tf.feature_column.numeric_column(key="MyMatrix"
                                                         shape=[10,5])
```

Bucketized Column

Often, you don't want to feed a number directly into the model, but instead split its value into different categories based on numerical ranges. To do so, create a bucketized column. For example, consider raw data that represents the year a house was built. Instead of representing that year as a scalar numeric column, we could split year into the following four buckets:

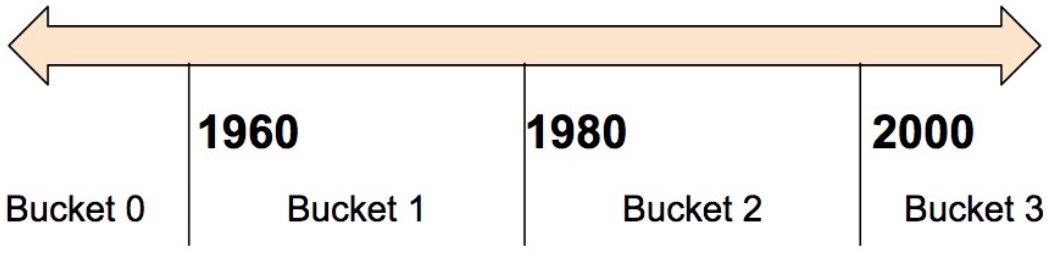

Figure 4. Dividing year data into four buckets.

The model will represent the buckets as follows:

Date Range	Represented as...
< 1960	[1, 0, 0, 0]
>= 1960 but < 1980	[0, 1, 0, 0]
>= 1980 but < 2000	[0, 0, 1, 0]
> 2000	[0, 0, 0, 1]

Why would you want to split a number—a perfectly valid input to our model—into a categorical value like this? Well, notice that the categorization splits a single input number into a four-element vector. Therefore, the model now can learn *four individual weights* rather than just one. Four weights creates a richer model than one. More importantly, bucketizing enables the model to clearly distinguish between different year categories since only one of the elements is set (1) and the other three elements are cleared (0). When we just use a single number (a year) as input, the model can't distinguish categories. So, bucketing provides the model with additional important information that it can use to learn.

The following code demonstrates how to create a bucketized feature:

```
# A numeric column for the raw input.
numeric_feature_column = tf.feature_column.numeric_column("Year")

# Bucketize the numeric column on the years 1960, 1980, and 2000
bucketized_feature_column = tf.feature_column.bucketized_column(
    source_column = numeric_feature_column,
    boundaries = [1960, 1980, 2000])
```

Note the following:

- Before creating the bucketized column, we first created a

numeric column to represent the raw year.

- We passed the numeric column as the first argument to `tf.feature_column.bucketized_column()`.
- Specifying a *three*-element `boundaries` vector creates a *four*-element bucketized vector.

Categorical identity column

Categorical identity columns are a special case of bucketized columns. In traditional bucketized columns, each bucket represents a *range* of values (for example, from 1960 to 1979). In a categorical identity column, each bucket represents a *single*, unique integer. For example, let's say you want to represent the integer range [0, 4). (That is, you want to represent the integers 0, 1, 2, or 3.) In this case, the categorical identity mapping looks like this:

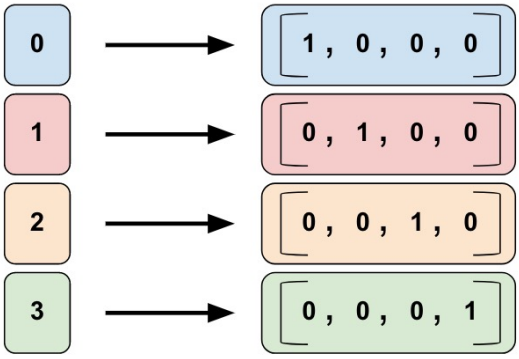

Figure 5. A categorical identity column mapping. Note that this is a one-hot encoding, not a binary numerical encoding.

So, why would you want to represent values as categorical identity columns? As with bucketized columns, a model can learn a separate weight for each class in a categorical identity column. For example,

instead of using a string to represent the `product_class`, let's represent each class with a unique integer value. That is:

- `0="kitchenware"`
- `1="electronics"`
- `2="sport"`

Call `tf.feature_column.categorical_column_with_identity()` to implement a categorical identity column. For example:

```
# Create a categorical output for input "feature_name_from_input_fn",
# which must be of integer type. Value is expected to be >= 0 and < num
identity_feature_column = tf.feature_column.categorical_column_with_ide
    key='feature_name_from_input_fn',
    num_buckets=4) # Values [0, 4)

# The 'feature_name_from_input_fn' above needs to match an integer key
# returned from input_fn (see below). So for this case, 'Integer_1' or
# 'Integer_2' would be valid strings instead of 'feature_name_from_inpu
# For more information, please check out Part 1 of this blog series.
def input_fn():
    ...<code>...
    return ({ 'Integer_1':[values], ..<etc>.., 'Integer_2':[values] },
            [Label_values])
```

Categorical vocabulary column

We cannot input strings directly to a model. Instead, we must first map strings to numeric or categorical values. Categorical vocabulary columns provide a good way to represent strings as a one-hot vector. For example:

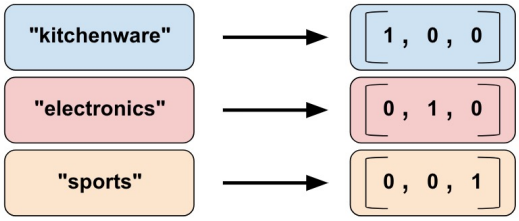

Figure 6. Mapping string values to vocabulary columns.

As you can see, categorical vocabulary columns are kind of an enum version of categorical identity columns. TensorFlow provides two different functions to create categorical vocabulary columns:

- `tf.feature_column.categorical_column_with_vocabulary_list()`
- `tf.feature_column.categorical_column_with_vocabulary_file()`

The `tf.feature_column.categorical_column_with_vocabulary_list()` function maps each string to an integer based on an explicit vocabulary list. For example:

```
# Given input "feature_name_from_input_fn" which is a string,
# create a categorical feature to our model by mapping the input to one
# the elements in the vocabulary list.
vocabulary_feature_column =
    tf.feature_column.categorical_column_with_vocabulary_list(
        key="feature_name_from_input_fn",
        vocabulary_list=["kitchenware", "electronics", "sports"])
```

The preceding function has a significant drawback; namely, there's way too much typing when the vocabulary list is long. For these cases, call `tf.feature_column.categorical_column_with_vocabulary_f`

`ile()` instead, which lets you place the vocabulary words in a separate file. For example:

```
# Given input "feature_name_from_input_fn" which is a string,
# create a categorical feature to our model by mapping the input to one
# the elements in the vocabulary file
vocabulary_feature_column =
    tf.feature_column.categorical_column_with_vocabulary_file(
        key="feature_name_from_input_fn",
        vocabulary_file="product_class.txt",
        vocabulary_size=3)

# product_class.txt should have one line for vocabulary element, in our
kitchenware
electronics
sports
```

Using hash buckets to limit categories

So far, we've worked with a naively small number of categories. For example, our `product_class` example has only 3 categories. Often though, the number of categories can be so big that it's not possible to have individual categories for each vocabulary word or integer because that would consume too much memory. For these cases, we can instead turn the question around and ask, "How many categories am I willing to have for my input?" In fact, the `tf.feature_column.categorical_column_with_hash_buckets()` function enables you to specify the number of categories. For example, the following code shows how this function calculates a hash value of the input, then puts it into one of the `hash_bucket_size` categories using the modulo operator:

```
# Create categorical output for input "feature_name_from_input_fn".
# Category becomes: hash_value("feature_name_from_input_fn") % hash_buc
hashed_feature_column =
```

```
tf.feature_column.categorical_column_with_hash_bucket(
    key = "feature_name_from_input_fn",
    hash_buckets_size = 100) # The number of categories
```

At this point, you might rightfully think: "This is crazy!" After all, we are forcing the different input values to a smaller set of categories. This means that two, probably completely unrelated inputs, will be mapped to the same category, and consequently mean the same thing to the neural network. Figure 7 illustrates this dilemma, showing that `kitchenware` and `sports` both get assigned to category (hash bucket) 12:

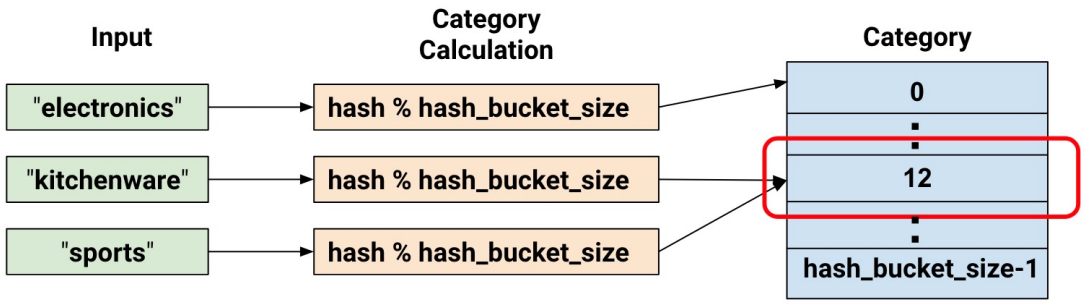

Figure 7. Representing data in hash buckets.

As with many counterintuitive phenomena in machine learning, it turns out that hashing often works well in practice. That's because hash categories provide the model with some separation. The model can use additional features to further separate `kitchenware` from `sports`.

Feature crosses

The last categorical column we'll cover allows us to combine multiple input features into a single one. Combining features, better known as feature crosses, enables the model to learn separate weights specifically for whatever that feature combination means.

More concretely, suppose we want our model to calculate real estate prices in Atlanta, GA. Real-estate prices within this city vary greatly depending on location. Representing latitude and longitude as separate features isn't very useful in identifying real-estate location dependencies; however, crossing latitude and longitude into a single feature can pinpoint locations. Suppose we represent Atlanta as a grid of 100x100 rectangular sections, identifying each of the 10,000 sections by a cross of its latitude and longitude. This cross enables the model to pick up on pricing conditions related to each individual section, which is a much stronger signal than latitude and longitude alone.

Figure 8 shows our plan, with the latitude & longitude values for the corners of the city:

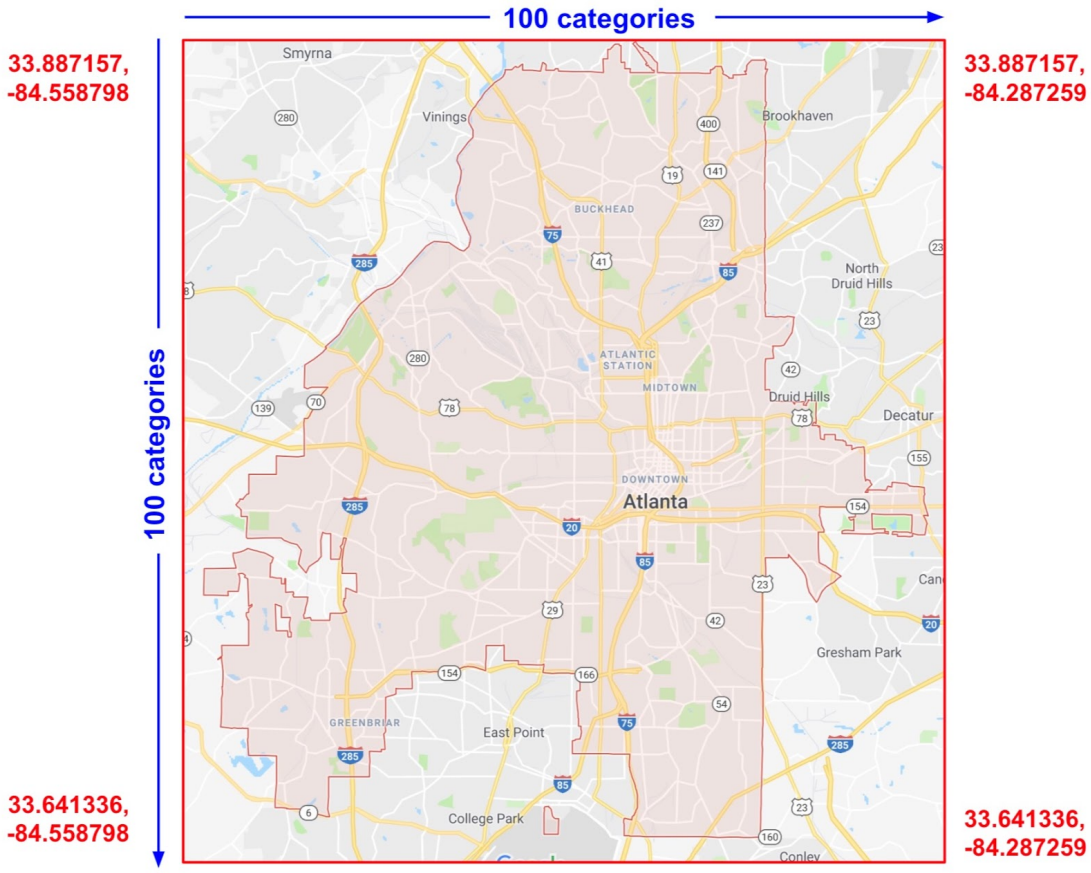

Figure 8. Map of Atlanta. Imagine this map divided into 10,000 sections of equal size.

For the solution, we used a combination of some feature columns we've looked at before, as well as the `tf.feature_columns.crossed_column()` function.

```
# In our input_fn, we convert input longitude and latitude to integer v
# in the range [0, 100)
def input_fn():
    # Using Datasets, read the input values for longitude and latitude
    latitude = ...   # A tf.float32 value
    longitude = ...  # A tf.float32 value

    # In our example we just return our lat_int, long_int features.
    # The dictionary of a complete program would probably have more key
    return { "latitude": latitude, "longitude": longitude, ...}, labels
```

```
# As can be see from the map, we want to split the latitude range
# [33.641336, 33.887157] into 100 buckets. To do this we use np.linspac
# to get a list of 99 numbers between min and max of this range.
# Using this list we can bucketize latitude into 100 buckets.
latitude_buckets = list(np.linspace(33.641336, 33.887157, 99))
latitude_fc = tf.feature_column.bucketized_column(
    tf.feature_column.numeric_column('latitude'),
    latitude_buckets)

# Do the same bucketization for longitude as done for latitude.
longitude_buckets = list(np.linspace(-84.558798, -84.287259, 99))
longitude_fc = tf.feature_column.bucketized_column(
    tf.feature_column.numeric_column('longitude'), longitude_buckets)

# Create a feature cross of fc_longitude x fc_latitude.
fc_san_francisco_boxed = tf.feature_column.crossed_column(
    keys=[latitude_fc, longitude_fc],
    hash_bucket_size=1000) # No precise rule, maybe 1000 buckets will b
```

You may create a feature cross from either of the following:

- Feature names; that is, names from the `dict` returned from `input_fn`.
- Any Categorical Column (see Figure 3), except `categorical_column_with_hash_bucket`.

When feature columns `latitude_fc` and `longitude_fc` are crossed, TensorFlow will create 10,000 combinations of (`latitude_fc`, `longitude_fc`) organized as follows:

```
(0,0),(0,1)...  (0,99)
(1,0),(1,1)...  (1,99)
…, …,           ...
(99,0),(99,1)...(99, 99)
```

The function `tf.feature_column.crossed_column` performs a hash calculation on these combinations and then slots the result into a

category by performing a modulo operation with `hash_bucket_size`. As discussed before, performing the hash and modulo function will probably result in category collisions; that is, multiple (latitude, longitude) feature crosses will end up in the same hash bucket. In practice though, performing feature crosses still provides significant value to the learning capability of your models.

Somewhat counterintuitively, when creating feature crosses, you typically still should include the original (uncrossed) features in your model. For example, provide not only the `(latitude, longitude)` feature cross but also `latitude` and `longitude` as separate features. The separate `latitude` and `longitude` features help the model separate the contents of hash buckets containing different feature crosses.

See this link for a full code example for this. Also, the reference section at the end of this post for lots more examples of feature crossing.

Indicator and embedding columns

Indicator columns and embedding columns never work on features directly, but instead take categorical columns as input.

When using an indicator column, we're telling TensorFlow to do exactly what we've seen in our categorical `product_class` example. That is, an **indicator column** treats each category as an element in a one-hot vector, where the matching category has value 1 and the rest have 0s:

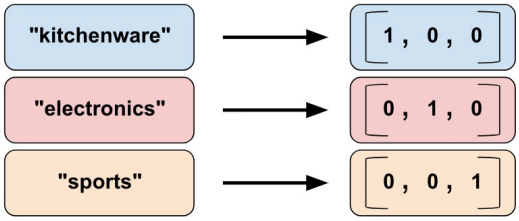

Figure 9. Representing data in indicator columns.

Here's how you create an indicator column:

```
categorical_column = ... # Create any type of categorical column, see F

# Represent the categorical column as an indicator column.
# This means creating a one-hot vector with one element for each categc
indicator_column = tf.feature_column.indicator_column(categorical_colum
```

Now, suppose instead of having just three possible classes, we have a million. Or maybe a billion. For a number of reasons (too technical to cover here), as the number of categories grow large, it becomes infeasible to train a neural network using indicator columns.

We can use an embedding column to overcome this limitation. Instead of representing the data as a one-hot vector of many dimensions, an **embedding column** represents that data as a lower-dimensional, ordinary vector in which each cell can contain any number, not just 0 or 1. By permitting a richer palette of numbers for every cell, an embedding column contains far fewer cells than an indicator column.

Let's look at an example comparing indicator and embedding columns. Suppose our input examples consists of different words from a limited palette of only 81 words. Further suppose that the data set provides the following input words in 4 separate examples:

- "dog"
- "spoon"
- "scissors"
- "guitar"

In that case, Figure 10 illustrates the processing path for embedding columns or Indicator columns.

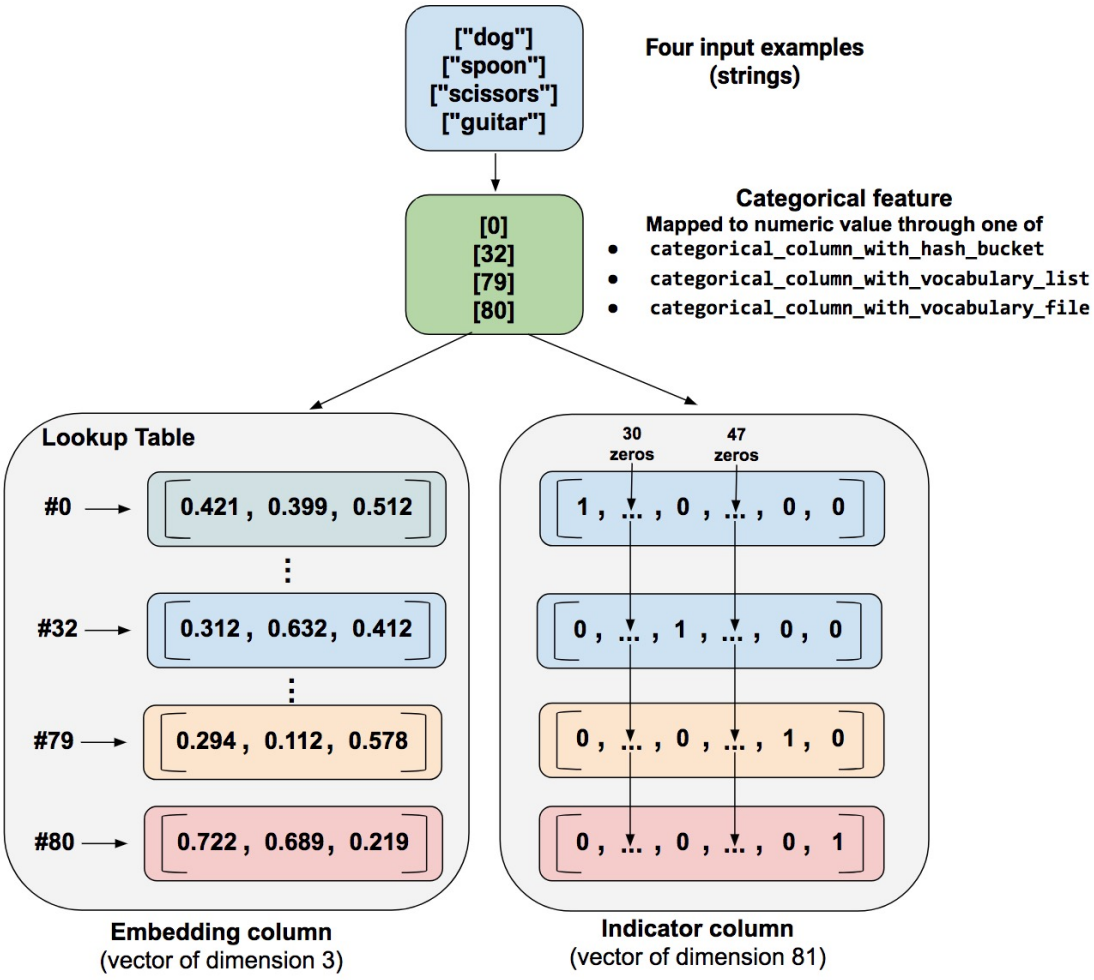

Figure 10. An embedding column stores categorical data in a lower-dimensional vector than an indicator column. (We just placed random numbers into the embedding vectors; training determines the actual numbers.)

When an example is processed, one of the `categorical_column_with...` functions maps the example string to a numerical categorical value. For example, a function maps `"spoon"` to `[32]`. (The `32` comes from our imagination—the actual values depend on the mapping function.) You may then represent these numerical categorical values in either of the following two ways:

- As an indicator column. A function converts each numeric categorical value into an 81-element vector (because our palette consists of 81 words), placing a 1 in the index of the categorical value (0, 32, 79, 80) and a 0 in all the other positions.
- As an embedding column. A function uses the numerical categorical values (0, 32, 79, 80) as indices to a lookup table. Each slot in that lookup table contains a 3-element vector.

How do the values in the embeddings vectors magically get assigned? Actually, the assignments happen during training. That is, the model learns the best way to map your input numeric categorical values to the embeddings vector value in order to solve your problem. Embedding columns increase your model's capabilities, since an embeddings vector learns new relationships between categories from the training data.

Why is the embedding vector size 3 in our example? Well, the following "formula" provides a general rule of thumb about the number of embedding dimensions:

```
embedding_dimensions =  number_of_categories**0.25
```

That is, the embedding vector dimension should be the 4th root of the

number of categories. Since our vocabulary size in this example is 81, the recommended number of dimensions is 3:

```
3 =  81**0.25
```

Note that this is just a general guideline; you can set the number of embedding dimensions as you please.

Call `tf.feature_column.embedding_column` to create an embedding_column. The dimension of the embedding vector depends on the problem at hand as described above, but common values go as low as 3 all the way to 300 or even beyond:

```
categorical_column = ... # Create any categorical column shown in Figur

# Represent the categorical column as an embedding column.
# This means creating a one-hot vector with one element for each categ
embedding_column = tf.feature_column.embedding_column(
    categorical_column=categorical_column,
    dimension=dimension_of_embedding_vector)
```

Embeddings is a big topic within machine learning. This information was just to get you started using them as feature columns. Please see the end of this post for more information.

Passing feature columns to Estimators

Still there? I hope so, because we only have a tiny bit left before you've graduated from the basics of feature columns.

As we saw in Figure 1, feature columns map your input data (described by the feature dictionary returned from `input_fn`) to values fed to your model. You specify feature columns as a list to a `feature_columns` argument of an estimator. Note that the `feature_columns` argument(s)

vary depending on the Estimator:

- `LinearClassifier` and `LinearRegressor`:
  - Accept all types of feature column.
- DNNClassifier and DNNRegressor:
  - Only accept dense columns, see Figure 3. Other column types must be wrapped in either an `indicator_column` or `embedding_column` as described earlier.
- `DNNLinearCombinedClassifier` and `DNNLinearCombinedRegressor`:
  - The `linear_feature_columns` argument can accept any column type, like the LinearClassifier and LinearRegressor above.
  - The `dnn_feature_columns` argument however is limited to dense columns, like `DNNClassifier` and `DNNRegressor` above.

The reason for the above rules are beyond the scope of this introductory post, but we will make sure to cover it in a future blogpost.

Summary

Use feature columns to map your input data to the representations you feed your model. We only used `numeric_column` in Part 1 of this series , but working with the other functions described in this post, you can easily

create other feature columns.

For more details on feature columns, be sure to check out:

- Josh Gordon's video on Feature Engineering
- And from the same author, a Jupyter notebook
- The TensorFlow - Wide & Deep Tutorial
- Examples of DNNs and linear models that use feature columns

If you want to learn more about embeddings:

- Deep Learning, NLP, and representations (Colah's blog)
- And checkout the TensorFlow Embedding Projector

Labels: Datasets , Estimators , TensorFlow

4 comments :

Bob Smith November 20, 2017 at 8:03 PM

Looks like the example Jupyter notebook works in Colaboratory with TensorFlow 1.4; no installation or git checkout required -- https://colab.research.google.com/notebook#fileId=1QhSnbh-WJVGZjQJF8u974msOL_vAgMeS

The external images are missing, but all the code seems to work.

Reply

windmaple November 21, 2017 at 1:07 AM

Great article!

Reply

NYAMUKKURUS November 26, 2017 at 7:40 AM

owh yeah, thanks feature columns

download game android gratis

Reply

Antoine Merval November 28, 2017 at 2:10 PM

Interesting post. I find the feature crosses (which is equivalent to what is usually called factor interaction in more classical classification methods such as logistic regression) and hash buckets (akin to a 'random-ish grouping') particularly interesting. Computing interaction terms is not new but it was somewhat limited before the advent of deep learning. Now we can afford to create a variable with, say, 10 thousands levels as in this example!

However I am surprised by the comments on the bucketised columns -that using n weights instead of one creates a richer model. My first remark would be that in most cases you have to decide on the buckets definition in an arbitrary manner (or, at best, using knowledge and intuition about the case). But imagine that in your example there was a 'real' underlying grouping of years say, over the period 1970-1990, that affects the target. Then your configuration would struggle to capture it!

What's more, this method increases the number of input since you're creating indicators per level of the bucketised variable. But because neural nets are a highly non-linear model -able to capture non linear relations- it seems nothing prevents us from simply defining one 'numerical' variable (with a finite number of values). Yes, there would be only one weight associated to this variable, but one weight per unit! This is thanks to the large number of units that the non linear relation will be captured. And note that this apply to categorical variables in general.

Last, bucketising numeric variables can have an adverse effect as well if the true, underlying effect of the variable on the target is actually continuous (say in your example that prices do go up every year). By bucketising the variable you will end up averaging the effects over the year that are grouped together (like transforming a linear function to a step one) and lose some accuracy. And again, if the true relation is non-linear, the neural model is supposed to be able to capture it anyway.

At the end of the day, the answer is probably that there is no true answer. The only option is to try several configurations
with variables bucketised or not and keep the best model!

Reply

Enter your comment...

Comment as: 小草 (Google) **Sign out**

Publish **Preview** Notify me