```
# evaluate the model
score = model.evaluate(test_ds)
print('Test loss: {:.2f} \nTest accuracy: {:.2f}%'.format(score[0],
score[1]*100))

# execute the command to run TensorBoard
%tensorboard --logdir tmp/logs_iris_keras
```

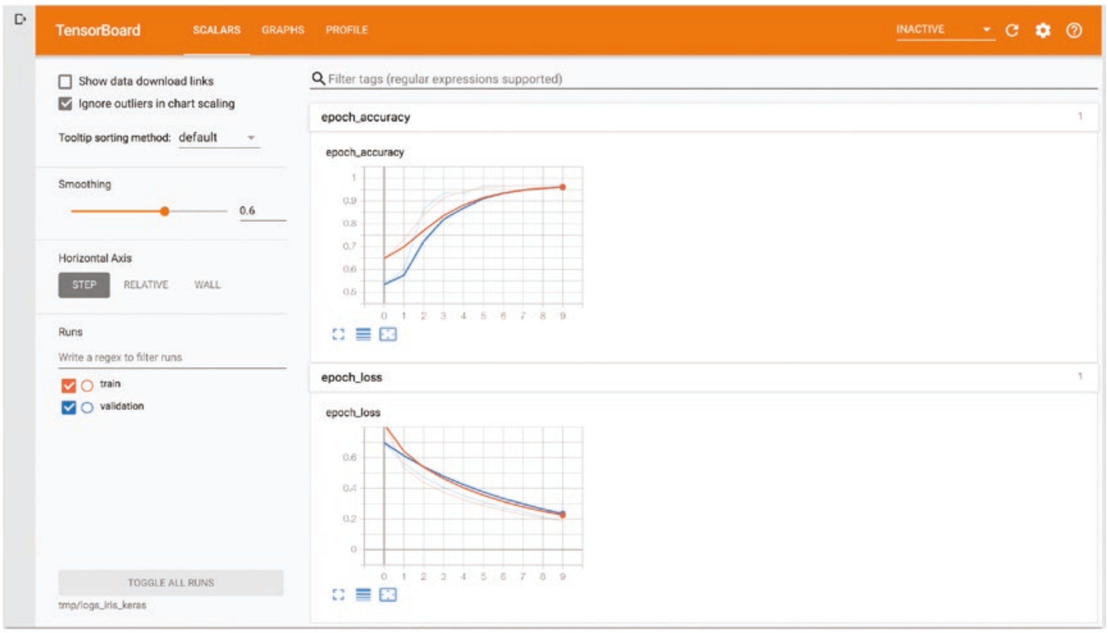

Figure 30-15. *TensorBoard output of Iris model*

Checkpointing to Select Best Models

Checkpointing makes it possible to save the weights of the neural network model when there is an increase in the validation accuracy metric. This is achieved in Keras using the **'tf.keras.callbacks.ModelCheckpoint()'**. The saved weights can then be loaded back into the model and used to make predictions. Using the Iris dataset, we'll build a model that saves the weights to file only when there is an improvement in the validation set performance. For completeness sake as we have done in the previous segments, we will produce this example within a complete code listing.

```
!pip install -q tensorflow==2.0.0-beta0
```

```
# import packages
import tensorflow as tf
import pandas as pd
from sklearn.preprocessing import OneHotEncoder

# dataset url
train_data_url = "https://storage.googleapis.com/download.tensorflow.org/
data/iris_training.csv"
test_data_url = "https://storage.googleapis.com/download.tensorflow.org/
data/iris_test.csv"

# define column names
columns = ['sepal_length', 'sepal_width', 'petal_length', 'petal_width', 'species']

# download and load the csv files
train_data = pd.read_csv(tf.keras.utils.get_file('iris_train.csv', train_
data_url),
                               skiprows=1, header=None, names=columns)

test_data = pd.read_csv(tf.keras.utils.get_file('iris_test.csv', test_data_url),
                               skiprows=1, header=None, names=columns)

# separate the features and targets
(X_train, y_train) = (train_data.iloc[:,0:-1], train_data.iloc[:,-1])
(X_test, y_test) = (test_data.iloc[:,0:-1], test_data.iloc[:,-1])

# apply one-hot encoding to targets
y_train=tf.keras.utils.to_categorical(y_train)
y_test=tf.keras.utils.to_categorical(y_test)

# create the functional model
def model_fn():
    # Model input
    model_input = tf.keras.layers.Input(shape=(4,))
    # Adds a densely-connected layer with 32 units to the model:
    x = tf.keras.layers.Dense(32, activation='relu')(model_input)
    # Add a softmax layer with 3 output units:
    predictions = tf.keras.layers.Dense(3, activation='softmax')(x)
    # the model
    model = tf.keras.Model(inputs=model_input,
                           outputs=predictions,
                           name='iris_model')

    # compile the model
    model.compile(optimizer='sgd',
                  loss='categorical_crossentropy',
                  metrics=['accuracy'])
    return model

# parameters
batch_size=50

# use tf.data to batch and shuffle the dataset
train_ds = tf.data.Dataset.from_tensor_slices(
    (X_train.values, y_train)).shuffle(len(X_train)).repeat().batch(batch_size)
test_ds = tf.data.Dataset.from_tensor_slices((X_test.values, y_test)).
batch(batch_size)

# build train model
model = model_fn()

# print train model summary
model.summary()

# checkpointing
checkpoint = tf.keras.callbacks.ModelCheckpoint(
    './tmp/iris_weights.h5',
    monitor='val_accuracy',
    verbose=1,
    save_best_only=True,
    mode='max')

# assign callback
callbacks = [checkpoint]
```

```
# train the model
history = model.fit(train_ds, epochs=10,
                    steps_per_epoch=100,
                    validation_data=test_ds,
                    callbacks=callbacks)

# build evaluation model and upload saved weights
eval_model = model_fn()
eval_model.load_weights('./tmp/iris_weights.h5')

# evaluate the model
score = eval_model.evaluate(test_ds)
print('Test loss: {:.2f} \nTest accuracy: {:.2f}%'.format(score[0],
score[1]*100))
```

This chapter covered the foundation of working with TensorFlow 2.0 and its exciting features for developing machine learning models. Some of these new features include a more pythonic feel to model design and debugging, using tf.function to transform a Python method into high-performance TensorFlow graphs, using Keras as the core high-level API for model design, using FeatureColumns to parse data as input into Keras models, and the ease of training on distributed architectures and devices. The chapter also covered the principles of building models using the high-level Estimator API.

In the next chapters, we will take a deeper dive into deep neural networks and how they are implemented in TensorFlow with Keras. In TensorFlow 2.0, Keras is the de facto method for developing neural network.

CHAPTER 31

The Multilayer Perceptron (MLP)

The multilayer perceptron (MLP) is the fundamental example of a deep neural network. The architecture of a MLP consists of multiple hidden layers to capture more complex relationships that exist in the training dataset. Another name for the MLP is the deep feedforward neural network (DFN). An illustration of an MLP is shown in Figure 31-1.

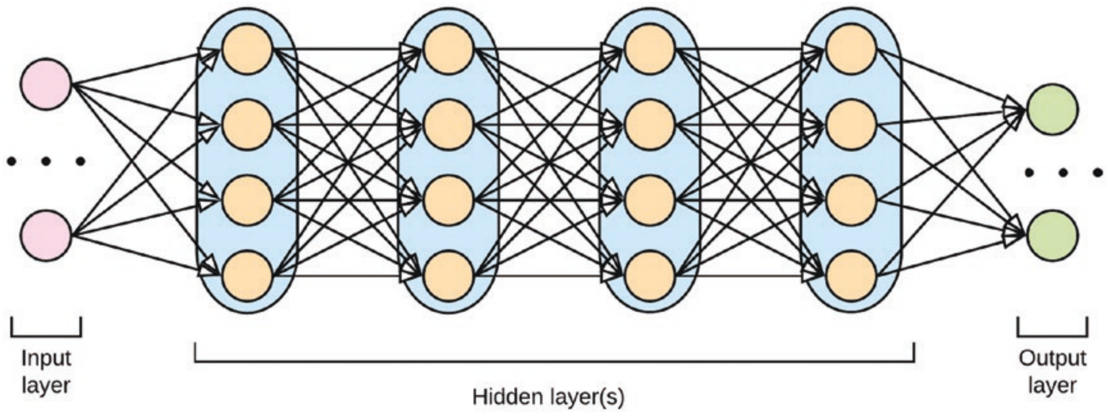

Figure 31-1. *Deep feedforward neural network*

The Concept of Hierarchies

The more the number of hidden layers in a neural network, the deeper the network becomes. Deep networks are able to learn more sophisticated representations of the inputs. The concept of hierarchical representation is when each layer learns a set of features that describe the input and hierarchically pass that information across the hidden layers. Initially, the hidden layers closer to the input layer learn a simple set

E. Bisong, *Building Machine Learning and Deep Learning Models on Google Cloud Platform*,
https://doi.org/10.1007/978-1-4842-4470-8_31