From the preceding code listing, take note of the following:

- The method **'tf.data.Dataset.from_tensor_slices()'** is used to create a Dataset whose elements are Tensor slices.
- The Dataset method **'shuffle()'** shuffles the Dataset at each epoch.
- The Dataset method **'batch()'** is used to set the size of each mini-batch of the Dataset. In the preceding example, each Dataset batch contains five observations.

Linear Regression with TensorFlow

In this section, we use TensorFlow to implement a linear regression machine learning model. In the following example, we use the Boston house-prices dataset from the **Keras dataset package** to build a linear regression model with TensorFlow 2.0.

```
# import packages
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 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])

# build the linear model
class LinearRegressionModel(Model):
```

```
  def __init__(self):
    super(LinearRegressionModel, self).__init__()
    # initialize weight and bias variables
    self.weight = tf.Variable(
        initial_value = tf. random.normal(
            [13, 1], dtype=tf.float64),
        trainable=True)
    self.bias = tf.Variable(initial_value = tf.constant(
        1.0, shape=[], dtype=tf.float64), trainable=True)

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

model = LinearRegressionModel()

# parameters
batch_size = 32
learning_rate = 0.01

# 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)

loss_object = tf.keras.losses.MeanSquaredError()

optimizer = tf.keras.optimizers.SGD(learning_rate=0.01)

train_loss = tf.keras.metrics.Mean(name='train_loss')
train_rmse = tf.keras.metrics.RootMeanSquaredError(name='train_rmse')

test_loss = tf.keras.metrics.Mean(name='test_loss')
test_rmse = tf.keras.metrics.RootMeanSquaredError(name='test_rmse')

# use tf.GradientTape to train the model
@tf.function
def train_step(inputs, labels):
  with tf.GradientTape() as tape:
    predictions = model(inputs)
    loss = loss_object(labels, predictions)
```

```
  gradients = tape.gradient(loss, model.trainable_variables)
  optimizer.apply_gradients(zip(gradients, model.trainable_variables))

  train_loss(loss)
  train_rmse(labels, predictions)

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

  test_loss(t_loss)
  test_rmse(labels, predictions)

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: {}, RMSE: {}, Test Loss: {}, Test RMSE: {}'

  if ((epoch+1) % 100 == 0):
    print (template.format(epoch+1,
                           train_loss.result(),
                           train_rmse.result(),
                           test_loss.result(),
                           test_rmse.result()))

'Output':
Epoch 100, Loss: 23.531124114990234, RMSE: 4.862841606140137, Test Loss:
21.077274322509766, Test RMSE: 4.591667175292969
Epoch 200, Loss: 23.51316261291504, RMSE: 4.860987663269043, Test Loss:
21.067768096923828, Test RMSE: 4.590633869171143
Epoch 300, Loss: 23.496540069580078, RMSE: 4.859271049499512, Test Loss:
21.058971405029297, Test RMSE: 4.589677333831787
```

```
Epoch 400, Loss: 23.481115341186523, RMSE: 4.857677459716797, Test Loss:
21.050806045532227, Test RMSE: 4.588788986206055
Epoch 500, Loss: 23.466760635375977, RMSE: 4.856194019317627, Test Loss:
21.043209075927734, Test RMSE: 4.587962627410889
Epoch 600, Loss: 23.453369140625, RMSE: 4.8548102378845215, Test Loss:
21.036123275756836, Test RMSE: 4.587191581726074
Epoch 700, Loss: 23.440847396850586, RMSE: 4.853515625, Test Loss:
21.029495239257812, Test RMSE: 4.586470603942871
Epoch 800, Loss: 23.429113388061523, RMSE: 4.852302074432373, Test Loss:
21.02336311340332, Test RMSE: 4.585799694061279
Epoch 900, Loss: 23.4180965423584, RMSE: 4.851161956787109, Test Loss:
21.017648696899414, Test RMSE: 4.585177898406982
Epoch 1000, Loss: 23.407730102539062, RMSE: 4.8500895500183105, Test Loss:
21.012271881103516, Test RMSE: 4.584592819213867
```

Here are a few points and methods to take note of in the preceding code listing for linear regression with TensorFlow:

- Note that transformation to standardize the feature dataset is performed after splitting the data into train and test sets. This action is performed in this manner to prevent information from the training data to pollute the test data which must remain unseen by the model.
- The class named **'LinearRegressionModel'** builds a Keras model by subclassing the **'tf.keras.Model'** class. The linear regression model is created as a layer of the neural network in the **'__init__'** method, and it is defined as a forward pass in the **'call'** method. In Chapter 31 on Keras, we will see how to use simpler routines with the Keras Functional API.
- The **'tf.data.Dataset.from_tensor_slices'** method uses the **'.minimize()'** method to update the loss function.
- The squared error loss function is defined with **'tf.keras.losses.MeanSquaredError()'**.
- The gradient descent optimization algorithm is defined using **'tf.keras.optimizers.SGD()'** with the learning rate set as a parameter to the method.

- 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')
```