TensorFlow High-Level APIs: Using Estimators

In this section, we will use the high-level TensorFlow Estimator API for modeling with premade Estimators. Estimators provide another high-level API for building TensorFlow models for execution on CPUs, GPUs, or TPUs with minimal code modification.

The following steps are typically followed when working with premade Estimators:

1. Write the **‘input_fn’** to handle the data pipeline.
2. Define the type of data attributes into the model using feature columns **‘tf.feature_column’**.
3. Instantiate one of the premade Estimators by passing in the feature columns and other relevant attributes.
4. Use the **‘train()’**, **‘evaluate()’**, and **‘predict()’** methods to train and evaluate the model on evaluation dataset and use the model to make prediction/inference.

Let’s see a simple example of working with a TensorFlow premade Estimator again using the Boston housing dataset.

Note Reset session before running the following cells and change runtime type to None.

```
# import packages
import datetime
import numpy as np
import pandas as pd
import tensorflow as tf
from tensorflow.keras.datasets import boston_housing
from tensorflow.keras import Model
from sklearn.preprocessing import StandardScaler

# load dataset and split in train and test sets
(X_train, y_train), (X_test, y_test) = boston_housing.load_data()
```

```
# standardize the dataset
scaler_X_train = StandardScaler().fit(X_train)
scaler_X_test = StandardScaler().fit(X_test)
X_train = scaler_X_train.transform(X_train)
X_test = scaler_X_test.transform(X_test)

# reshape y-data to become column vector
y_train = np.reshape(y_train, [-1, 1])
y_test = np.reshape(y_test, [-1, 1])

# parameters
batch_size = 32
learning_rate = 0.01

# create an input_fn
def input_fn(features, labels, batch_size=30, training=True):
  dataset = tf.data.Dataset.from_tensor_slices((features, labels))
  if training:
      dataset = dataset.shuffle(buffer_size=1000)
      dataset = dataset.repeat()
  return dataset.batch(batch_size)

# use feature columns to define the attributes to the model
feature_columns = []
columns_names = []
for i in range(X_train.shape[1]):
  feature_columns.append(tf.feature_column.numeric_column(key=str(i)))
  columns_names.append(str(i))

# instantiate a LinearRegressor Estimator
estimator = tf.estimator.DNNRegressor(
    feature_columns=feature_columns,
    hidden_units=[20]
)

# convert feature datasets to dictionary
X_train_pd = pd.DataFrame(X_train)
X_train_pd.columns = columns_names
```

```
X_test_pd = pd.DataFrame(X_test)
X_test_pd.columns = columns_names

# train model
estimator.train(input_fn=lambda:input_fn(dict(X_train_pd), y_train),
steps=2000)

# evaluate model
metrics = estimator.evaluate(input_fn=lambda:input_fn(dict(X_test_pd),
y_test, training=False))

# print model metrics
metrics
```

Neural Networks with Keras

In this section, we will use the Sequential and Functional Keras API to build a simple neural network model. A Sequential API is the most commonly used method to build deep neural network models by stacking one layer on another. The Functional API offers more flexibility to build more complex neural network architectures. Both API methods are relatively easy to construct in Keras as we will see in the examples.

Subclassing a model as we did in the preceding examples provides even more flexibility for building and inspecting complex models. However, the code is more verbose and may be prone to errors. This technique should be used when it makes the most sense to, depending on the problem use case. We used them previously to serve as an illustration.

The following examples will use the Iris Dataset to build a neural network with one hidden layer as illustrated in Figure 30-11.