Building Input Functions with tf.contrib.learn

This tutorial introduces you to creating input functions in tf.contrib.learn. You'll get an overview of how to construct an `input_fn` to preprocess and feed data into your models. Then, you'll implement an `input_fn` that feeds training, evaluation, and prediction data into a neural network regressor for predicting median house values.

Custom Input Pipelines with input_fn

When training a neural network using tf.contrib.learn, it's possible to pass your feature and target data directly into your `fit`, `evaluate`, or `predict` operations. Here's an example taken from the tf.contrib.learn quickstart tutorial (https://www.tensorflow.org/get_started/tflearn) :

```
training_set = tf.contrib.learn.datasets.base.load_csv_with_header(
    filename=IRIS_TRAINING, target_dtype=np.int, features_dtype=np.float32)
test_set = tf.contrib.learn.datasets.base.load_csv_with_header(
    filename=IRIS_TEST, target_dtype=np.int, features_dtype=np.float32)
...

classifier.fit(x=training_set.data,
               y=training_set.target,
               steps=2000)
```

This approach works well when little to no manipulation of source data is required. But in cases where more feature engineering is needed, `tf.contrib.learn` supports using a custom input function (`input_fn`) to encapsulate the logic for preprocessing and piping data into your models.

Anatomy of an input_fn

The following code illustrates the basic skeleton for an input function:

```
def my_input_fn():

    # Preprocess your data here...

    # ...then return 1) a mapping of feature columns to Tensors with
```

```
    # the corresponding feature data, and 2) a Tensor containing labels
    return feature_cols, labels
```

The body of the input function contains the specific logic for preprocessing your input data, such as scrubbing out bad examples or feature scaling (https://en.wikipedia.org/wiki/Feature_scaling).

Input functions must return the following two values containing the final feature and label data to be fed into your model (as shown in the above code skeleton):

`feature_cols`

A dict containing key/value pairs that map feature column names to `Tensor`s (or `SparseTensor`s) containing the corresponding feature data.

`labels`

A `Tensor` containing your label (target) values: the values your model aims to predict.

Converting Feature Data to Tensors

If your feature/label data is stored in *pandas* (http://pandas.pydata.org/) dataframes or numpy (http://www.numpy.org/) arrays, you'll need to convert it to `Tensor`s before returning it from your `input_fn`.

For continuous data, you can create and populate a `Tensor` using `tf.constant`:

```
feature_column_data = [1, 2.4, 0, 9.9, 3, 120]
feature_tensor = tf.constant(feature_column_data)
```

For sparse, categorical data (https://en.wikipedia.org/wiki/Sparse_matrix) (data where the majority of values are 0), you'll instead want to populate a `SparseTensor`, which is instantiated with three arguments:

`dense_shape`

The shape of the tensor. Takes a list indicating the number of elements in each dimension. For example, `dense_shape=[3,6]` specifies a two-dimensional 3x6 tensor, `dense_shape=[2,3,4]` specifies a three-dimensional 2x3x4 tensor, and `dense_shape=[9]` specifies a one-dimensional tensor with 9 elements.

`indices`

The indices of the elements in your tensor that contain nonzero values. Takes a list of terms, where each term is itself a list containing the index of a nonzero element. (Elements are zero-indexed—i.e., [0,0] is the index value for the element in the first column of the first row in a two-dimensional tensor.) For example, `indices=[[1,3], [2,4]]` specifies that the elements with indexes of [1,3] and [2,4] have nonzero values.

`values`

A one-dimensional tensor of values. Term `i` in `values` corresponds to term `i` in `indices` and specifies its value. For example, given `indices=[[1,3], [2,4]]`, the parameter `values=[18, 3.6]` specifies that element [1,3] of the tensor has a value of 18, and element [2,4] of the tensor has a value of 3.6.

The following code defines a two-dimensional `SparseTensor` with 3 rows and 5 columns. The element with index [0,1] has a value of 6, and the element with index [2,4] has a value of 0.5 (all other values are 0):

```
sparse_tensor = tf.SparseTensor(indices=[[0,1], [2,4]],
                                values=[6, 0.5],
                                dense_shape=[3, 5])
```

This corresponds to the following dense tensor:

```
[[0, 6, 0, 0, 0]
 [0, 0, 0, 0, 0]
 [0, 0, 0, 0, 0.5]]
```

For more on `SparseTensor`, see the `tf.SparseTensor` (https://www.tensorflow.org/api_docs/python/tf/SparseTensor).

Passing input_fn Data to Your Model

To feed data to your model for training, you simply pass the input function you've created to your `fit` operation as the value of the `input_fn` parameter, e.g.:

```
classifier.fit(input_fn=my_input_fn, steps=2000)
```

Note that the `input_fn` is responsible for supplying both feature and label data to the model, and replaces both the `x` and `y` parameters in `fit`. If you supply an `input_fn` value to `fit` that is not `None` in conjunction with either an `x` or `y` parameter that is not `None`, it will result in a `ValueError`.

Also note that the `input_fn` parameter must receive a function object (i.e., `input_fn=my_input_fn`), not the return value of a function call (`input_fn=my_input_fn()`). This means that if you try to pass parameters to the input function in your `fit` call, as in the following code, it will result in a `TypeError`:

```
classifier.fit(input_fn=my_input_fn(training_set), steps=2000)
```

However, if you'd like to be able to parameterize your input function, there are other methods for doing so. You can employ a wrapper function that takes no arguments as your `input_fn` and use it to invoke your input function with the desired parameters. For example:

```
def my_input_function_training_set():
  return my_input_function(training_set)

classifier.fit(input_fn=my_input_fn_training_set, steps=2000)
```

Alternatively, you can use Python's `functools.partial` (https://docs.python.org/2/library/functools.html#functools.partial) function to construct a new function object with all parameter values fixed:

```
classifier.fit(input_fn=functools.partial(my_input_function,
                                          data_set=training_set), steps=2000)
```

A third option is to wrap your input_fn invocation in a `lambda` (https://docs.python.org/3/tutorial/controlflow.html#lambda-expressions) and pass it to the `input_fn` parameter:

```
classifier.fit(input_fn=lambda: my_input_fn(training_set), steps=2000)
```

One big advantage of architecting your input pipeline as shown above—to accept a parameter for data set—is that you can pass the same `input_fn` to `evaluate` and `predict` operations by just changing the data set argument, e.g.:

```
classifier.evaluate(input_fn=lambda: my_input_fn(test_set), steps=2000)
```

This approach enhances code maintainability: no need to capture `x` and `y` values in separate variables (e.g., `x_train`, `x_test`, `y_train`, `y_test`) for each type of operation.

A Neural Network Model for Boston House Values

In the remainder of this tutorial, you'll write an input function for preprocessing a subset of Boston housing data pulled from the UCI Housing Data Set (https://archive.ics.uci.edu/ml/datasets/Housing) and use it to feed data to a neural network regressor for predicting median house values.

The Boston CSV data sets (#setup) you'll use to train your neural network contain the following feature data (https://archive.ics.uci.edu/ml/machine-learning-databases/housing/housing.names) for Boston suburbs:

Feature	Description
CRIM	Crime rate per capita
ZN	Fraction of residential land zoned to permit 25,000+ sq ft lots
INDUS	Fraction of land that is non-retail business
NOX	Concentration of nitric oxides in parts per 10 million
RM	Average Rooms per dwelling
AGE	Fraction of owner-occupied residences built before 1940
DIS	Distance to Boston-area employment centers
TAX	Property tax rate per $10,000
PTRATIO	Student-teacher ratio

And the label your model will predict is MEDV, the median value of owner-occupied residences in thousands of dollars.

Setup

Download the following data sets: boston_train.csv (http://download.tensorflow.org/data/boston_train.csv), boston_test.csv (http://download.tensorflow.org/data/boston_test.csv), and boston_predict.csv (http://download.tensorflow.org/data/boston_predict.csv).

The following sections provide a step-by-step walkthrough of how to create an input function, feed these data sets into a neural network regressor, train and evaluate the model, and make house value predictions. The full, final code is available here

(https://www.github.com/tensorflow/tensorflow/blob/r1.2/tensorflow/examples/tutorials/input_fn/boston.py)

.

Importing the Housing Data

To start, set up your imports (including `pandas` and `tensorflow`) and set logging verbosity (https://www.tensorflow.org/get_started/monitors#enabling_logging_with_tensorflow) to `INFO` for more detailed log output:

```
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

import itertools

import pandas as pd
import tensorflow as tf

tf.logging.set_verbosity(tf.logging.INFO)
```

Define the column names for the data set in `COLUMNS`. To distinguish features from the label, also define `FEATURES` and `LABEL`. Then read the three CSVs (`tf.train` (https://www.tensorflow.org/api_docs/python/tf/train), `tf.test` (https://www.tensorflow.org/api_docs/python/tf/test), and predict (http://download.tensorflow.org/data/boston_predict.csv)) into *pandas* `DataFrame`s:

```
COLUMNS = ["crim", "zn", "indus", "nox", "rm", "age",
           "dis", "tax", "ptratio", "medv"]
FEATURES = ["crim", "zn", "indus", "nox", "rm",
            "age", "dis", "tax", "ptratio"]
LABEL = "medv"

training_set = pd.read_csv("boston_train.csv", skipinitialspace=True,
                           skiprows=1, names=COLUMNS)
test_set = pd.read_csv("boston_test.csv", skipinitialspace=True,
                       skiprows=1, names=COLUMNS)
prediction_set = pd.read_csv("boston_predict.csv", skipinitialspace=True,
                             skiprows=1, names=COLUMNS)
```

Defining FeatureColumns and Creating the Regressor

Next, create a list of `FeatureColumn`s for the input data, which formally specify the set of features to use for training. Because all features in the housing data set contain continuous values, you can create their `FeatureColumn`s using the `tf.contrib.layers.real_valued_column()` function:

```
feature_cols = [tf.contrib.layers.real_valued_column(k)
                  for k in FEATURES]
```

NOTE: For a more in-depth overview of feature columns, see this introduction (https://www.tensorflow.org/tutorials/linear#feature_columns_and_transformations), and for an example that illustrates how to define `FeatureColumns` for categorical data, see the Linear Model Tutorial (https://www.tensorflow.org/tutorials/wide).

Now, instantiate a `DNNRegressor` for the neural network regression model. You'll need to provide two arguments here: `hidden_units`, a hyperparameter specifying the number of nodes in each hidden layer (here, two hidden layers with 10 nodes each), and `feature_columns`, containing the list of `FeatureColumns` you just defined:

```
regressor = tf.contrib.learn.DNNRegressor(feature_columns=feature_cols,
                                          hidden_units=[10, 10],
                                          model_dir="/tmp/boston_model")
```

Building the input_fn

To pass input data into the `regressor`, create an input function, which will accept a *pandas* `Dataframe` and return feature column and label values as `Tensor`s:

```
def input_fn(data_set):
  feature_cols = {k: tf.constant(data_set[k].values)
                  for k in FEATURES}
  labels = tf.constant(data_set[LABEL].values)
  return feature_cols, labels
```

Note that the input data is passed into `input_fn` in the `data_set` argument, which means the function can process any of the `DataFrame`s you've imported: `training_set`, `test_set`, and `prediction_set`.

Training the Regressor

To train the neural network regressor, run `fit` with the `training_set` passed to the `input_fn` as follows:

```
regressor.fit(input_fn=lambda: input_fn(training_set), steps=5000)
```

You should see log output similar to the following, which reports training loss for every 100 steps:

```
INFO:tensorflow:Step 1: loss = 483.179
INFO:tensorflow:Step 101: loss = 81.2072
INFO:tensorflow:Step 201: loss = 72.4354
...
INFO:tensorflow:Step 1801: loss = 33.4454
INFO:tensorflow:Step 1901: loss = 32.3397
INFO:tensorflow:Step 2001: loss = 32.0053
INFO:tensorflow:Step 4801: loss = 27.2791
INFO:tensorflow:Step 4901: loss = 27.2251
INFO:tensorflow:Saving checkpoints for 5000 into /tmp/boston_model/model.ckpt
INFO:tensorflow:Loss for final step: 27.1674.
```

Evaluating the Model

Next, see how the trained model performs against the test data set. Run `evaluate`, and this time pass the `test_set` to the `input_fn`:

```
ev = regressor.evaluate(input_fn=lambda: input_fn(test_set), steps=1)
```

Retrieve the loss from the `ev` results and print it to output:

```
loss_score = ev["loss"]
print("Loss: {0:f}".format(loss_score))
```

You should see results similar to the following:

```
INFO:tensorflow:Eval steps [0,1) for training step 5000.
INFO:tensorflow:Saving evaluation summary for 5000 step: loss = 11.9221
Loss: 11.922098
```

Making Predictions

Finally, you can use the model to predict median house values for the `prediction_set`, which contains feature data but no labels for six examples:

```
y = regressor.predict(input_fn=lambda: input_fn(prediction_set))
# .predict() returns an iterator; convert to a list and print predictions
```

```
predictions = list(itertools.islice(y, 6))
print ("Predictions: {}".format(str(predictions)))
```

Your results should contain six house-value predictions in thousands of dollars, e.g:

```
Predictions: [ 33.30348587  17.04452896  22.56370163  34.74345398  14.5595397
  19.58005714]
```

Additional Resources

This tutorial focused on creating an `input_fn` for a neural network regressor. To learn more about using `input_fn`s for other types of models, check out the following resources:

- Large-scale Linear Models with TensorFlow (https://www.tensorflow.org/tutorials/linear): This introduction to linear models in TensorFlow provides a high-level overview of feature columns and techniques for transforming input data.
- TensorFlow Linear Model Tutorial (https://www.tensorflow.org/tutorials/wide): This tutorial covers creating `FeatureColumn`s and an `input_fn` for a linear classification model that predicts income range based on census data.
- TensorFlow Wide & Deep Learning Tutorial (https://www.tensorflow.org/tutorials/wide_and_deep): Building on the Linear Model Tutorial (https://www.tensorflow.org/tutorials/wide), this tutorial covers `FeatureColumn` and `input_fn` creation for a "wide and deep" model that combines a linear model and a neural network using `DNNLinearCombinedClassifier`.

上次更新日期：六月 19, 2017