- The method to capture the loss and root mean squared error estimates is defined using **'tf.keras.metrics.Mean(name='train_loss')'** and **'tf.keras.metrics.RootMeanSquaredError()'** functions, respectively.
- The @tf.function is a python decorator to transform a method into high-performance TensorFlow graphs.
- The method **'train_step'** uses the **'tf.GradientTape()'** method to record operations for automatic differentiation. These gradients are later used to minimize the cost function by calling the **'apply_gradients()'** method of the optimization algorithm.
- The method **'test_step'** uses the trained model to obtain predictions on test data.

Classification with TensorFlow

In this example, we'll use the Iris flower dataset to build a multivariable logistic regression machine learning classifier with TensorFlow 2.0. The dataset is gotten from the Scikit-learn dataset package.

```
# import packages
import numpy as np
import tensorflow as tf
from sklearn import datasets
from tensorflow.keras import Model
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import OneHotEncoder

 # load dataset
data = datasets.load_iris()

# separate features and target
X = data.data
y = data.target

# apply one-hot encoding to targets
one_hot_encoder = OneHotEncoder(categories='auto')
```

```
encode_categorical = y.reshape(len(y), 1)
y = one_hot_encoder.fit_transform(encode_categorical).toarray()

# split in train and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, shuffle=True)

# build the linear model
class LogisticRegressionModel(Model):
  def __init__(self):
    super(LogisticRegressionModel, self).__init__()
    # initialize weight and bias variables
    self.weight = tf.Variable(
        initial_value = tf.random.normal(
            [4, 3], dtype=tf.float64),
        trainable=True)
    self.bias = tf.Variable(initial_value = tf.random.normal(
        [3], dtype=tf.float64), trainable=True)

  def call(self, inputs):
    return tf.add(tf.matmul(inputs, self.weight), self.bias)

model = LogisticRegressionModel()

# parameters
batch_size = 32
learning_rate = 0.1

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

optimizer = tf.keras.optimizers.SGD(learning_rate=learning_rate)

train_loss = tf.keras.metrics.Mean(name='train_loss')
train_accuracy = tf.keras.metrics.Accuracy(name='train_accuracy')

test_loss = tf.keras.metrics.Mean(name='test_loss')
test_accuracy = tf.keras.metrics.Accuracy(name='test_accuracy')
```

```
# use tf.GradientTape to train the model
@tf.function
def train_step(inputs, labels):
  with tf.GradientTape() as tape:
    predictions = model(inputs)
    loss = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(labels,
    predictions))
  gradients = tape.gradient(loss, model.trainable_variables)
  optimizer.apply_gradients(zip(gradients, model.trainable_variables))

  train_loss(loss)
  train_accuracy(tf.argmax(labels,1), tf.argmax(predictions,1))

@tf.function
def test_step(inputs, labels):
  predictions = model(inputs)
  t_loss = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(labels,
  predictions))

  test_loss(t_loss)
  test_accuracy(tf.argmax(labels,1), tf.argmax(predictions,1))

num_epochs = 1000

for epoch in range(num_epochs):
  for train_inputs, train_labels in train_ds:
    train_step(train_inputs, train_labels)

  for test_inputs, test_labels in test_ds:
    test_step(test_inputs, test_labels)

  template = 'Epoch {}, Loss: {}, Accuracy: {}, Test Loss: {}, Test Accuracy: {}'

  if ((epoch+1) % 100 == 0):
    print (template.format(epoch+1,
                           train_loss.result(),
                           train_accuracy.result()*100,
                           test_loss.result(),
                           test_accuracy.result()*100))
```

```
'Output':
Epoch 100, Loss: 0.3510790765285492, Accuracy: 89.63029479980469, Test
Loss: 0.4492445290088653 6, Test Accuracy: 84.37885284423828
Epoch 200, Loss: 0.3282322287559509, Accuracy: 91.29582214355469, Test
Loss: 0.4327660202980041 5, Test Accuracy: 85.73675537109375
Epoch 300, Loss: 0.3093726634979248, Accuracy: 92.46343231201172, Test
Loss: 0.41915151476860046, Test Accuracy: 86.6886978149414
Epoch 400, Loss: 0.29340484738349915, Accuracy: 93.3273696899414, Test
Loss: 0.4076262712478637 7, Test Accuracy: 87.43070220947266
Epoch 500, Loss: 0.2796294391155243, Accuracy: 93.99247741699219, Test
Loss: 0.3976936936378479, Test Accuracy: 88.27145385742188
Epoch 600, Loss: 0.2675718069076538, Accuracy: 94.52030944824219, Test
Loss: 0.38901543617248535, Test Accuracy: 88.93867492675781
Epoch 700, Loss: 0.25689396262168884, Accuracy: 94.94937896728516, Test
Loss: 0.38134896755218506, Test Accuracy: 89.48106384277344
Epoch 800, Loss: 0.24734711647033691, Accuracy: 95.3050537109375, Test
Loss: 0.3745149075984955, Test Accuracy: 89.9306640625
Epoch 900, Loss: 0.23874221742153168, Accuracy: 95.60466766357422, Test
Loss: 0.3683767020702362, Test Accuracy: 90.30940246582031
Epoch 1000, Loss: 0.23093272745609283, Accuracy: 95.86051177978516, Test
Loss: 0.3628271818161011, Test Accuracy: 90.63280487060547
```

From the preceding code, listing is similar to the example on linear regression with TensorFlow 2.0. However, take note of the following procedures:

- The target variable **'y'** is converted to a one-hot encoded matrix by using the **'OneHotEncoder'** function from Scikit-learn. There exists a TensorFlow method named **'tf.one_hot'** for performing the same function, even easier! The reader is encouraged to Experiment with this.
- Observe how the **'tf.reduce_mean'** and the **'tf.nn.softmax_cross_entropy_with_logits'** methods are used to implement the loss for optimizing the logistic model.
- The Stochastic Gradient Descent optimization algorithm **'tf.keras.optimizers.SGD()'** is used to train the logistic model.

- Observe how the **'weight'** and **'bias'** variables are updated by the gradient descent optimizer within the **'train_step'** method using **'tf. GradientTape()'** to capture and compute the derivatives from the trainable model variables.
- The **'tf.keras.metrics.Accuracy'** method is used to evaluate the accuracy of the model.

Visualizing with TensorBoard

In this section, we will go through visualizing TensorFlow graphs and statistics with TensorBoard. The following code improves on the previous code to build a linear regression model by adding methods to visualize the graph and other variable statistics in TensorBoard using the **'tf.summary'** method calls. The TensorBoard output (illustrated in Figure 30-9) is displayed within the notebook.

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

# load the TensorBoard notebook extension
%load_ext tensorboard

# 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])
```