CHAPTER 3

Neural Networks and Deep Learning with TensorFlow

This chapter focuses on neural networks and how we can build them to perform machine learning, by closely mimicking the human brain. We will start by determining what neural networks are and how similarly they are structured to the neural network in humans. Then, we will deep dive into the architecture of neural networks, exploring the different layers within. We will explain how a simple neural network is built and delve into the concepts of forward and backward propagation. Later, we will build a simple neural network, using TensorFlow and Keras. In the final sections of this chapter, we will discuss deep neural networks, how they differ from simple neural networks, and how to implement deep neural networks with TensorFlow and Keras, again with performance comparisons to simple neural networks.

What Are Neural Networks?

Neural networks are a type of machine learning algorithm that tries to mimic the human brain. Computers always have been better at performing complex computations, compared to humans. They can do the calculations

P. Singh and A. Manure, *Learn TensorFlow 2.0*,
https://doi.org/10.1007/978-1-4842-5558-2_3

in no time, whereas for humans, it takes a while to perform even the simplest of operations manually. Then why do we need machines to mimic the human brain? The reason is that humans have common sense and imagination. They can be inspired by something to which computers cannot. If the computational capability of computers is combined with the common sense and imagination of humans, which can function continually 365 days a year, what is created? Superhumans? The response to those questions defines the whole purpose of artificial intelligence (AI).

Neurons

The human body consists of neurons, which are the basic building blocks of the nervous system. A neuron consists of a cell body, or soma, a single axon, and dendrites (Figure 3-1). Neurons are connected to one another by the dendrites and axon terminals. A signal from one neuron is passed to the axon terminal and dendrites of another connected neuron, which receives it and passes it through the soma, axon, and terminal, and so on.

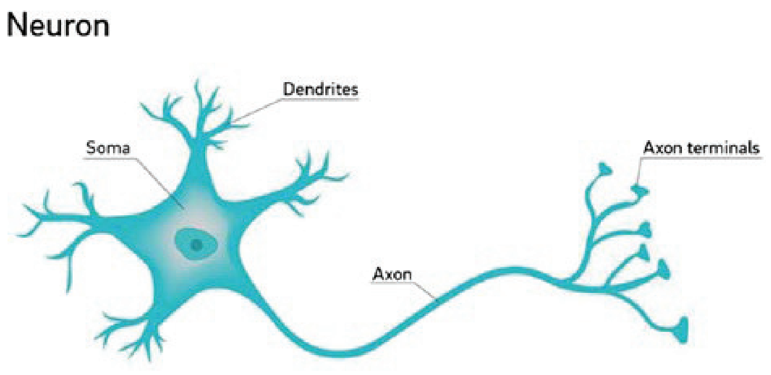


Figure 3-1. *Structure of a neuron (Source: `https://bit.ly/2zOekEL`)*

Neurons are interconnected in such a way that they have different functions, such as sensory neurons, which respond to such stimuli as sound, touch, or light; motor neurons, which control the muscle movements in the body; and interneurons, which are connected neurons within the same region of the brain or spinal cord.

Artificial Neural Networks (ANNs)

An artificial neural network tries to mimic the brain at its most basic level, i.e., that of the neuron. An artificial neuron has a similar structure to that of a human neuron and comprises the following sections (Figure 3-2):

Input layer: This layer is similar to dendrites and takes input from other networks/neurons.

Summation layer: This layer functions like the soma of neurons. It aggregates the input signal received.

Activation layer: This layer is also similar to a soma, and it takes the aggregated information and fires a signal only if the aggregated input crosses a certain threshold value. Otherwise, it does not fire.

Output layer: This layer is similar to axon terminals in that it might be connected to other neurons/networks or act as a final output layer (for predictions).

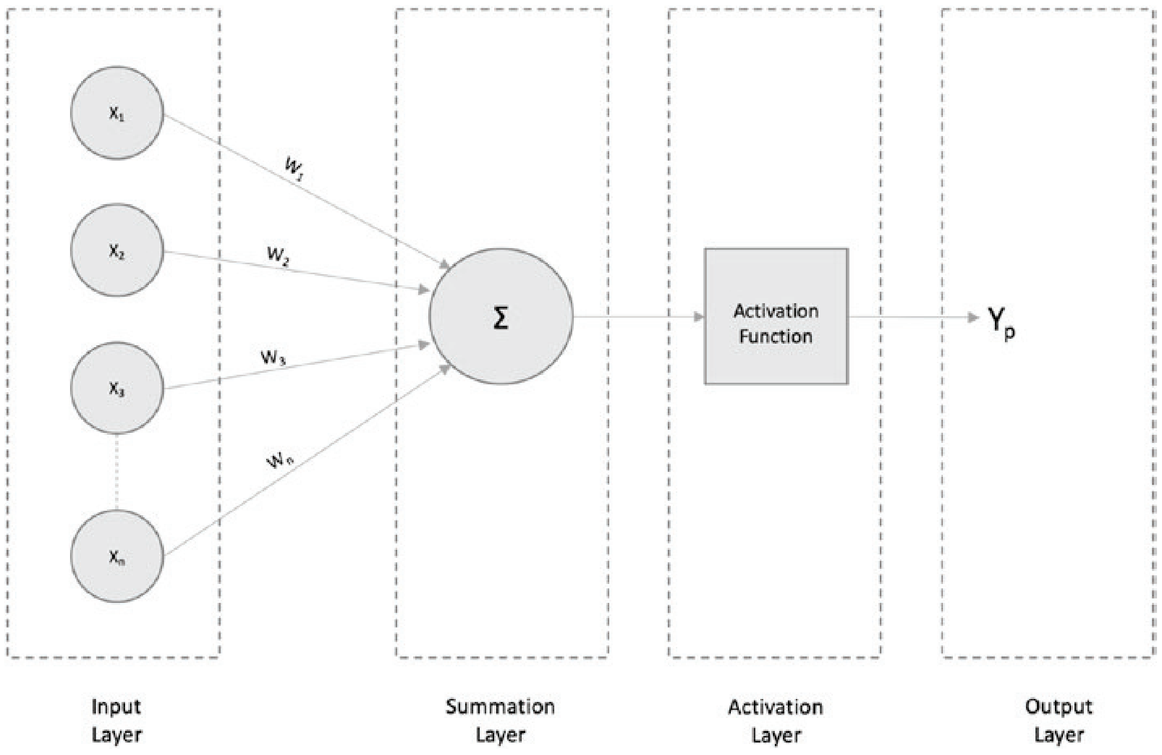


Figure 3-2. *Artificial neural network*

In the preceding figure, X_1, X_2, X_3,.........X_n are the inputs fed to the neural network. W_1, W_2, W_3,...........W_n are the weights associated with the inputs, and Y is the final prediction.

Many activation functions can be used in the activation layer, to convert all the linear details produced at the input and make the summation layer nonlinear. This helps users acquire more details about the input data that would not be possible if this were a linear function. Therefore, the activation layer plays an important role in predictions. Some of the most familiar types of activation functions are sigmoid, ReLU, and softmax.

Simple Neural Network Architecture

As shown in Figure 3-3, a typical neural network architecture is made up of an

Input layer

Hidden layer

Output layer

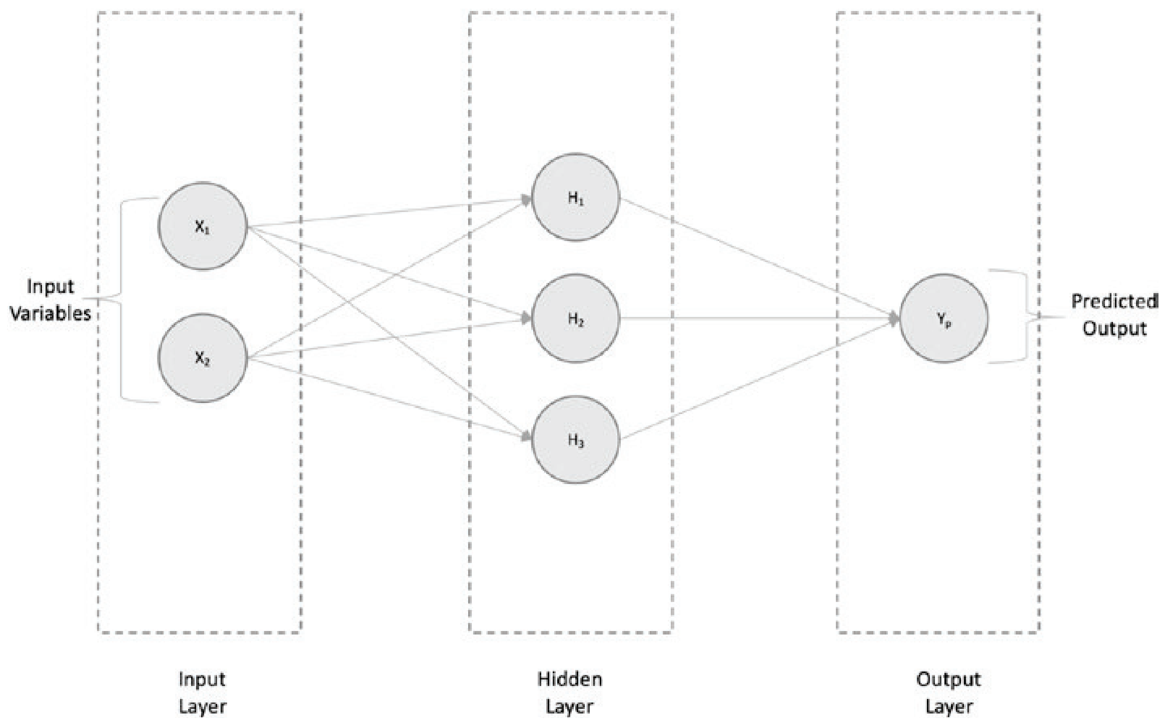


Figure 3-3. *Simple neural network architecture—regression*

Every input is connected to every neuron of the hidden layer and, in turn, connected to the output layer. If we are solving a regression problem, the architecture looks like the one shown in Figure 3-3, in which we have the output Y_p, which is continuous if predicted at the output layer. If we are solving a classification (binary, in this case), we will have the outputs Y_{class1} and Y_{class2}, which are the probability values for each of the binary classes 1 and 2 at the output layer, as shown in Figure 3-4.

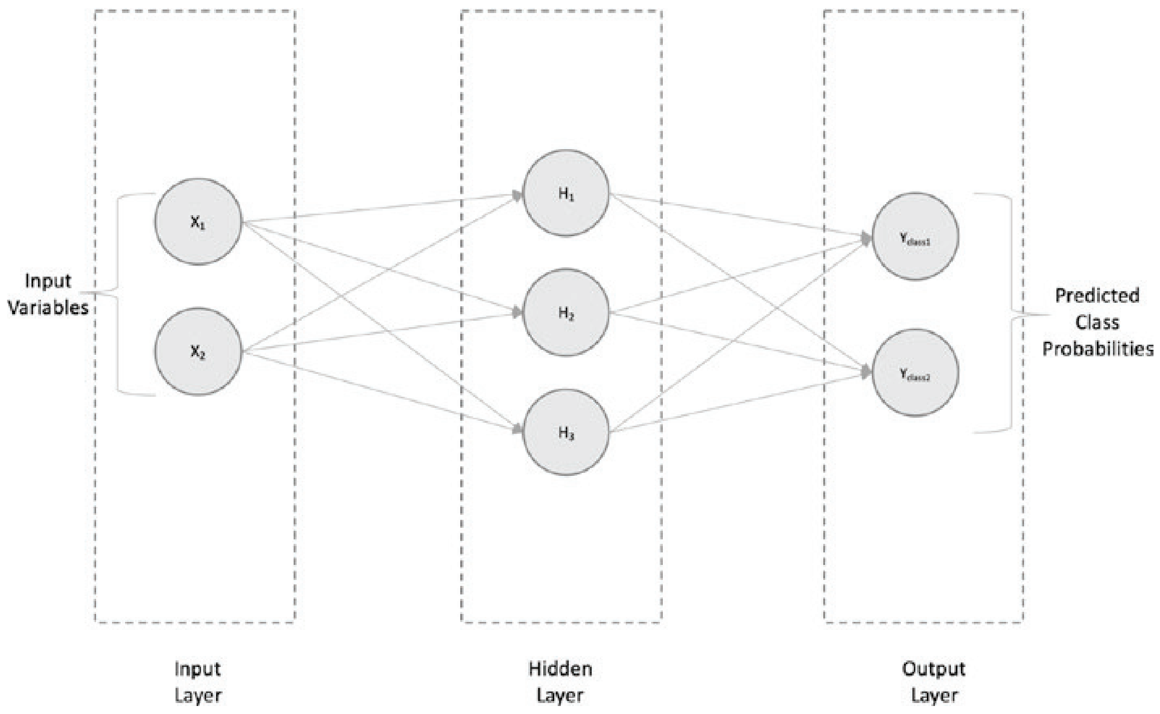


Figure 3-4. *Simple neural network architecture—classification*

Forward and Backward Propagation

In a fully connected neural network, when the inputs pass through the neurons (hidden layer to output layer), and the final value is calculated at the output layer, we say that the inputs have *forward propagated* (Figure 3-5). Consider, for example, a fully connected neural network with two inputs, X_1 and X_2, and one hidden layer with three neurons and an output layer with a single output Y_p (numeric value).

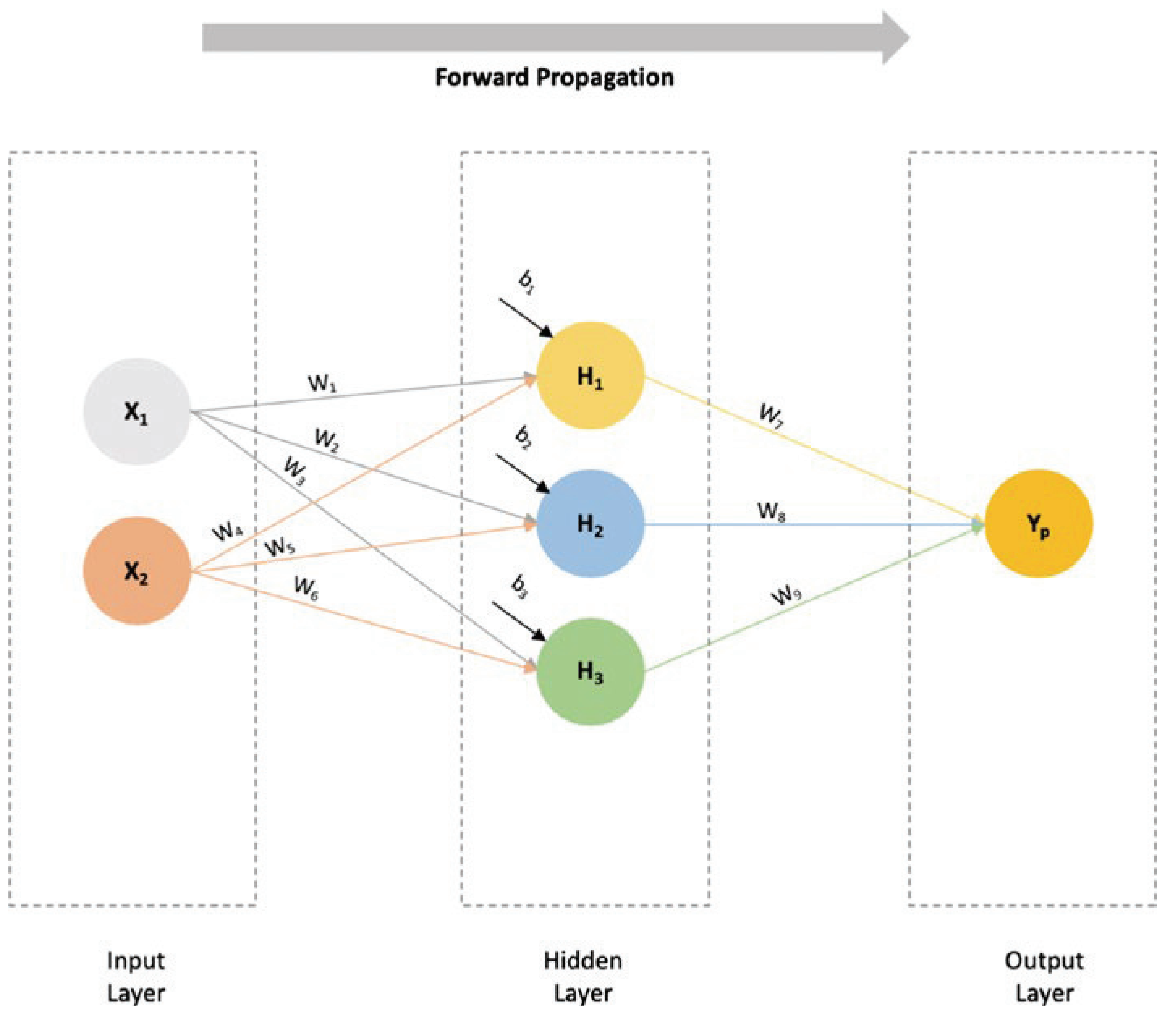


Figure 3-5. *Forward propagation*

The inputs will be fed to each of the hidden layer neurons, by multiplying each input value with a weight (W) and summing them with a bias value (b). So, the equations at the neurons' hidden layer will be as follows:

$$H_1 = W_1*X_1 + W_4*X_2 + b_1$$
$$H_2 = W_2*X_1 + W_5*X_2 + b_2$$
$$H_3 = W_3*X_1 + W_6*X_2 + b_3$$

The values H_1, H_2, and H_3 will be passed to the output layer, with weights W_7, W_8, and W_9, respectively. The output layer will produce the final predicted value of Y_p.

$$Y_p = W_7*H_1 + W_8*H_2 + W_9*H_3$$

As the input data (X_1 and X_2) in this network flows in a forward direction to produce the final outcome, Y_p, it is said to be a feed forward network, or, because the data is propagating in a forward manner, a *forward propagation*.

Now, suppose the actual value of the output is known (denoted by Y). In this case, we can calculate the difference between the actual value and the predicted value, i.e., $L = (Y - Y_p)^2$, where L is the loss value.

To minimize the loss value, we will try to optimize the weights accordingly, by taking a derivate of the loss function to the previous weights, as shown in Figure 3-6.

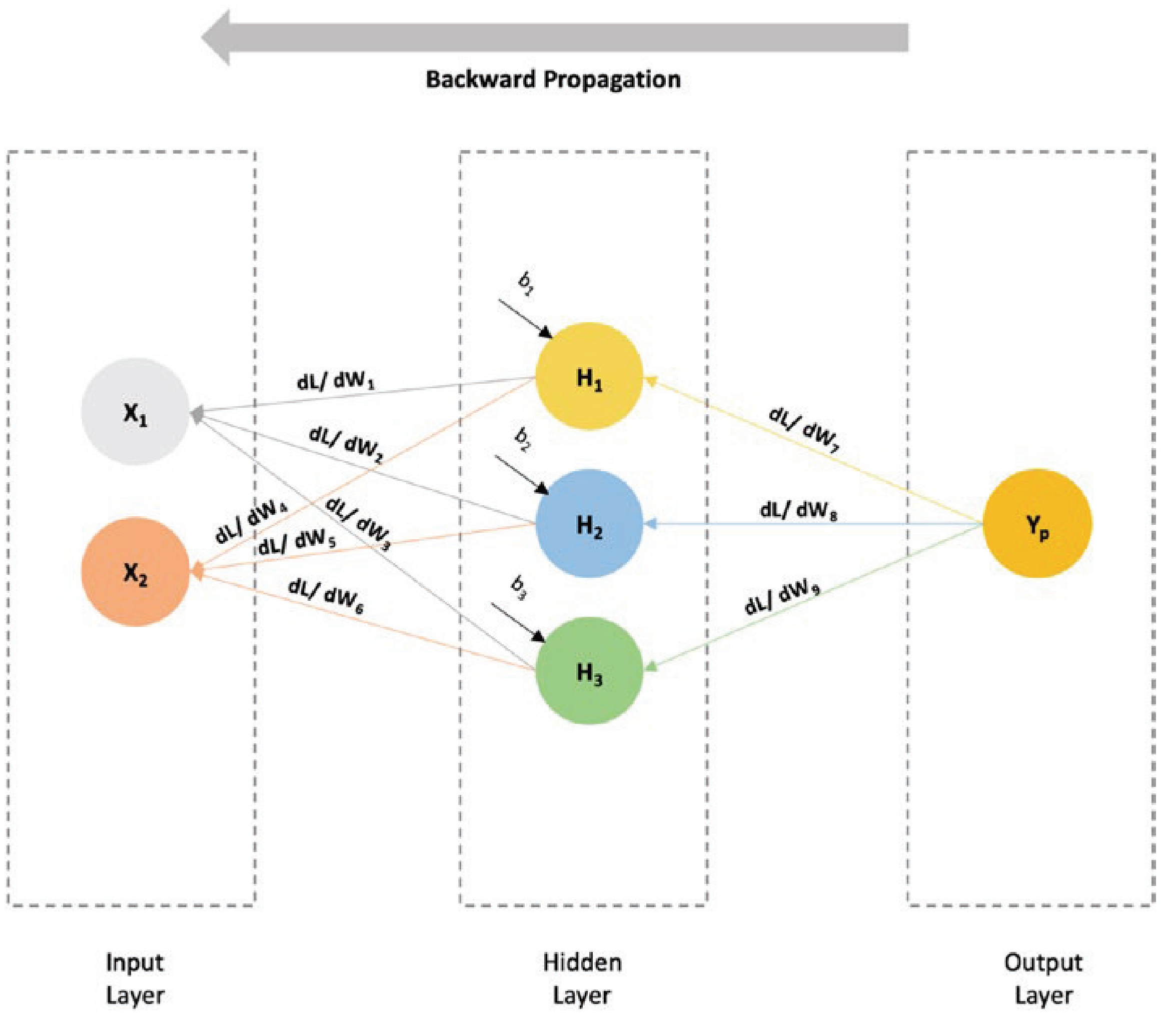


Figure 3-6. *Backward propagation*

For example, if we have to find the rate of change of loss function as compared to `W`$_7$, we would take a derivate of the `Loss` function to that of `W`$_7$ (`dL/dW`$_7$), and so on. As we can see from the preceding diagram, the process of taking the derivates is moving in a backward direction, that is, a backward propagation is occurring. There are multiple optimizers available to perform backward propagation, such as stochastic gradient descent (SGD), AdaGrad, among others.

Building Neural Networks with TensorFlow 2.0

Using the Keras API with TensorFlow, we will be building a simple neural network with only one hidden layer.

About the Data Set

Let's implement a simple neural network, using TensorFlow 2.0. For this, we will make use of the Fashion-MNIST data set by Zalando (The MIT License [MIT] Copyright © [2017] Zalando SE, `https://tech.zalando.com`), which contains 70,000 images (in grayscale) in 10 different categories. The images are 28 × 28 pixels of individual articles of clothing, with values ranging from 0 to 255, as shown in Figure 3-7.

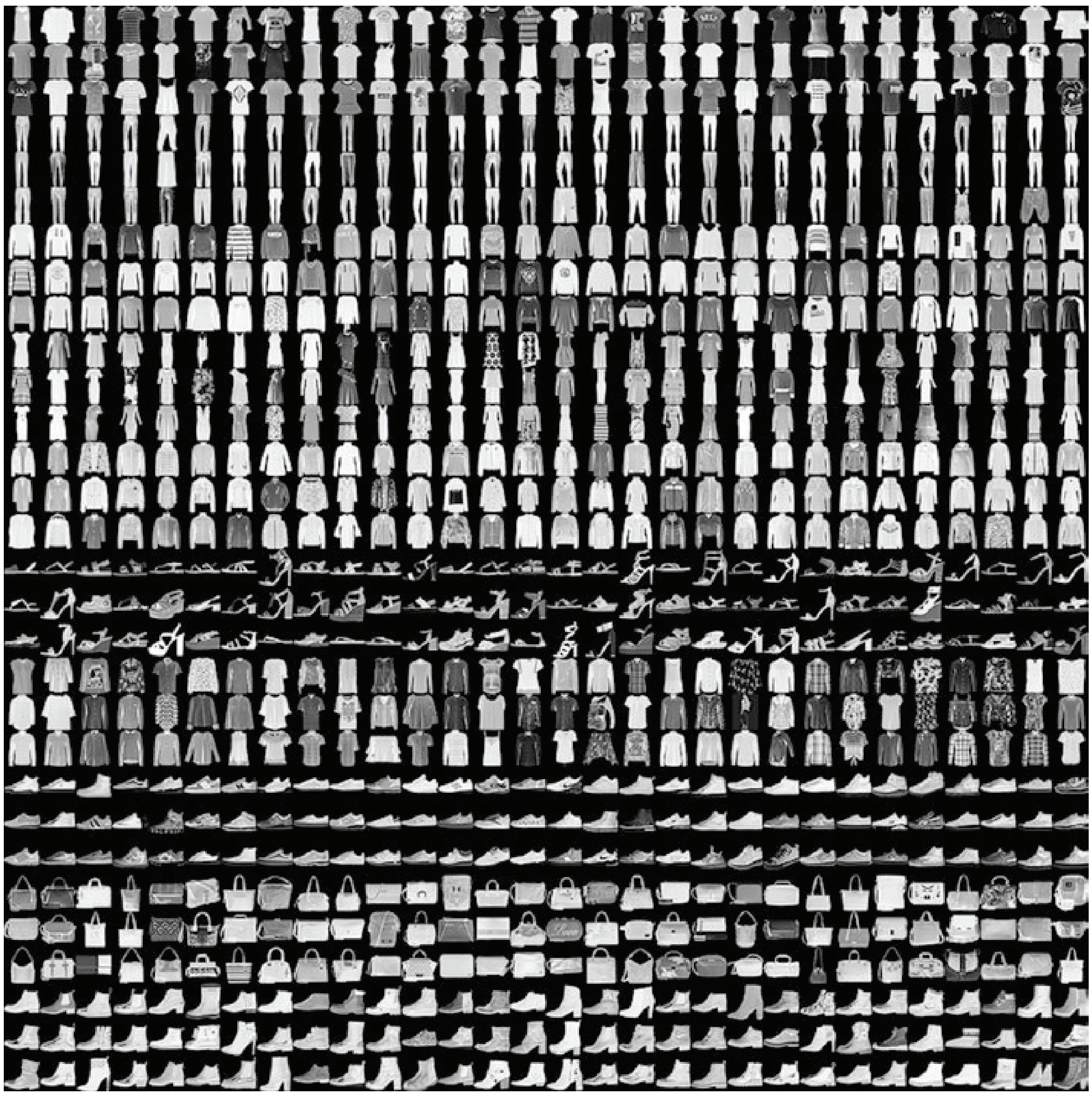
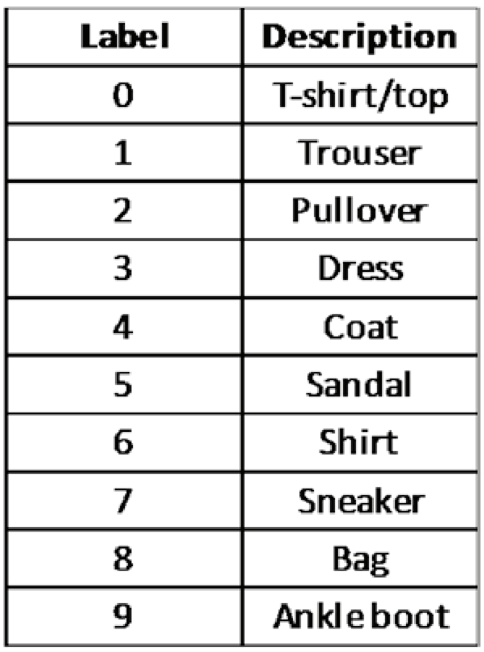


***Figure 3-7.** Sample from the Fashion-MNIST data set (Source: `https://bit.ly/2xqIwCH`)*

Of the total 70,000 images, 60,000 are used for training, and the remaining 10,000 images are used for testing. The labels are integer arrays ranging from 0 to 9. The class names are not part of the data set; therefore, we must include the following mapping for training/prediction:



Label	Description
0	T-shirt/top
1	Trouser
2	Pullover
3	Dress
4	Coat
5	Sandal
6	Shirt
7	Sneaker
8	Bag
9	Ankle boot

(Source: `https://bit.ly/2xqIwCH`)

Let's start by loading the necessary modules, as follows:

```
[In]: from __future__ import absolute_import, division, print_
      function, unicode_literals
[In]: import numpy as np
[In]: import tensorflow as tf
[In]: from tensorflow import keras as ks
[In]: print(tf.__version__)
[Out]: 2.0.0-rc1
```

Now, load the Fashion-MNIST data set.

```
[In]: (training_images, training_labels), (test_images, test_
      labels) = ks.datasets.fashion_mnist.load_data()
```

Let's undertake a little bit of data exploration, as follows:

```
[In]: print('Training Images Dataset Shape: {}'.
      format(training_images.shape))
[In]: print('No. of Training Images Dataset Labels: {}'.
      format(len(training_labels)))
[In]: print('Test Images Dataset Shape: {}'.format(test_images.
      shape))
[In]: print('No. of Test Images Dataset Labels: {}'.
      format(len(test_labels)))
[Out]: Training Images Dataset Shape: (60000, 28, 28)
[Out]: No. of Training Images Dataset Labels: 60000
[Out]: Test Images Dataset Shape: (10000, 28, 28)
[Out]: No. of Test Images Dataset Labels: 10000
```

As the pixel values range from 0 to 255, we have to rescale these values in the range 0 to 1 before pushing them to the model. We can scale these values (both for training and test data sets) by dividing the values by 255.

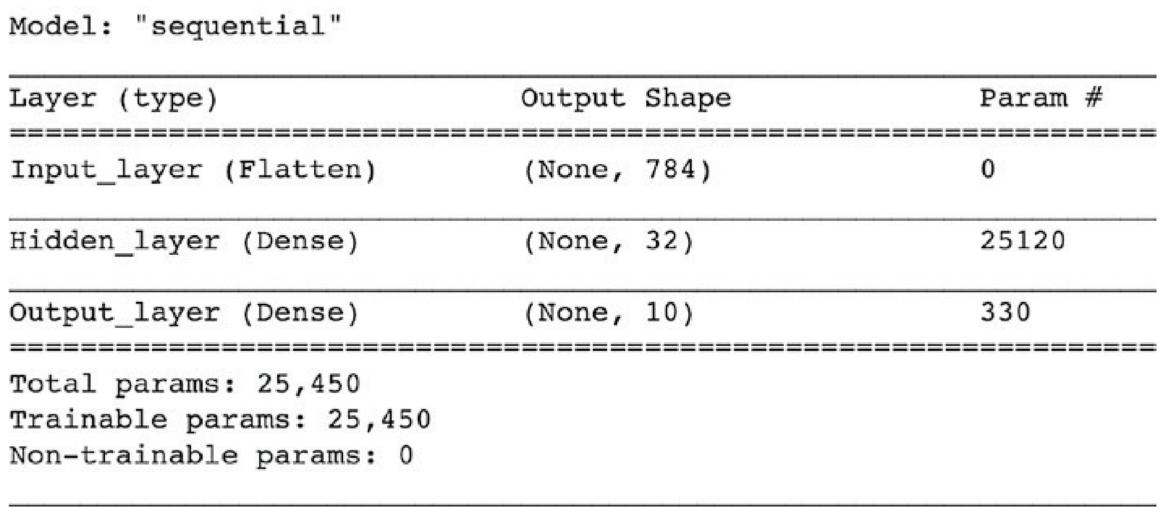
```
[In]: training_images = training_images / 255.0
[In]: test_images = test_images / 255.0
```

We will be using the Keras implementation to build the different layers of a neural network. We will keep it simple by having only one hidden layer.

```
[In]: input_data_shape = (28, 28)
[In]: hidden_activation_function = 'relu'
[In]: output_activation_function = 'softmax'

[In]: nn_model = models.Sequential()
[In]: nn_model.add(ks.layers.Flatten(input_shape=input_data_
      shape, name='Input_layer'))
[In]: nn_model.add(ks.layers.Dense(32, activation=hidden_
      activation_function, name='Hidden_layer'))
```

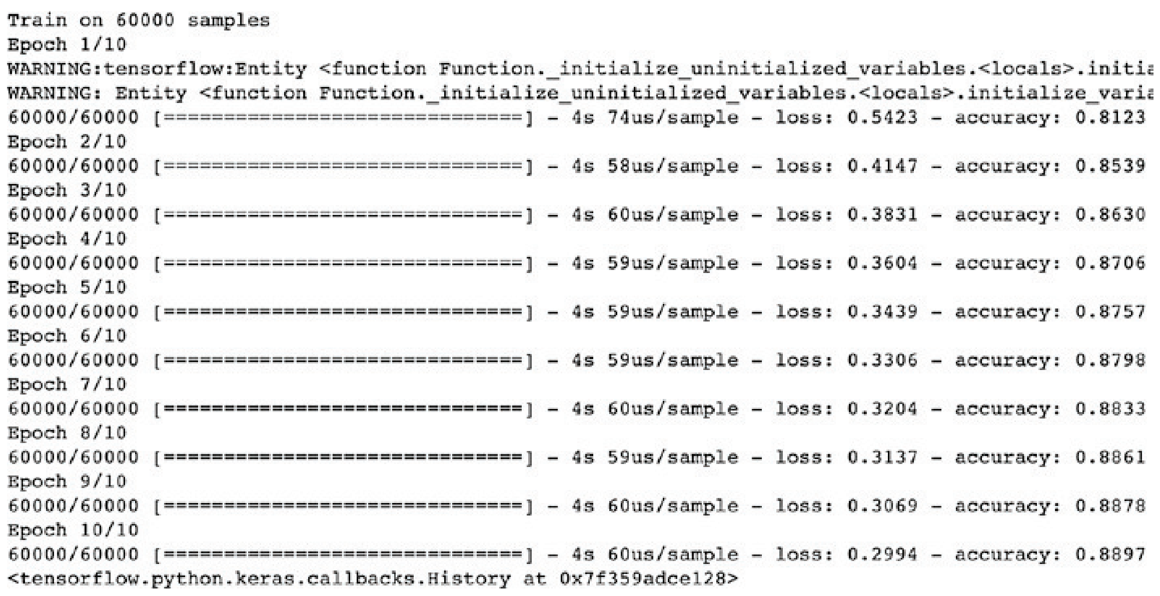
```
[In]: nn_model.add(ks.layers.Dense(10, activation=output_
      activation_function, name='Output_layer'))
[In]: nn_model.summary()
[Out]:
```



```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
Input_layer (Flatten)        (None, 784)               0
_________________________________________________________________
Hidden_layer (Dense)         (None, 32)                25120
_________________________________________________________________
Output_layer (Dense)         (None, 10)                330
=================================================================
Total params: 25,450
Trainable params: 25,450
Non-trainable params: 0
_________________________________________________________________
```

Now, we will use an optimization function with the help of the `compile` method. An Adam optimizer with the objective function `sparse_categorical_crossentropy`, which optimizes for the accuracy metric, can be built as follows:

```
[In]: optimizer = 'adam'
[In]: loss_function = 'sparse_categorical_crossentropy'
[In]: metric = ['accuracy']
[In]: nn_model.compile(optimizer=optimizer, loss=loss_function,
      metrics=metric)
[In]: nn_model.fit(training_images, training_labels, epochs=10)
[Out]:
```

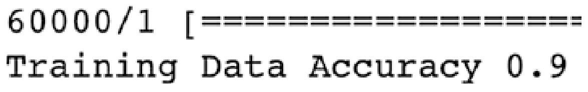


```
Train on 60000 samples
Epoch 1/10
WARNING:tensorflow:Entity <function Function._initialize_uninitialized_variables.<locals>.initia
WARNING: Entity <function Function._initialize_uninitialized_variables.<locals>.initialize_varia
60000/60000 [==============================] - 4s 74us/sample - loss: 0.5423 - accuracy: 0.8123
Epoch 2/10
60000/60000 [==============================] - 4s 58us/sample - loss: 0.4147 - accuracy: 0.8539
Epoch 3/10
60000/60000 [==============================] - 4s 60us/sample - loss: 0.3831 - accuracy: 0.8630
Epoch 4/10
60000/60000 [==============================] - 4s 59us/sample - loss: 0.3604 - accuracy: 0.8706
Epoch 5/10
60000/60000 [==============================] - 4s 59us/sample - loss: 0.3439 - accuracy: 0.8757
Epoch 6/10
60000/60000 [==============================] - 4s 59us/sample - loss: 0.3306 - accuracy: 0.8798
Epoch 7/10
60000/60000 [==============================] - 4s 60us/sample - loss: 0.3204 - accuracy: 0.8833
Epoch 8/10
60000/60000 [==============================] - 4s 59us/sample - loss: 0.3137 - accuracy: 0.8861
Epoch 9/10
60000/60000 [==============================] - 4s 60us/sample - loss: 0.3069 - accuracy: 0.8878
Epoch 10/10
60000/60000 [==============================] - 4s 60us/sample - loss: 0.2994 - accuracy: 0.8897
<tensorflow.python.keras.callbacks.History at 0x7f359adce128>
```

Following is the model evaluation:

1. Training evaluation

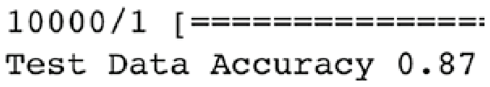
```
[In]: training_loss, training_accuracy = nn_model.
      evaluate(training_images, training_labels)
[In]: print('Training Data Accuracy {}'.
      format(round(float(training_accuracy),2)))
[Out]:
```



```
60000/1 [==================
Training Data Accuracy 0.9
```

2. Test evaluation

```
[In]: test_loss, test_accuracy = nn_model.evaluate(test_images,
      test_labels)
[In]: print('Test Data Accuracy {}'.format(round(float(test_
      accuracy),2))) [Out]:
```



```
10000/1 [===============
Test Data Accuracy 0.87
```

The code for the simple neural network implementation using TensorFlow 2.0 can be found at `http://bit.ly/NNetTF2`. You can save a copy of the code and run it in the Google Colab environment. Try experimenting with different parameters and note the results.

Deep Neural Networks (DNNs)

When a simple neural network has more than one hidden layer, it is known as a deep neural network (DNN). Figure 3-8 shows the architecture of a typical DNN.

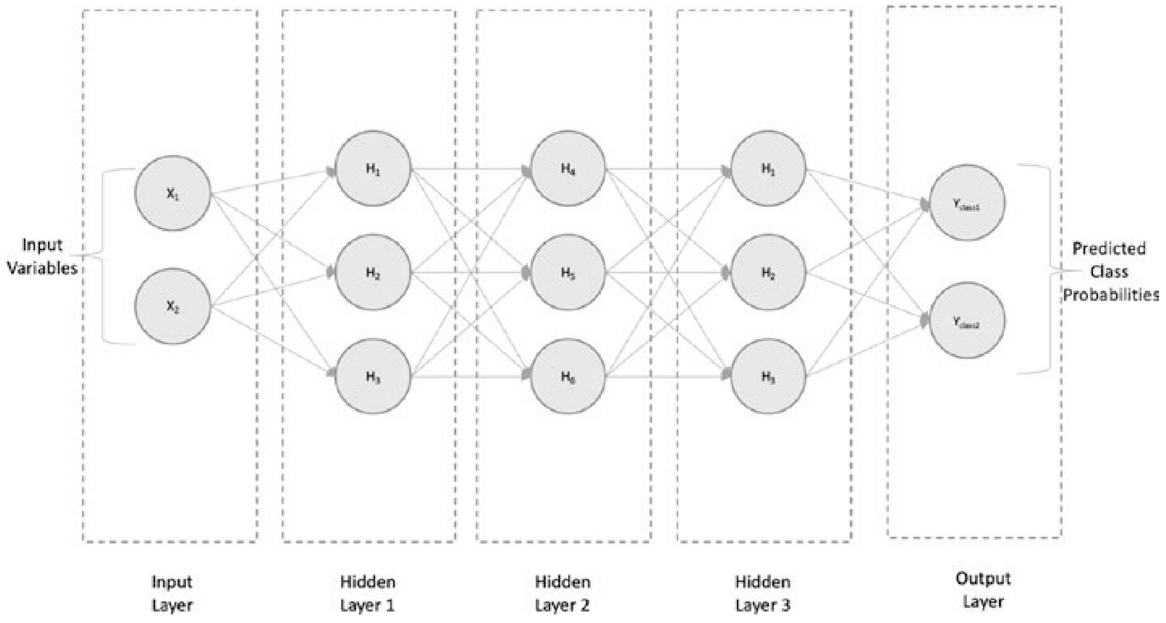


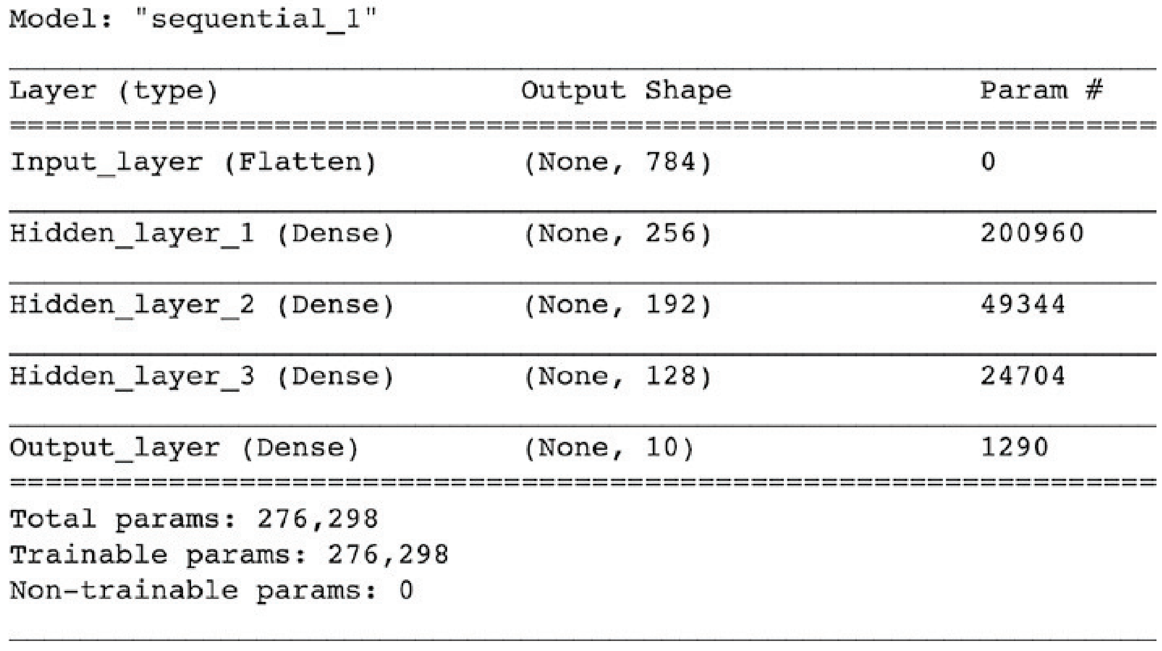
Figure 3-8. *Deep neural network with three hidden layers*

It consists of an input layer with two input variables, three hidden layers with three neurons each, and an output layer (consisting either of a single output for regression or multiple outputs for classification). The more hidden layers, the more neurons. Hence, the neural network is able to learn the nonlinear (non-convex) relation between the inputs and output. However, having more hidden layers adds to the computation cost, so one has to think in terms of a trade-off between computation cost and accuracy.

Building DNNs with TensorFlow 2.0

We will be using the Keras implementation to build a DNN with three hidden layers. The steps in the previous implementation of a simple neural network, up to the scaling part, is same for building the DNN. Therefore, we will skip those steps and start directly with building the input and hidden and output layers of the DNN, as follows:

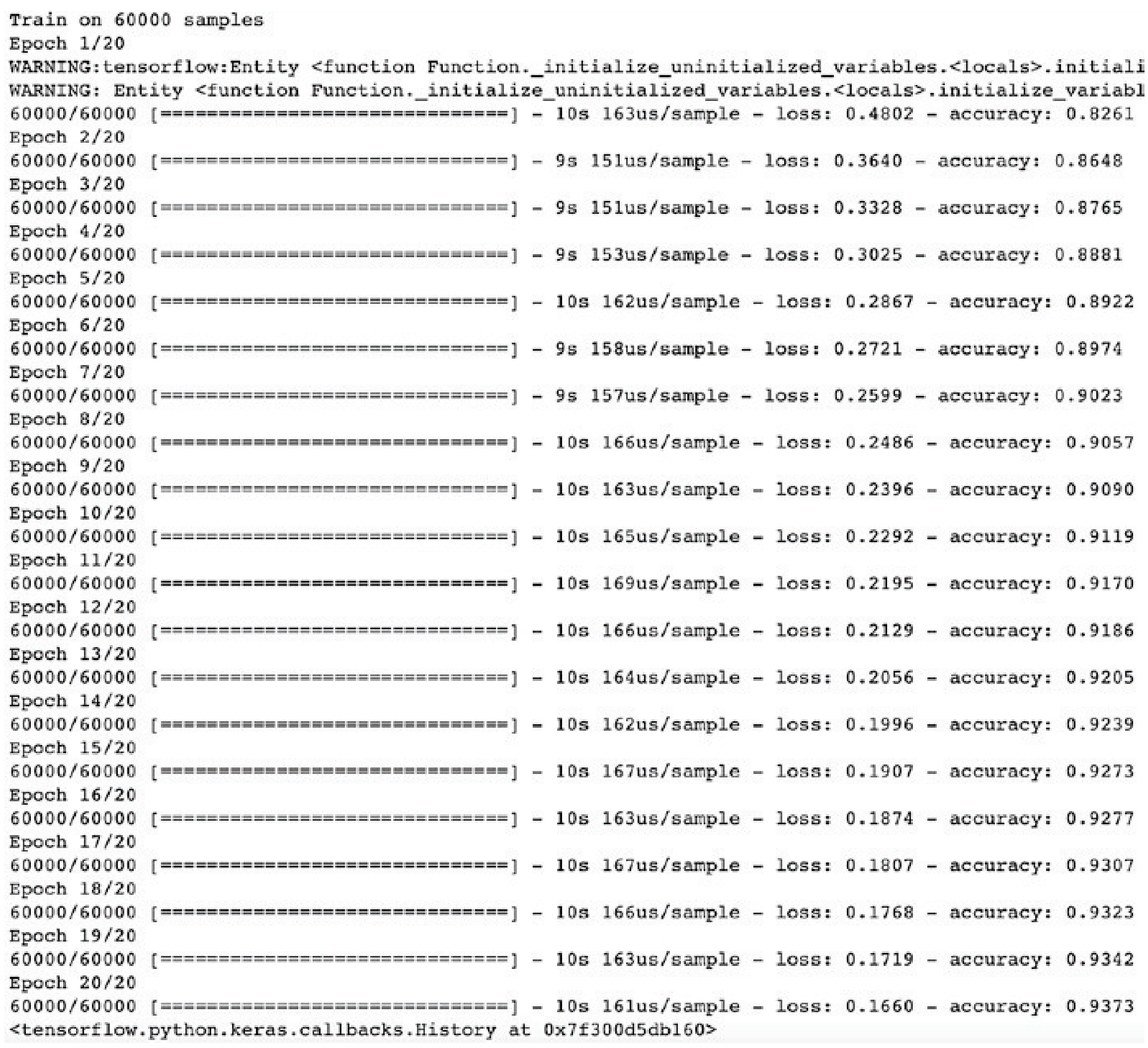
```
[In]: input_data_shape = (28, 28)
[In]: hidden_activation_function = 'relu'
[In]: output_activation_function = 'softmax'
[In]: dnn_model = models.Sequential()
[In]: dnn_model.add(ks.layers.Flatten(input_shape=input_data_
      shape, name='Input_layer'))
[In]: dnn_model.add(ks.layers.Dense(256, activation=hidden_
      activation_function, name='Hidden_layer_1'))
[In]: dnn_model.add(ks.layers.Dense(192, activation=hidden_
      activation_function, name='Hidden_layer_2'))
[In]: dnn_model.add(ks.layers.Dense(128, activation=hidden_
      activation_function, name='Hidden_layer_3'))
[In]: dnn_model.add(ks.layers.Dense(10, activation=output_
      activation_function, name='Output_layer'))
[In]: dnn_model.summary()
[Out]:
```



```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
Input_layer (Flatten)        (None, 784)               0
_________________________________________________________________
Hidden_layer_1 (Dense)       (None, 256)               200960
_________________________________________________________________
Hidden_layer_2 (Dense)       (None, 192)               49344
_________________________________________________________________
Hidden_layer_3 (Dense)       (None, 128)               24704
_________________________________________________________________
Output_layer (Dense)         (None, 10)                1290
=================================================================
Total params: 276,298
Trainable params: 276,298
Non-trainable params: 0
_________________________________________________________________
```

Now, we will use an optimization function with the help of the `compile` method. An Adam optimizer with the objective function `sparse_categorical_crossentropy`, which optimizes for the accuracy metric, can be built as follows:

```
[In]: optimizer = 'adam'
[In]: loss_function = 'sparse_categorical_crossentropy'
      metric = ['accuracy']
[In]: dnn_model.compile(optimizer=optimizer, loss=loss_
      function, metrics=metric)
[In]: dnn_model.fit(training_images, training_labels, epochs=20)
[Out]:
```

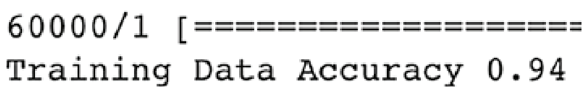


```
Train on 60000 samples
Epoch 1/20
WARNING:tensorflow:Entity <function Function._initialize_uninitialized_variables.<locals>.initiali
WARNING: Entity <function Function._initialize_uninitialized_variables.<locals>.initialize_variabl
60000/60000 [==============================] - 10s 163us/sample - loss: 0.4802 - accuracy: 0.8261
Epoch 2/20
60000/60000 [==============================] - 9s 151us/sample - loss: 0.3640 - accuracy: 0.8648
Epoch 3/20
60000/60000 [==============================] - 9s 151us/sample - loss: 0.3328 - accuracy: 0.8765
Epoch 4/20
60000/60000 [==============================] - 9s 153us/sample - loss: 0.3025 - accuracy: 0.8881
Epoch 5/20
60000/60000 [==============================] - 10s 162us/sample - loss: 0.2867 - accuracy: 0.8922
Epoch 6/20
60000/60000 [==============================] - 9s 158us/sample - loss: 0.2721 - accuracy: 0.8974
Epoch 7/20
60000/60000 [==============================] - 9s 157us/sample - loss: 0.2599 - accuracy: 0.9023
Epoch 8/20
60000/60000 [==============================] - 10s 166us/sample - loss: 0.2486 - accuracy: 0.9057
Epoch 9/20
60000/60000 [==============================] - 10s 163us/sample - loss: 0.2396 - accuracy: 0.9090
Epoch 10/20
60000/60000 [==============================] - 10s 165us/sample - loss: 0.2292 - accuracy: 0.9119
Epoch 11/20
60000/60000 [==============================] - 10s 169us/sample - loss: 0.2195 - accuracy: 0.9170
Epoch 12/20
60000/60000 [==============================] - 10s 166us/sample - loss: 0.2129 - accuracy: 0.9186
Epoch 13/20
60000/60000 [==============================] - 10s 164us/sample - loss: 0.2056 - accuracy: 0.9205
Epoch 14/20
60000/60000 [==============================] - 10s 162us/sample - loss: 0.1996 - accuracy: 0.9239
Epoch 15/20
60000/60000 [==============================] - 10s 167us/sample - loss: 0.1907 - accuracy: 0.9273
Epoch 16/20
60000/60000 [==============================] - 10s 163us/sample - loss: 0.1874 - accuracy: 0.9277
Epoch 17/20
60000/60000 [==============================] - 10s 167us/sample - loss: 0.1807 - accuracy: 0.9307
Epoch 18/20
60000/60000 [==============================] - 10s 166us/sample - loss: 0.1768 - accuracy: 0.9323
Epoch 19/20
60000/60000 [==============================] - 10s 163us/sample - loss: 0.1719 - accuracy: 0.9342
Epoch 20/20
60000/60000 [==============================] - 10s 161us/sample - loss: 0.1660 - accuracy: 0.9373
<tensorflow.python.keras.callbacks.History at 0x7f300d5db160>
```

Following is the model evaluation:

1. Training valuation

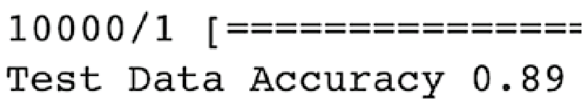
```
[In]: training_loss, training_accuracy = dnn_model.
      evaluate(training_images, training_labels)
[In]: print('Training Data Accuracy {}'.
      format(round(float(training_accuracy),2)))
[Out]:
```



```
60000/1 [==================
Training Data Accuracy 0.94
```

2. Test evaluation

```
[In]: test_loss, test_accuracy = dnn_model.evaluate(test_
      images, test_labels)
[In]: print('Test Data Accuracy {}'.format(round(float(test_
      accuracy),2)))
[Out]:
```



```
10000/1 [===============
Test Data Accuracy 0.89
```

The code for the DNN implementation using TensorFlow 2.0 can be found at `http://bit.ly/DNNTF2`. You can save a copy of the code and run it in the Google Colab environment. Try experimenting with different parameters and note the results.

As observed, the training accuracy for the simple neural network is about 90%, whereas it is 94% for the DNNs, and the test accuracy for the simple neural network is about 87%, whereas it is 89% for DNNs. It goes to show that we were able to achieve higher accuracy by adding more hidden layers to the neural network architecture.

Estimators Using the Keras Model

In Chapter 2, we built various machine learning models, using premade estimators. However, the TensorFlow API also provides enough flexibility for us to build custom estimators. In this section, you will see how we can create a custom estimator, using a Keras model. The implementation follows.

Let's start by loading the necessary modules.

```
[In]: from __future__ import absolute_import, division, print_
      function, unicode_literals
[In]: import numpy as np
[In]: import pandas as pd
```

```
[In]: import tensorflow as tf
[In]: from tensorflow import keras as ks
[In]: import tensorflow_datasets as tf_ds
[In]: print(tf.__version__)
[Out]: 2.0.0-rc1
```

Now, create a function to load the iris data set.

```
[In]: def data_input():
          train_test_split = tf_ds.Split.TRAIN
          iris_dataset = tf_ds.load('iris', split=train_test_
          split, as_supervised=True)
          iris_dataset = iris_dataset.map(lambda features,
          labels: ({'dense_input':features}, labels))
          iris_dataset = iris_dataset.batch(32).repeat()
          return iris_dataset
```

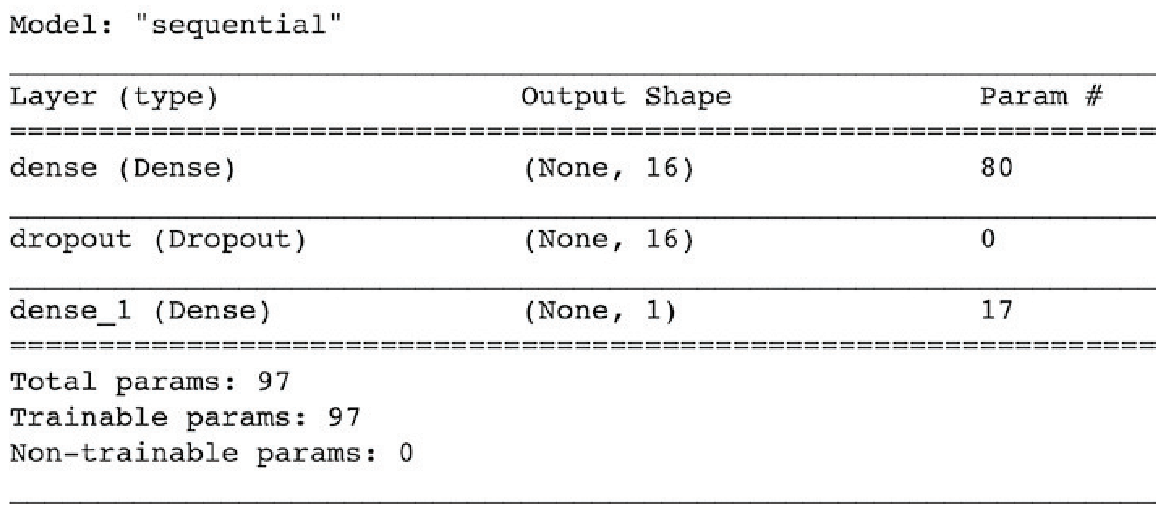
Build a simple Keras model.

```
[In]: activation_function = 'relu'
[In]: input_shape = (4,)
[In]: dropout = 0.2
[In]: output_activation_function = 'sigmoid'
[In]: keras_model = ks.models.Sequential([ks.layers.Dense(16,
      activation=activation_function, input_shape=input_
      shape), ks.layers.Dropout(dropout), ks.layers.Dense(1,
      activation=output_activation_function)])
```

Now, we will use an optimization function with the help of the compile method. An Adam optimizer with the loss function categorical_ crossentropy can be built as follows:

```
[In]: loss_function = 'categorical_crossentropy'
[In]: optimizer = 'adam'
[In]: keras_model.compile(loss=loss_function, optimizer=optimizer)
```

```
[In]: keras_model.summary()
[Out]:
```



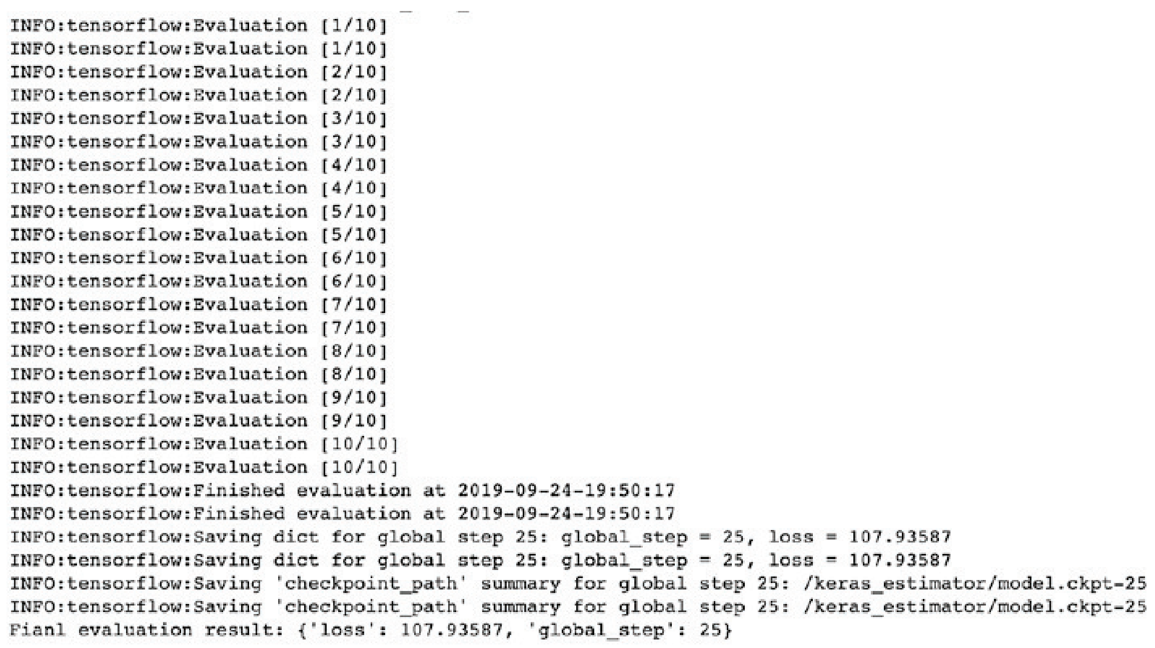
```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 16)                80
_________________________________________________________________
dropout (Dropout)            (None, 16)                0
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 17
=================================================================
Total params: 97
Trainable params: 97
Non-trainable params: 0
_________________________________________________________________
```

Build the estimator, using `tf.keras.estimator.model_to_estimator`:

```
[In]: model_path = "/keras_estimator/"
[In]: estimator_keras_model = ks.estimator.model_to_
      estimator(keras_model=keras_model, model_dir=model_path)
```

Train and evaluate the model.

```
[In]: estimator_keras_model.train(input_fn=data_input, steps=25)
[In]: evaluation_result = estimator_keras_model.evaluate(input_
      fn=data_input, steps=10)
[In]: print('Final evaluation result: {}'.format(evaluation_result))
[Out]:
```



```
INFO:tensorflow:Evaluation [1/10]
INFO:tensorflow:Evaluation [1/10]
INFO:tensorflow:Evaluation [2/10]
INFO:tensorflow:Evaluation [2/10]
INFO:tensorflow:Evaluation [3/10]
INFO:tensorflow:Evaluation [3/10]
INFO:tensorflow:Evaluation [4/10]
INFO:tensorflow:Evaluation [4/10]
INFO:tensorflow:Evaluation [5/10]
INFO:tensorflow:Evaluation [5/10]
INFO:tensorflow:Evaluation [6/10]
INFO:tensorflow:Evaluation [6/10]
INFO:tensorflow:Evaluation [7/10]
INFO:tensorflow:Evaluation [7/10]
INFO:tensorflow:Evaluation [8/10]
INFO:tensorflow:Evaluation [8/10]
INFO:tensorflow:Evaluation [9/10]
INFO:tensorflow:Evaluation [9/10]
INFO:tensorflow:Evaluation [10/10]
INFO:tensorflow:Evaluation [10/10]
INFO:tensorflow:Finished evaluation at 2019-09-24-19:50:17
INFO:tensorflow:Finished evaluation at 2019-09-24-19:50:17
INFO:tensorflow:Saving dict for global step 25: global_step = 25, loss = 107.93587
INFO:tensorflow:Saving dict for global step 25: global_step = 25, loss = 107.93587
INFO:tensorflow:Saving 'checkpoint_path' summary for global step 25: /keras_estimator/model.ckpt-25
INFO:tensorflow:Saving 'checkpoint_path' summary for global step 25: /keras_estimator/model.ckpt-25
Fianl evaluation result: {'loss': 107.93587, 'global_step': 25}
```

The code for the DNN implementation using TensorFlow 2.0 can be found at `http://bit.ly/KerasEstTF2`. You can save a copy of the code and run it in the Google Colab environment. Try experimenting with different parameters and note the results.

Conclusion

In this chapter, you have seen how easy it is to build neural networks in TensorFlow 2.0 and also how to leverage Keras models, to build custom TensorFlow estimators.