CHAPTER 6

TensorFlow Recognition Application

Building a DL model such as CNN from scratch using NumPy as we did helps us have a better understanding of how each layer works in detail. For practical applications, it is not recommended to use such implementation. One reason is that it is computationally intensive in its calculations and needs efforts to optimize the code. Another is that it does not support distributed processing, GPUs, and many more features. On the other hand, there are different already existing libraries that support these features in a time-efficient manner. These libraries include TF, Keras, Theano, PyTorch, Caffe, and more.

This chapter starts with introducing the TF DL library from scratch by building and visualizing the computational graph for a simple linear model and a two-class classifier using ANN. The computational graph is visualized using TensorBoard (TB). Using TF-Layers API, a CNN model is created to apply the concepts previously discussed for recognizing images from the CIFAR10 dataset.

Introduction to TF

There are different programming paradigms or styles for building software programs. They include sequential, which builds the programs as a set of sequential lines that the program follows from the beginning until the end; functional, which organizes the code into a set of functions that can be called multiple times; imperative, which tells the computer about every detailed step about how the program works; and more. One programming language might support different paradigms. But these paradigms have the disadvantage of being dependent on the language being written in.

© Ahmed Fawzy Gad 2018
A. F. Gad, *Practical Computer Vision Applications Using Deep Learning with CNNs*,
https://doi.org/10.1007/978-1-4842-4167-7_6

Another paradigm is dataflow. Dataflow languages represent their programs as text instructions that describe computational steps from receiving the data until returning the results. A dataflow program could be visualized as a graph that shows the operations in addition to their inputs and outputs. Dataflow languages support parallel processing because it is much easier to deduce the independent operations that could be executed at the same time.

The name “TensorFlow” consists of two words. The first is “tensor,” which is the data unit that TF uses in its computations. The second word is “flow,” reflecting that it uses the dataflow paradigm. As a result, TF builds a computational graph that consists of data represented as tensors and the operations applied to them. To make things easier to understand, just remember that rather than using variables and methods, TF uses tensors and operations.

Here are some advantages of using dataflow with TF:

- **Parallelism**: It is easier to identify the operations that can be executed in parallel.
- **Distributed Execution**: The TF program can be partitioned across multiple devices (CPUs, GPUs, and TF Processing Units [TPUs]). TF itself handles the necessary work for communication and cooperation between the devices.
- **Portability**: The dataflow graph is a language-independent representation of the code of the model. The dataflow graph can be created using Python, get saved, and then be restored in the C++ program.

TF provides multiple APIs; each supports a different level of control. The lowest-level API is called TF Core, which gives the programmer the ability to control every piece of code and have much better control over the created models.

But there are also a number of higher-level APIs in TF that make things easier by just providing a simple interface for frequently used tasks, such as Estimators, TF-Layers, and TF-Learn. All higher-level APIs are built on top of TF Core. For example, TF Estimators is a high-level API in TF that creates models much easier than TF Core.

Tensor

Tensor is the basic data unit in TF; it is similar to arrays in NumPy. Tensor consists of a set of primitive data types, such as integer, floating point, character, and string, which are shaped into an array.

A tensor has both rank and shape. Table 6-1 gives some tensor examples showing their ranks and shapes.

Table 6-1. *Ranks and Shapes of TF Tensors*

Tensor	Rank	Shape
5	0	()
[4, 8]	1	(2)
[[3, 1, 7], [1, 5, 2]]	2	(2,2)
[[[8, 3]], [[11, 9]]]]	2	(2,1,2)

The rank of a tensor is the number of dimensions. The tensor shape is similar to NumPy array shape. The NumPy array shape returns the number of elements within each dimension, and this is how tensor shape works. But tensor rank returns just the number of dimensions, which is similar to the ndim property of a NumPy array. Tensor rank is just a scalar value representing the number of dimensions in the tensor, while the shape is a tuple such as (4, 3) representing an array with two dimensions, where the sizes of these dimensions are 4 and 3, respectively.

Let's get started in TF Core.

TF Core

In order to create TF Core programs, there are two steps:

1. Building the computational graph.
2. Running the computational graph.

TF uses a dataflow graph to represent the computations in the program. After specifying the sequence of computations, it gets executed within a TF session on local or remote machines. Assume that Figure 6-1 represents a graph that has four operations, A, B, C, and D, where the inputs are fed into operation A and then propagated to operation D.

It is possible for the graph to just execute a selected part of it, and it is not required to run the complete graph. For example, by specifying that the target of session execution is operation C, then the program will run until reaching the operation C result only. That way will not execute operation D. Also, if operation B is the target, then operations C and D will not get executed.

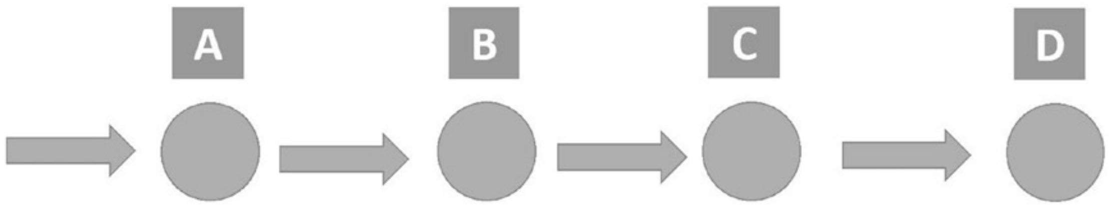

Figure 6-1. *A graph with four operations*

Working with TF Core API requires understanding of how dataflow graphs and sessions work. Working with high-level APIs such as Estimators hides some of the overhead from the user. But understanding how graphs and sessions work is useful for understanding in turn how such high-level APIs are implemented.

Dataflow Graph

A dataflow graph consists of nodes and edges. Nodes represent units of operation. Edges represent inputs to and outputs from an operation node. For example, the method tensorflow.matmul() accepts two input tensors, multiples them, and returns an output tensor. The operation itself is represented with a single node connected to two edges, one for each input tensor. There is also an edge that represents the output tensor. Later, we will see how to build the computational graph using TB.

A special kind of node is the constant, which accepts zero tensors as input. The output that the constant node returns is a value stored internally. Listing 6-1 creates a single constant node of type float32 and prints it.

Listing 6-1. Constant Node

```
import tensorflow
tensor1 = tensorflow.constant(3.7, dtype=tensorflow.float32)
print(tensor1)
```

When printing the constant node, the result is

```
Tensor("Const:0", shape=(), dtype=float32)
```

Based on the output of the `print` statement, there are three things to note:

- The shape is (), which means that the tensor is of rank 0.
- The output tensor has a string equal to "Const:0". This string is the name of the tensor. The tensor name is an important property because it is used to retrieve the tensor value from the graph. It is also the label printed in the TF graph. The default name for constant tensors is "Const". The 0 appended to this string defines it as the first output returned. There are some operations that return more than one output. The first output is given 0, the second one is given 1, and so on.
- The `print` statement does not print the value 3.7 but prints the node itself. The value will get printed only after evaluating the nodes.

Tensor Names

There might be multiple constant tensors within the graph. For this reason, TF appends the string "`Const`" with a number that identifies the constant among all constants in the graph. Listing 6-2 gives an example of three constants and prints them.

Listing 6-2. Creating Three Constants

```
import tensorflow

tensor1 = tensorflow.constant(value=3.7, dtype=tensorflow.float32)
tensor2 = tensorflow.constant(value=[[0.5], [7]], dtype=tensorflow.float32)
tensor3 = tensorflow.constant(value=[[12, 9]], dtype=tensorflow.float32)
print(tensor1)
print(tensor2)
print(tensor3)
```

Here is the result of the three print statements:

```
Tensor("Const:0", shape=(), dtype=float32)
Tensor("Const_1:0", shape=(2, 1), dtype=float32)
Tensor("Const_2:0", shape=(1, 2), dtype=float32)
```

The first tensor name is "Const:0". To differentiate it from other tensors, the string "Const" is appended by an underscore and a number. For example, the second tensor name is "Const_1:0". The number "1" is the identifier to that constant in the graph. But we can change the name of a tensor by using the name attribute as in Listing 6-3.

Listing 6-3. Setting Names of the Tensors Using the Name Attribute

```
import tensorflow

tensor1 = tensorflow.constant(value=3.7, dtype=tensorflow.float32,
name"firstConstant")
tensor2 = tensorflow.constant(value=[[0.5], [7]], dtype=tensorflow.float32,
name"secondConstant")
tensor3 = tensorflow.constant(value=[[12, 9]], dtype=tensorflow.float32,
name"thirdConstant")
print(tensor1)
print(tensor2)
print(tensor3)
```

The results of the three print statements are as follows:

```
Tensor("firstConstant:0", shape=(), dtype=float32)
Tensor("secondConstant:0", shape=(2, 1), dtype=float32)
Tensor("thirdConstant:0", shape=(1, 2), dtype=float32)
```

Because each tensor is given a unique name, there are no appended numbers to the string. If the same value of the name attribute is used for more than one tensor, the number will be used as in Listing 6-4. The first two tensors are given the value myConstant and thus the second tensor is appended by a number "1".

Listing 6-4. Two Tensors with the Same Value for the Name Attribute

```
import tensorflow

tensor1 = tensorflow.constant(value=3.7, dtype=tensorflow.float32,
name"myConstant")
tensor2 = tensorflow.constant(value=[[0.5], [7]], dtype=tensorflow.float32,
name"myConstant")
```

```
tensor3 = tensorflow.constant(value=[[12, 9]], dtype=tensorflow.float32,
name"thirdConstant")
print(tensor1)
print(tensor2)
print(tensor3)
```

The results of Listing 6-4 are as follows:

```
Tensor("myConstant:0", shape=(), dtype=float32)
Tensor("myConstant_1:0", shape=(2, 1), dtype=float32)
Tensor("thirdConstant:0", shape=(1, 2), dtype=float32)
```

In Listing 6-5, the operation `tensorflow.nn.top_k` is used to return the largest K values for a vector. In other words, this operation returns multiple values as outputs. Based on the output string, the two outputs are given the string "`TopKV2`" but with a different number after the colon. The first output is given number "0" and the second output is given "1".

Listing 6-5. Operation Returning Multiple Outputs

```
import tensorflow
aa = tensorflow.nn.top_k([1, 2, 3, 4], 2)
print(aa)
```

The print output is

```
TopKV2(values=<tf.Tensor 'TopKV2:0' shape=(2,) dtype=int32>, indices=<tf.
Tensor 'TopKV2:1' shape=(2,) dtype=int32>)
```

Up to this point, we have been able to print the tensor but not evaluate its result. Let's create a TF session for evaluating the operations.

Creating a TF Session

TF uses the `tensorflow.Session` class to represent a connection between the client program (typically a Python program) and the runtime environment. A `tensorflow.Session` object provides access to devices in the local machine and to remote devices using the distributed TF runtime environment. It also caches information about the `tensorflow.Graph` so that we can efficiently rerun the same graph. Listing 6-6 creates a TF session for evaluating the results of a single constant tensor. The tensor to be evaluated is assigned to the `fetches` attribute.

The session is created and returned into a variable named `sess`. After running the session using the `tensorflow.Session.run()` method to evaluate the tensor `tensor1`, the result will be 3.7, which is the constant value. This method runs the `tensorflow.Operation` and evaluates the `tensorflow.Tensor`. This method could accept more than one tensor for evaluation by typing them in a list and assigning this list to the `fetches` attribute.

Listing 6-6. Evaluating a Single Constant Tensor

```
import tensorflow

tensor1 = tensorflow.constant(value=3.7, dtype=tensorflow.float32)
sess = tensorflow.Session()
print(sess.run(fetches=tensor1))
sess.close()
```

As the `tensorflow.Session` owns physical resources such as CPUs, GPUs, and network connections, it must free these resources after finishing execution. According to Listing 6-6, we have to manually exit the session using the `tensorflow.Session.close()` to free resources. There is also another way to create a session, in which it gets closed automatically. This is by creating it using the `with` block as in Listing 6-7. When the session is created within the `with` block, it will get closed automatically after getting outside the block.

Listing 6-7. Creating a Session Using the With Block

```
import tensorflow

tensor1 = tensorflow.constant(value=3.7, dtype=tensorflow.float32)

with tensorflow.Session() as sess:
    print(sess.run(fetches=tensor1))
```

We can also specify more than one tensor in the `tensorflow.Session.run()` method to get their outputs, as in Listing 6-8.

Listing 6-8. Evaluating More Than One Tensor

```
import tensorflow

tensor1 = tensorflow.constant(value=3.7, dtype=tensorflow.float32)
tensor2 = tensorflow.constant(value=[[0.5], [7]], dtype=tensorflow.float32)
tensor3 = tensorflow.constant(value=[[12, 9]], dtype=tensorflow.float32)

with tensorflow.Session() as sess:
    print(sess.run(fetches=[tensor1, tensor2, tensor3]))
```

Here are the outputs of the three evaluated tensors.

```
3.7
array([[ 0.5], [7.]], dtype=float32)
array([[ 12.,   9.]], dtype=float32)
```

The previous examples just print the evaluated results for tensors. It is possible to store such values and reuse them in the program. Listing 6-9 returns the evaluation results in the `results` tensor.

Listing 6-9. Evaluating More Than One Tensor

```
import tensorflow

node1 = tensorflow.constant(value=3.7, dtype=tensorflow.float32)
node2 = tensorflow.constant(value=7.7, dtype=tensorflow.float32)
node3 = tensorflow.constant(value=9.1, dtype=tensorflow.float32)

with tensorflow.Session() as sess:
    results = sess.run(fetches=[node1, node2, node3])

vIDX = 0
for value in results:
    print("Value ", vIDX, " : ", value)
    vIDX = vIDX + 1
```

Because there are three tensors to be evaluated, all three outputs will be stored into the `results` tensor, which is a list. Using `for` loop, we can iterate and print each output separately. The outputs are as follows:

```
Value  0  :  3.7
Value  1  :  7.7
Value  2  :  9.1
```

The previous examples just evaluated the value of constant tensors without applying any operation. We can apply some operations over such tensors. Listing 6-10 creates two tensors and adds them together using the `tensorflow.add` operation. This operation accepts two tensors and adds them together. Both tensors must have the same data type (i.e., dtype attribute). It returns a new tensor of the same type as the input tensors. Using the + operator is identical to using the `tensorflow.add()` method.

Listing 6-10. Adding Two Tensors Using the tensorflow.add Operation

```
import tensorflow

tensor1 = tensorflow.constant(value=3.7, dtype=tensorflow.float32)
tensor2 = tensorflow.constant(value=7.7, dtype=tensorflow.float32)

add_op = tensorflow.add(tensor1, tensor2)

with tensorflow.Session() as sess:
    add_result = sess.run(fetches=[add_op])

print("Result of addition : ", add_result)
```

The output of the print statement is

```
Result of addition :  [11.4]
```

In Figure 6-2, the graph of the program in Listing 6-10 is visualized using TB. Note that all nodes and edges are given labels. These labels are the name of each tensor and operation. The default values are used. Later in this chapter, we will learn how to visualize graphs in TB.

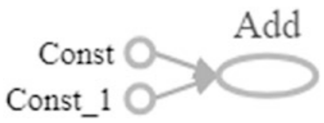

Figure 6-2. *Visualization of the graph using TB*

The name of the operation is descriptive and reflects its job, but the names of the tensors are not. We can change them to `num1` and `num2` and visualize the graph as in Figure 6-3.

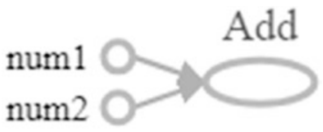

Figure 6-3. *Changing the name of the tensors*

Parameterized Graph Using Placeholder

The previous graph is static because it uses constant tensors. It always accepts the same inputs and generates the same output each time it gets evaluated. To be able to modify the inputs each time the program runs, we can use `tensorflow.placeholder`. In other words, for evaluating the same operation but using different inputs, you should use `tensorflow.placeholder`. Note that `placeholder` can get its value changed only by rerunning the graph.

The `tensorflow.placeholder`accepts three arguments as follows:

- `dtype`: Data type of elements the tensor will accept.
- `shape` (Optional – default None): Shape of the array within the tensor. If not specified, then you can feed the tensor with any shape.
- `name` (Optional – default None): Name for the operation.

It returns a tensor with these specifications.

We can modify the previous example in Listing 6-10 to use `tensorflow.placeholder` as in Listing 6-11. When running the session previously, the `tensorflow.Session.run()` accepts only the operations to be evaluated. When using `placeholders`, this method will also accept the initial values of the placeholders in the `feed_dict` argument. The `feed_dict` argument accepts the values as a dictionary that maps the name of each placeholder to its value.

Listing 6-11. Parameterized Graph Using a Placeholder

```
import tensorflow

tensor1 = tensorflow.placeholder(dtype=tensorflow.float32, shape=(),
name="num1")
tensor2 = tensorflow.placeholder(dtype=tensorflow.float32, shape=(),
name="num2")

add_op = tensorflow.add(tensor1, tensor2, name="Add_Op")

with tensorflow.Session() as sess:
    add_result = sess.run(fetches=[add_op], feed_dict={tensor1: 3.7,
    tensor2: 7.7})

print("Result of addition : ", add_result)
```

Assigning the placeholders the same values used by the constants in Listing 6-10, the same result will be returned. The benefit of using placeholders is that their values can be changed even within the program, but constants cannot be changed once created.

After using a third placeholder and a multiply operation, Listing 6-12 runs the session multiple times with different values for placeholders. It uses a `for` loop iterating through a list of five numbers returned by the `range()` native Python function. Values of all tensors are set equal to the list values, one value at each iteration. Values of the first two tensors are added using the `tensorflow.add` operation. The result of the addition is returned into the `add_op` tensor. Its value is then multiplied by the third tensor using the `tensorflow.multiply` operation. The multiplication result is returned in the `mul_op` tensor. Using the * operator is identical to using the `tensorflow.add()` method. The `fetches` argument in Listing 6-11 is a set of `add_op` compared to `mul_op` in Listing 6-12.

Listing 6-12. Running the Session for Different Values for the Placeholders

```
import tensorflow

tensor1 = tensorflow.placeholder(dtype=tensorflow.float32, shape=(),
name="num1")
tensor2 = tensorflow.placeholder(dtype=tensorflow.float32, shape=(),
name="num2")
tensor3 = tensorflow.placeholder(dtype=tensorflow.float32, shape=(),
name="num3")
```

```
add_op = tensorflow.add(tensor1, tensor2, name="Add_Op")
mul_op = tensorflow.multiply(add_op, tensor3, name="Add_Op")

with tensorflow.Session() as sess:
    for num in range(5):
result = sess.run(fetches=[mul_op], feed_dict={tensor1: num, tensor2: num,
tensor3: num})
        print("Result at iteration ", num, " : ", result)
```

The output of the print statement is as follows:

```
Result at iteration  0  :  [0.0]
Result at iteration  1  :  [2.0]
Result at iteration  2  :  [8.0]
Result at iteration  3  :  [18.0]
Result at iteration  4  :  [32.0]
```

A visualization of the previous graph is given in Figure 6-4. Note that all operations and tensors are renamed. The first two tensors `num1` and `num2` are connected with the first operation `Add_Op`. The result of this operation is used as input, with the third tensor `num3` as input to the second operation `Mul_Op`.

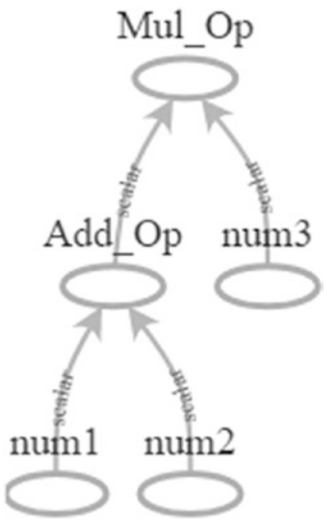

Figure 6-4. *Visualization of the graph in Listing 6-12 using TB*

The mul_op tensor is selected to be a member of the fetches list in Listing 6-12. Why not just select just add_op? The answer is that the last tensor in the graph chain is selected for evaluation. Evaluating mul_op will implicitly evaluate all other tensors in the graph. If "add_op" is selected for evaluation, then mul_op will not be evaluated because add_op doesn't depend on mul_op, and we have nothing to do for evaluating it. But mul_op is what depends on add_mul and all other tensors. Thus, mul_op is selected for evaluation. Remember that it is possible to use more than one tensor for evaluation.

TF Variables

Placeholders are used to allocate memory for future use. Their main use is for feeding input data for a model to get trained with. If the same operation is to be applied for different input data, then place the input data into a placeholder and run the session by assigning different values to the placeholder.

Placeholders are not initialized and their value is assigned only during runtime; in other words, only after calling the tensorflow.Session.run() are the placeholders assigned values. A placeholder allows the creation of an unconstrained shape tensor, which makes it suitable for use to hold the training data.

Suppose that you want to assign the training data to the placeholder and you just know that each sample gets described by 35 features. We have not decided yet how many samples to use for training. We can create a placeholder that accepts a tensor with an unspecified number of samples but a specific number of features (columns) per sample as follows:

```
data_tensor = tensorflow.placeholder(dtype=tensorflow.float16,
shape=[None, 35])
```

The placeholder just accepts the value and cannot get changed after being assigned. Remember that in Listing 6-12 we changed the value of the placeholders only by rebuilding the graph with the new values. In the same graph, it is not possible to change the placeholder value.

ML models have a number of trainable parameters that are changed multiple times until reaching their best values. How do we allow a tensor to change its values multiple times? This is not provided by constants and placeholders, but by variables (tensorflow.Variable()).

TF variables are identical to the normal variables used in other languages. They are assigned initial values, and such a value can be updated during the execution of the program based on the operations applied to it. A placeholder doesn't allow data modifications once assigned during execution time.

Constant tensors have their values initialized once the tensorflow.constant() is called, but variables won't be initialized after calling tensorflow.Variable(). There is an easy way to initialize all global variables within the program by running the tensorflow.global_variables_initializer() operation within the session. Note that initializing the variable does not mean it is evaluated. The variable needs to be evaluated after being initialized. Listing 6-13 gives an example of creating a single variable named "Var1" where its value is initialized, then the variable is evaluated, and finally, its value is printed.

Listing 6-13. Creating, Initializing, and Evaluating the Variable

```
import tensorflow

var1 = tensorflow.Variable(initial_value=5.8, dtype=tensorflow.float32,
name="Var1")

with tensorflow.Session() as sess:
    init = tensorflow.global_variables_initializer()
    sess.run(fetches=init)
    var_value = sess.run(fetches=var1)
    print("Variable value : ", var_value)
```

The print statement will return:

```
Variable value :5.8
```

Note that there are two runs to the session: the first for initializing all variables and the second for evaluating the variable. Remember that placeholder is a function but variable is a class, and thus its name starts with uppercase.

Variables can be initialized by a tensor of any type and shape. The type and shape of this tensor will define the type and shape of the variable, which cannot be changed. The variable value can be changed. Working with a distributed environment, variables can be stored once and get shared across all devices. They have a state that helps in debugging. Moreover, the variable value can be saved and restored when required.

Variable Initialization

There are different ways to initialize a variable. All variable initialization methods have can set both the shape and data type of the variable. One way is by using the initial value of a previously initialized variable. For example, the variable named "Var1" in Listing 6-13 is initialized by a rank 0 tensor of value 5.8. This initialized variable can be used to initialize other variables. Note that the initial value of a variable can be returned using the initialized_value() method of the tensorflow.Variable class. The initial value can be assigned to another variable as in the following. The variable "var3" is initialized by multiplying the initial value of "var1" by 5.

```
var2 = tensorflow.Variable(initial_value=var1.initialized_value(),
dtype=tensorflow.float32)
var3 = tensorflow.Variable(initial_value=var1.initialized_value()*5,
dtype=tensorflow.float32)
```

A variable can be initialized based on another tensor created by one of the build-in operations in TF. There are different operations to generate tensors, including the following:

- tensorflow.lin_space(start, stop, num, name=None)
- tensorflow.range(start, limit=None, delta=1, dtype=None, name='range')
- tensorflow.zeros(shape, dtype=tf.float32, name=None)
- tensorflow.ones(shape, dtype=tf.float32, name=None)
- tensorflow.constant(value, dtype=None, shape=None, name='Const', verify_shape=False)

They have the same meaning as their corresponding methods in NumPy. All of these operations return the tensor of the specified data type and shape. For example, we can create a tensorflow.Variable() whose values are initialized using tensorflow.zeros(), which returns a 1D row vector with 12 elements as follows:

```
var1 = tensorflow.Variable(tensorflow.zeros([12]))
```

Graph Visualization Using TB

TF is designed to work with deep models trained with large amounts of data. TF supports a suite of visualization tools called TB to help to optimize and to debug TF programs easier. The computational dataflow graph is visualized as a set of nodes representing the operations, which are connected together with edges representing the input and output tensors.

Here are the summarized steps for visualizing a simple graph using TB:

1. Build the dataflow graph.
2. Write the graph in a directory using tensorflow.summary. FileWriter.
3. Launch TB within the directory of the saved graph.
4. Access TB from a web browser.
5. Visualize the graph.

Let's use the code in Listing 6-14 for visualization. This code creates six variables that are fed into nine operations. After writing the instructions for building the graph, next is to save it using FileWriter. The tensorflow.summary.FileWriter() constructor accepts two important arguments: "graph" and "logdir". The "graph" argument accepts the session graph, which is returned by "sess.graph" assuming that the session variable is named "sess". The graph is exported into the directory specified using the "logdir" argument. Change the "logdir" to match your system. Note that we do not have to initialize the variables nor run the session because our target is not to execute the graph but just to visualize it.

Listing 6-14. Saving Dataflow Graph for Visualization Using TB

```
import tensorflow

tensor1 = tensorflow.Variable(initial_value=4, dtype=tensorflow.float32,
name="Var1")
tensor2 = tensorflow.Variable(initial_value=15, dtype=tensorflow.float32,
name="Var2")
tensor3 = tensorflow.Variable(initial_value=-2, dtype=tensorflow.float32,
name="Var3")
```

```
tensor4 = tensorflow.Variable(initial_value=1.8, dtype=tensorflow.float32,
name="Var4")
tensor5 = tensorflow.Variable(initial_value=14, dtype=tensorflow.float32,
name="Var5")
tensor6 = tensorflow.Variable(initial_value=8, dtype=tensorflow.float32,
name="Var6")

op1 = tensorflow.add(x=tensor1, y=tensor2, name="Add_Op1")
op2 = tensorflow.subtract(x=op1, y=tensor1, name="Subt_Op1")
op3 = tensorflow.divide(x=op2, y=tensor3, name="Divide_Op1")
op4 = tensorflow.multiply(x=op3, y=tensor4, name="Mul_Op1")
op5 = tensorflow.multiply(x=op4, y=op1, name="Mul_Op2")
op6 = tensorflow.add(x=op5, y=2, name="Add_Op2")
op7 = tensorflow.subtract(x=op6, y=op2, name="Subt_Op2")
op8 = tensorflow.multiply(x=op7, y=tensor6, name="Mul_Op3")
op9 = tensorflow.multiply(x=op8, y=tensor5, name="Mul_Op4")

with tensorflow.Session() as sess:
    writer = tensorflow.summary.FileWriter(logdir="\\AhmedGad\\
TensorBoard\\", graph=sess.graph)
    writer.close()
```

After exporting the graph, the next step is to launch TB to access the graph. Launching TB differs a bit based on whether TF is installed in a separate virtual environment (venv) or as a regular library within the site-packages directory.

If it is installed into a venv, then TF must be activated using the activate.bat file located under the Scripts directory of the Python installation. Assuming that the Scripts directory is added to either the user or system PATH variable environment and the venv folder is named "tensorflow", then TF will be activated according to the following command:

```
activate tensorflow
```

After activating TF, next is to launch TB into the directory at which the graph is saved according to this command:

```
tensorBoard --logdir=\\AhmedGad\\TensorBoard\\
```

In case the TF is installed within the site-packages directory, then it can be activated by issuing this command:

```
python -m tensorboard.main --logdir="\\AhmedGad\\TensorBoard\\"
```

This will activate TB, and we will then be ready to visualize the graph by navigating to "http://localhost:6006" from a web browser. The graph is shown in Figure 6-5. It is easier to debug the graph in this case. For example, an isolated node that is not connected to any other node of the graph is easily detected in the graph than the code.

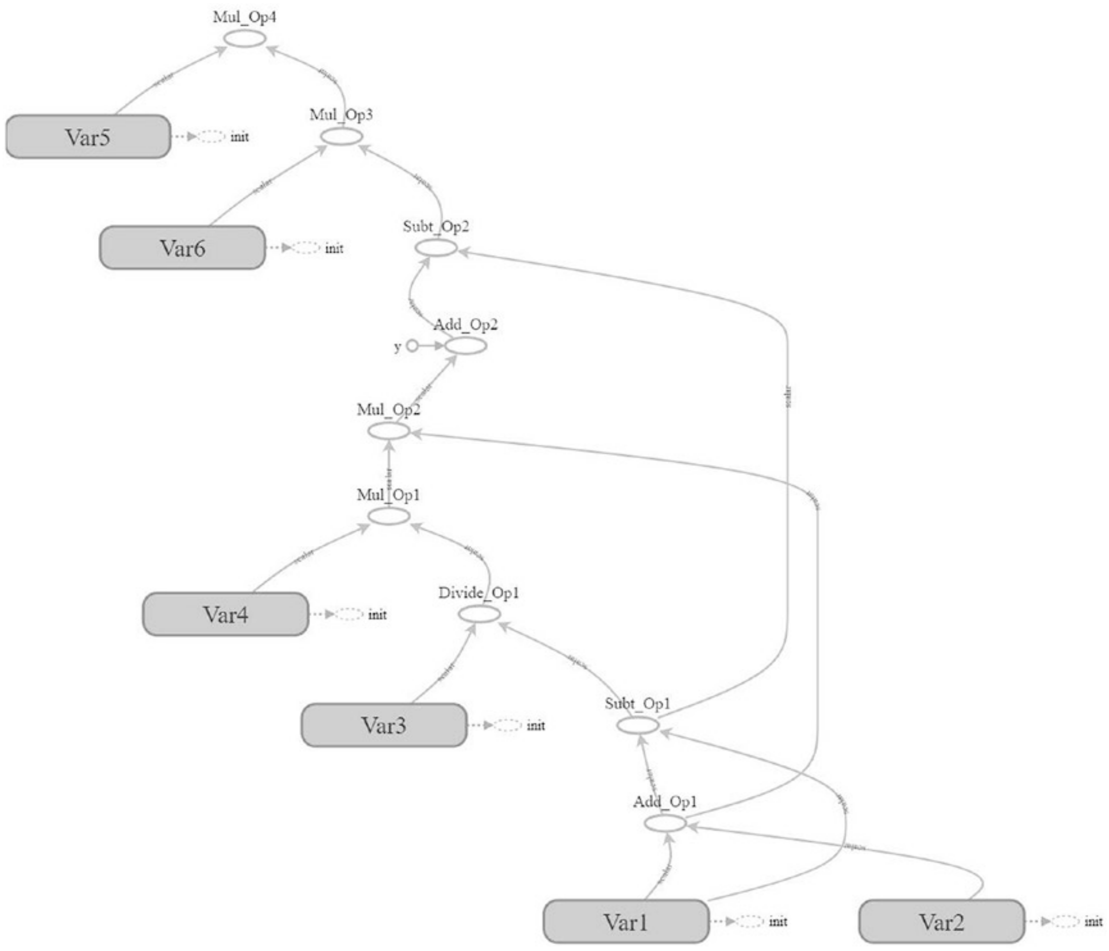

Figure 6-5. *Visualization of a dataflow graph using TB*

Linear Model

A linear model has the general form in Equation 6-1. There are n input variables x_n, and each variable is assigned a weight w_n for a total of n weights. A bias b is added to the SOP of each input and its corresponding bias.

$$y = w_1x_1 + w_2x_2 + \cdots + w_nx_n + b \qquad \text{(Equation 6-1)}$$

For a simple linear model, there are input data, weights, and biases. Which is the most suitable option between placeholders and variables to hold each of these? Generally, the placeholder is used when applying the same operation multiple times over different inputs. The inputs will be assigned for the placeholder one by one and the operation will get applied to each one. Variables are used for storing trainable parameters. As a result, the input data is to be assigned to a placeholder, but weights and biases are stored in variables. Remember to use tensorflow.global_variables_initializer() for initializing the variables.

The code that prepares the placeholder and the two variables is given in Listing 6-15. The input samples have just one input x_1 and one output y. The placeholder "data_input_placeholder" represents the input, and the placeholder "data_output_placeholder" represents the output.

Because there is only one input variable per sample, there will be a single weight w_1. The weight is represented as the "weight_variable" variable and assigned an initial value of 0.2. The bias, represented as the "bias_variable" variable, is assigned an initial value of 0.1. Note that the placeholder is assigned a value inside the tensorflow.Session.run() method using the "feed_dict" argument. The input placeholder is assigned 2.0 and the output placeholder is assigned 5.0. The visualization of the graph is in Figure 6-6.

Note that the "fetches" argument of the run() method is set to a list of three elements: "loss", "error", and "output". The "loss" tensor representing the loss function is fetched because it is the target tensor in the graph. All other tensors will be evaluated once it is evaluated. The "error" and "output" tensors are fetched just to print the prediction error in addition to the predicted output as in the print statement at the end of the code.

Note the difference between the tensors "error" and "loss". The "error" tensor calculates the square error between the predicted and desired outputs for each sample. To just summarize all errors in a single value, the tensor "loss" is used. It calculates the summation of all square errors.

Listing 6-15. Preparing Inputs, Weight, and Bias for a Linear Model

```
import tensorflow

data_input_placeholder = tensorflow.placeholder(dtype=tensorflow.float32,
name="DataInput")
data_output_placeholder = tensorflow.placeholder(dtype=tensorflow.float32,
name="DataOutput")
weight_variable = tensorflow.Variable(initial_value=0.1, dtype=tensorflow.
float32, name="Weight")
bias_variable = tensorflow.Variable(initial_value=0.2, dtype=tensorflow.
float32, name="Bias")

output = tensorflow.multiply(x=data_input_placeholder, y=weight_variable)
output = tensorflow.add(x=output, y=bias_variable)

diff = tensorflow.subtract(x=output, y=data_output_placeholder,
name="Diff")
error = tensorflow.square(x=diff, name="PredictError")
loss = tensorflow.reduce_sum(input_tensor=error, name="Loss")

with tensorflow.Session() as sess:
    writer = tensorflow.summary.FileWriter(logdir="\\AhmedGad\\
    TensorBoard\\", graph=sess.graph)
    init = tensorflow.global_variables_initializer()
    sess.run(fetches=init)
loss, predict_error, predicted_output = sess.run(fetches=[loss, error,
output], feed_dict={data_input_placeholder: 2.0,data_output_placeholder:
5.0})
    print("Loss : ", loss, "\nPredicted output : ", predicted_output,"\
    nPrediction error : ", predict_error)
    writer.close()
```

Based on the values assigned to the placeholders and the variables, the output of the print message is as follows:

```
Loss :  21.16
Predicted output :  0.4
Prediction error :  21.16
```

The predicted output is 0.4 and the desired output is 5.0. There is an error equal to 21.16. There is only one value returned in the fetched tensors because the program is working with just one sample. Also, the loss value is equal to the error value because there is just one sample. We can run the program for multiple samples.

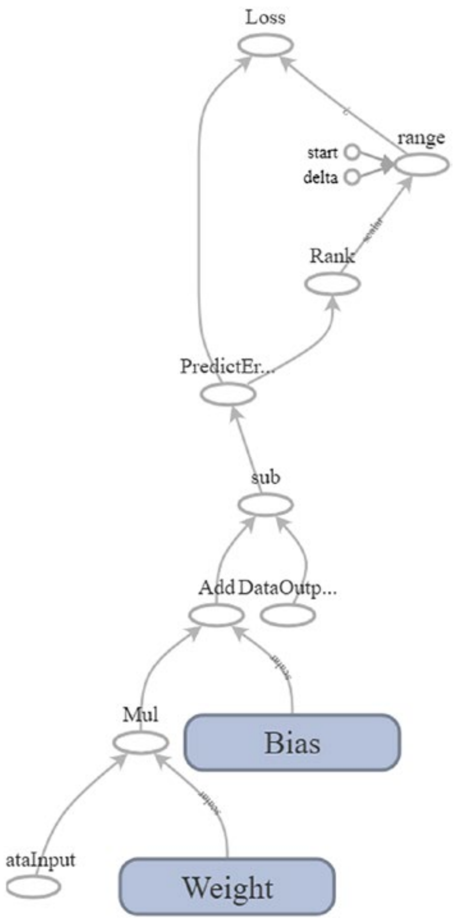

Figure 6-6. *Visualization of a dataflow graph of a linear model with one input*

Rather than assigning just a single value to the placeholder "data_input_placeholder", we can assign multiple values enclosed in a list. This also applies to the "data_output_placeholder" placeholder. Note that they must have identical shapes. The modified program after using two samples is in Listing 6-16. The print message is as follows:

```
Loss :  51.41
Predicted output :  [ 0.4  0.5]
Prediction error :  [21.16  30.25]
```

This means the prediction errors are 21.16 and 30.25 for the first and second samples, respectively. The sum of all square errors is 51.41. Because there is a high value for the loss function, we have to update the parameters (weights and bias) in order to minimize the prediction error.

Listing 6-16. Running the TF Program for Multiple Samples

```
import tensorflow

data_input_placeholder = tensorflow.placeholder(dtype=tensorflow.float32, 
name="DataInput")
data_output_placeholder = tensorflow.placeholder(dtype=tensorflow.float32, 
name="DataOutput")
weight_variable = tensorflow.Variable(initial_value=0.1, dtype=tensorflow.
float32, name="Weight")
bias_variable = tensorflow.Variable(initial_value=0.2, dtype=tensorflow.
float32, name="Bias")

output = tensorflow.multiply(x=data_input_placeholder, y=weight_variable)
output = tensorflow.add(x=output, y=bias_variable)

diff = tensorflow.subtract(x=output, y=data_output_placeholder, 
name="Diff")
error = tensorflow.square(x=diff, name="PredictError")
loss = tensorflow.reduce_sum(input_tensor=error, name="Loss")

with tensorflow.Session() as sess:
    init = tensorflow.global_variables_initializer()
    sess.run(fetches=init)
loss, predict_error, predicted_output = sess.run(fetches=[loss, error, 
output], feed_dict={data_input_placeholder: [2.0, 3.0],data_output_
placeholder: [5.0, 6.0]})
    print("Loss : ", loss, "\nPredicted output : ", predicted_output, 
    "\nPrediction error : ", predict_error)
```

Currently, there is no way to update the parameters. A number of optimizers already exist in TF for doing that job.

GD Optimizer from TF Train API

There are a number of optimizers that TF provides for optimizing model parameters automatically. GD is an example that changes the values of each parameter slowly until reaching the value that minimizes the loss. GD modifies each variable according to the magnitude of the derivative of loss with respect to the variable. This is identical to what is discussed in Chapter 3 in the backward pass of training ANN. The "tensorflow.train" API has a class called "GradientDescentOptimizer" that can both calculate the derivatives and optimize the parameters. The program after using "GradientDescentOptimizer" is in Listing 6-17.

Listing 6-17. Using GD for Optimizing the Model Parameters

```
import tensorflow

data_input_placeholder = tensorflow.placeholder(dtype=tensorflow.float32,
name="DataInput")
data_output_placeholder = tensorflow.placeholder(dtype=tensorflow.float32,
name="DataOutput")
weight_variable = tensorflow.Variable(initial_value=0.1, dtype=tensorflow.
float32, name="Weight")
bias_variable = tensorflow.Variable(initial_value=0.2, dtype=tensorflow.
float32, name="Bias")

output = tensorflow.multiply(x=data_input_placeholder, y=weight_variable,
name="Multiply")
output = tensorflow.add(x=output, y=bias_variable, name="Add")

diff = tensorflow.subtract(x=output, y=data_output_placeholder,
name="Diff")
error = tensorflow.square(x=diff, name="PredictError")
loss = tensorflow.reduce_sum(input_tensor=error, name="Loss")
train_optim = tensorflow.train.GradientDescentOptimizer(learning_rate=0.01,
name="Optimizer")
minimizer = train_optim.minimize(loss=loss, name="Minimizer")

with tensorflow.Session() as sess:
```

```
writer = tensorflow.summary.FileWriter(graph=sess.graph, logdir=
"\\AhmedGad\\TensorBoard\\")
init = tensorflow.global_variables_initializer()
sess.run(fetches=init)
for k in range(1000):
    _, data_loss, predict_error, predicted_output = sess.
    run(fetches=[minimizer,loss, error, output], feed_dict={data_input_
    placeholder: [1.0, 2.0],data_output_placeholder: [5.0, 6.0]})

print("Loss : ", data_loss,"\nPredicted output : ", predicted_output,
"\nPrediction error : ", predict_error)
writer.close()
```

The program uses a loop that iterates through 1,000 iterations. For each iteration, the current parameters are used for predicting the outputs, the loss is calculated, and the GD optimizer updates the parameters to minimize the loss. Note that the “minimize()” operation returns an operation that minimizes the loss.

After the end of the iterations, the print statement is executed. Here are its outputs:

```
Loss :  0.00323573
Predicted output :  [ 4.951612    6.02990532]
Prediction error :  [ 0.0023414   0.00089433]
```

Thanks to GD, the loss is reduced from 51.41 to just 0.0032. The graph of the previous program in Listing 6-17 is available in Figure 6-7.

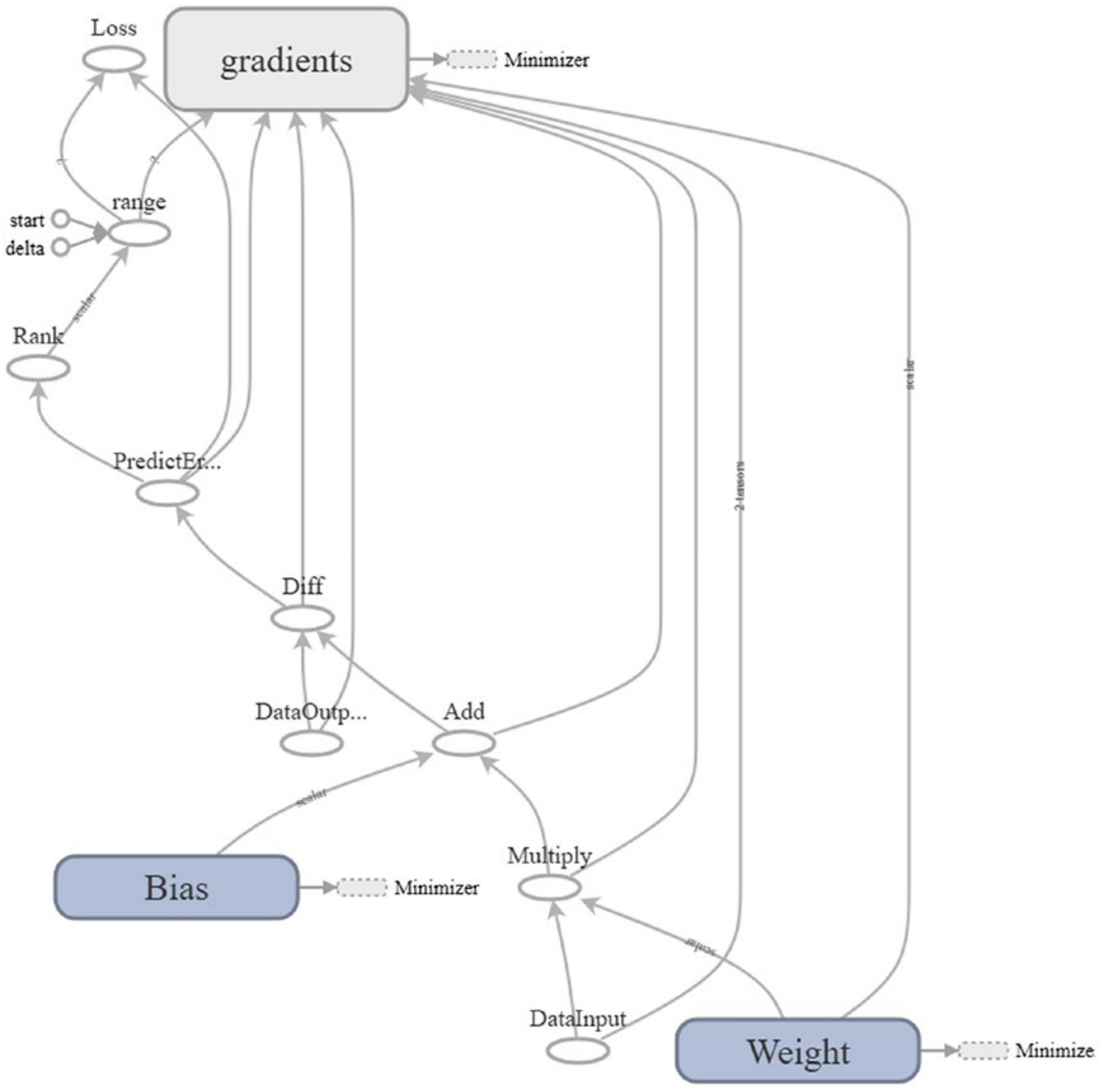

Figure 6-7. *Dataflow graph of a linear model optimized using GD*

Locating Parameters to Optimize

An important question now arises: How does the optimizer know the parameters to change their values? Let's see how it knows that.

After running the session, the "minimizer" operation will be executed. TF will follow the chain of graph nodes to evaluate such an operation. TF found that the "minimizer" operation depends on a single argument, which is the "loss" tensor. Thus, our goal is to minimize the value of such a tensor. How can we minimize that tensor? We have to follow the graph back.

The “loss” tensor is evaluated using the “tensorflow.reduce_sum()” operation. As a result, our goal is to minimize the result of the “tensorflow.reduce_sum()” operation.

Stepping back, this operation is evaluated using the “error” tensor. As a result, our goal now is to minimize the “error” tensor. Stepping back again, we find that the “error” tensor depends on the “tensorflow.square()” operation. As a result, we have to minimize “tensorflow.square()” the operation. The input tensor to this operation is the “diff” tensor. Thus, our goal is to minimize the “diff” tensor. Because the “diff” tensor is the result of the “tensorflow.subtract()” operation, then our goal is to minimize this operation.

Minimizing the “tensorflow.subtract()” asks us to minimize its input tensors, which are “output” and “data_output_placeholder”. Looking at these two tensors, which one can be modified? Only the variable tensors can be modified. Because “data_output_placeholder” is not a variable but a placeholder, we can’t modify it. Thus, we have only the “output” tensor to minimize in order to minimize the result.

The “output” tensor is calculated according to Equation 6-1. It has three inputs: input, weight, and bias, which are represented by the tensors “data_input_placeholder”, “weight_variable”, and “bias_variable”, respectively. Looking for these three tensors, only “weight_variable” and “bias_variable” can be changed because they are variables. Thus, finally we know that our goal is to minimize “weight_variable” and “bias_variable” tensors.

In order to minimize the “tensorflow.train.GradientDescentOptimizer.minimize()” operation, we have to change the values of the “weight_variable” and “bias_variable” tensors. This is how TF deduced that to minimize the loss it should minimize the weight and bias parameters.

Building FFNN

In this section, two basic feed-forward ANNs (FFNNs) will be created for classification using TF Core API. We will follow the same steps used previously to build an ANN using NumPy but with changes.

The summarized steps are as follows:

1. Reading the training data (inputs and outputs).
2. Building the neural network layers and preparing their parameters (weights, biases, and activation functions).
3. Building a loss function to assess the prediction error.
4. Create a training loop for training the network and updating its parameters.
5. Assessing the accuracy of the trained ANN using new unseen test data.

We will start by building a single-layer FFANN.

Linear Classification

Table 6-2 gives the data of the first classification problem. It is a binary classification problem to classify the RGB colors into either red or blue based on the color channels red, green, and blue.

***Table 6-2.** RGB Color Classification Problem*

Class	Red	Green	Blue
Red	255	0	0
	248	80	68
Blue	0	9	255
	67	15	210

According to Listing 6-18, two placeholders ("training_inputs" and "training_outputs") are created for holding the training data inputs and outputs. Their data type is set to "float32" but they do not have a specific shape. The shape of the "training_inputs" placeholder is N×3. What does that mean?

Regularly, placeholders are used to hold the training data of the model. The size of the training data is not always fixed. There might be a change in the number of samples, the number of features, or both. For example, we might train a model with 100 samples, where each sample is represented by 15 features. The shape of the placeholder, in this case, is 100×15. Assume that we later decided to change the number of training samples to be 50. The shape of the placeholder must get changed to be 50×15.

Listing 6-18. Placeholders for the Training Data Inputs and Outputs

```
import tensorflow
training_inputs = tensorflow.placeholder(shape=[None, 3], dtype=tensorflow.
float32)
training_outputs = tensorflow.placeholder(shape=[None, 1],
dtype=tensorflow.float32)
```

To make life easier, TF supports creating placeholders of variable shape. The placeholder shape is determined based on the data assigned to it. The shape might be variable across all dimensions or for just some dimensions. If we decided to use 30 features but had not decided on the number of training samples, then the shape is N×15, where N is the number of samples. Feeding 20 samples to the placeholder, N will be set to 20. This is the case for the two placeholders in Listing 6-18. To leave the placeholder generic for holding any number of training samples, its shape is set to (None, 3). None means that this dimension (representing the number of samples) does not have a static size.

After preparing the inputs and the outputs, the next step is to decide the network architecture for preparing their parameters (weights and bias). Because the data is simple, we could plot it. Listing 6-19 gives the code used to plot the data. Note that the data has three dimensions, and thus the plot is 3D as in Figure 6-8.

Listing 6-19. 3D Scatter Plot of the Training Data

```
import matplotlib.pyplot
import mpl_toolkits.mplot3d

figure3D = matplotlib.pyplot.figure()
axis3D = mpl_toolkits.mplot3d.Axes3D(figure3D)

red = [255, 248, 0, 67]
green = [0, 80, 9, 15]
blue = [0, 68, 255, 210]

axis3D.scatter(red, green, blue, color="black")
axis3D.set_xlabel(xlabel="Red")
```

```
axis3D.set_ylabel(ylabel="Green")
axis3D.set_zlabel(zlabel="Blue")
matplotlib.pyplot.show()
```

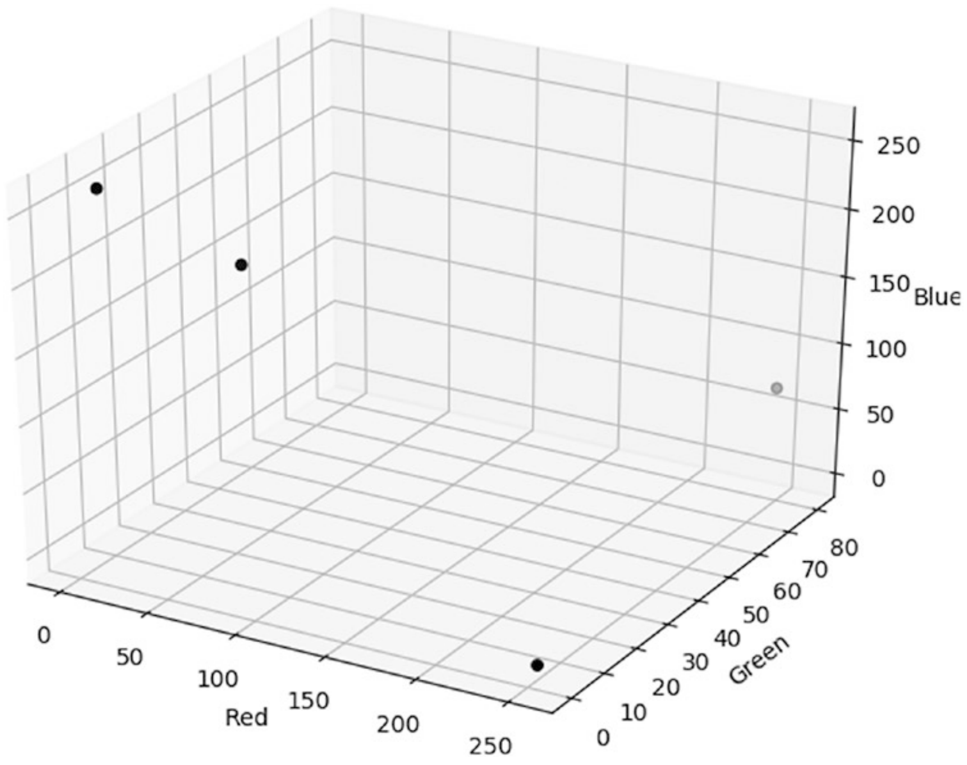

Figure 6-8. *3D scatter plot of the training data*

Based on Figure 6-8, it is obvious that the two classes can be separated linearly. The two samples of the class red are located on the right of the plot, and the blue samples are on the left. Knowing it is a linear problem guides us to not use any hidden layer. Thus, the network architecture will just have input and output layers. Because each sample is represented using three features, then the input layer will have just three inputs, one for each feature. The network architecture is in Figure 6-9, where $X_0 = 1.0$ is the bias input and W_0 is the bias. W_1, W_2, and W_3 are the weights for the three inputs R (Red), G (Green), and B (Blue).

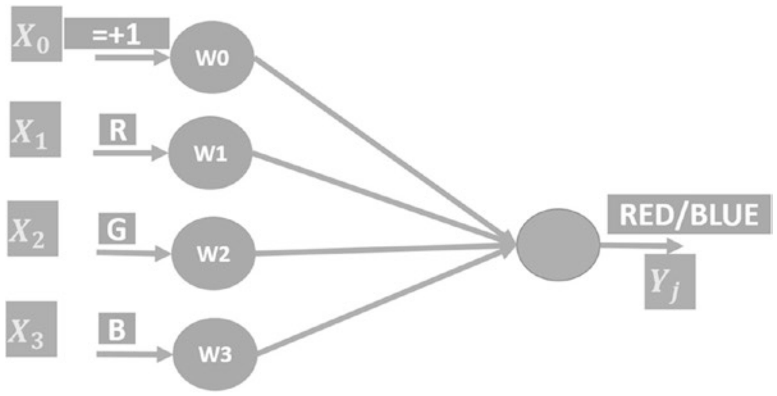

Figure 6-9. *ANN architecture for classifying RGB colors linearly*

Listing 6-20 prepares the variables holding these parameters. Because there are three inputs and each input has a weight, the shape of the weights is 3×1 according to the "weights" variable. The shape is 3×1 to make matrix multiplication between the inputs and the weights valid. The input data of shape N×3 could be multiplied by the weights of shape 3×1, and the result will be N×1. There is just one bias according to the "bias" variable.

Listing 6-20. Preparing ANN Parameter Variables

```
import tensorflow
weights = tensorflow.Variable(initial_value=[[0.003], [0.001], [0.008]],
dtype=tensorflow.float32)
bias = tensorflow.Variable(initial_value=[0.001], dtype=tensorflow.float32)
```

After preparing the data, network architecture, and the parameters, next is to feed the training input data into the network, predict their outputs, and calculate the loss according to Listing 6-21. The input data matrix is multiplied by the weights vector using the "matmul()" operation and the result is stored in the "sop" tensor. According to Equation 6-1, the result of the multiplication is added to the bias. The result of the addition is stored in the "sop_bias" tensor. The result is then applied to the sigmoid function defined by the "tensorflow.nn.sigmoid()" operation and returned into the "predictions" tensor.

Listing 6-21. Using the Network Parameters to Predict the Outputs of the Training Data

```
import tensorflow
sop = tensorflow.matmul(a=training_inputs, b=weights, name="SOPs")
sop_bias = tensorflow.add(x=sop, y=bias)
predictions = tensorflow.nn.sigmoid(x=sop_bias, name="Sigmoid")

error = tensorflow.subtract(x=training_outputs, y=predictions,
name="Error")
square_error = tensorflow.square(x=error, name="SquareError")
loss = tensorflow.reduce_sum(square_error, name="Loss")

train_optim = tensorflow.train.GradientDescentOptimizer(learning_rate=0.05,
name="GradientDescent")
minimizer = train_op.minimize(loss, name="Minimizer")
```

After predicting the outputs, next is to measure the loss. At first, the difference between the predicted and the correct outputs are calculated using the "subtract()" operation, and the result is stored in the "error" tensor. The square of that error is then calculated using the "square" tensor and the result is stored into the "square_error" tensor. Finally, the squared errors are reduced into a single value by summing them all. The result is stored into the "loss" tensor.

The loss is calculated to learn how we far we currently are from the optimal results where the loss is 0. Based on the loss, the GD optimizer is initialized in the "train_optim" tensor to update the network parameters in order to minimize the loss. The update operation is returned into the "minimizer" tensor.

Up to this point, the network architecture is complete and ready for training using the input and output data. Two Python lists are created in Listing 6-22 to hold the training data inputs and outputs. Note that the red class label is "1.0" and the blue one is "0.0". The lists are assigned to the placeholders "training_inputs" and "training_outputs" using the "feed_dict" argument inside the "tensorflow.Session.run()" operation. Note that the target of execution is the "minimizer" operation. The session goes through a number of iterations to update the ANN parameters.

Listing 6-22. Training Data Inputs and Outputs

```
training_inputs_data = [[255, 0, 0],
                        [248, 80, 68],
                        [0, 0, 255],
                        [67, 15, 210]]
training_outputs_data = [[1.0],
                         [1.0],
                         [0.0],
                         [0.0]]

with tensorflow.Session() as sess:
    init = tensorflow.global_variables_initializer()
    sess.run(init)

    for step in range(10):
        sess.run(fetches=minimizer, feed_dict={training_inputs: training_
        inputs_data, training_outputs: training_outputs_data})
```

The complete code for building a single-layer ANN for classifying the two-class problem in Table 6-2 is in Listing 6-23.

Listing 6-23. The Complete Code for Classifying the Two-Class RGB Color Problem

```
import tensorflow

# Preparing a placeholder for the training data inputs of shape (N, 3)
training_inputs = tensorflow.placeholder(shape=[None, 3], dtype=tensorflow.
float32, name="Inputs")

# Preparing a placeholder for the training data outputs of shape (N, 1)
training_outputs = tensorflow.placeholder(shape=[None, 1],
dtype=tensorflow.float32, name="Outputs")

# Initializing neural network weights of shape (3, 1)
weights = tensorflow.Variable(initial_value=[[0.003], [0.001], [0.008]],
dtype=tensorflow.float32, name="Weights")
```

```
# Initializing the ANN bias
bias = tensorflow.Variable(initial_value=[0.001], dtype=tensorflow.float32,
name="Bias")

# Calculating the SOPs by multiplying the weights matrix by the data inputs
matrix
sop = tensorflow.matmul(a=training_inputs, b=weights, name="SOPs")

# Adding the bias to the SOPs
sop_bias = tensorflow.add(x=sop, y=bias, name="AddBias")

# Sigmoid activation function of the output layer neuron
predictions = tensorflow.nn.sigmoid(x=sop_bias, name="Sigmoid")

# Calculating the difference (error) between the ANN predictions and the
correct outputs
error = tensorflow.subtract(x=training_outputs, y=predictions,
name="Error")

# Square error.
square_error = tensorflow.square(x=error, name="SquareError")

# Measuring the prediction error of the network after being trained
loss = tensorflow.reduce_sum(square_error, name="Loss")

# Minimizing the prediction error using gradient descent optimizer
train_optim = tensorflow.train.GradientDescentOptimizer(learning_rate=0.05,
name="GradientDescent")
minimizer = train_optim.minimize(loss, name="Minimizer")

# Training data inputs of shape (N, 3)
training_inputs_data = [[255, 0, 0],
                        [248, 80, 68],
                        [0, 0, 255],
                        [67, 15, 210]]

# Training data desired outputs
training_outputs_data = [[1.0],
                         [1.0],
                         [0.0],
                         [0.0]]
```

```
# Creating a TensorFlow Session
with tensorflow.Session() as sess:
    writer = tensorflow.summary.FileWriter(logdir="\\AhmedGad\\
    TensorBoard\\", graph=sess.graph)
    # Initializing the TensorFlow Variables (weights and bias)
    init = tensorflow.global_variables_initializer()
    sess.run(init)

    # Training loop of the neural network
    for step in range(10):
        sess.run(fetches=minimizer, feed_dict={training_inputs: training_
        inputs_data, training_outputs: training_outputs_data})

    # Class scores of training data
    print("Expected Outputs for Train Data:\n", sess.
    run(fetches=[predictions, weights, bias], feed_dict={training_inputs:
    training_inputs_data}))

    # Class scores of new test data
    print("Expected Outputs for Test Data:\n", sess.
    run(fetches=predictions, feed_dict={training_inputs: [[230, 60, 76],
    [93, 52, 180]]}))
    writer.close()
```

After all training iterations, the trained network is used to predict the output of both the training samples and the two other unseen test samples. The following is the output of the print statements at the end of Listing 6-23. The network is able to predict all training and test samples correctly.

```
Expected Outputs for Train Data:
 [[ 1.]
 [ 1.]
 [ 0.]
 [ 0.]]
Expected Outputs for Test Data:
 [[ 1.]
 [ 0.]]
```

The weighs and bias after training the network are as follows:

```
Weights:[[1.90823114], [0.11530305], [-4.13670015]],
Bias: [-0.00771546].
```

Figure 6-10 visualizes the graph created in Listing 6-23.

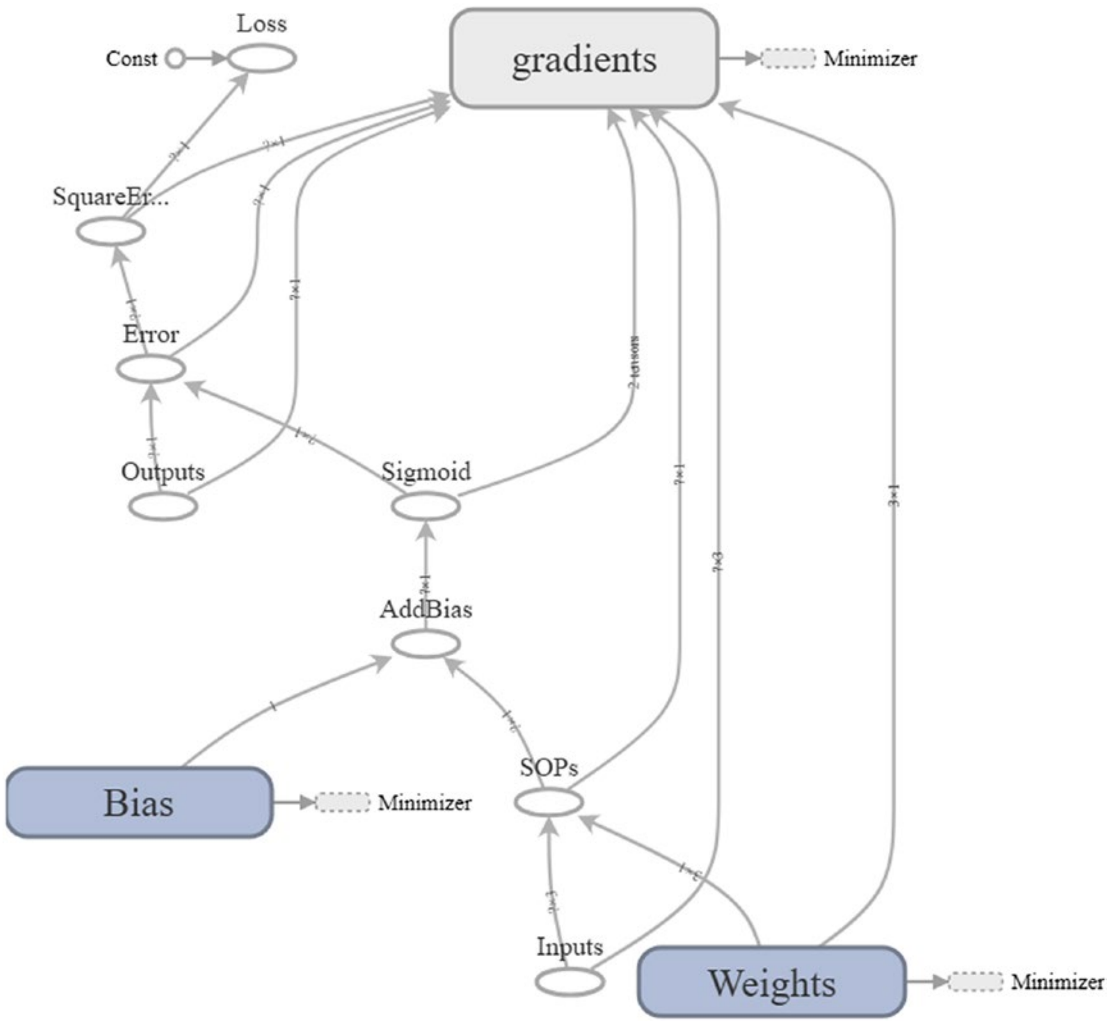

Figure 6-10. *Graph of ANN with a single layer*

Nonlinear Classification

Now we are going to build an ANN that simulates the operation of an XOR gate with two inputs. The truth table for the problem is in Table 6-3. Because the problem is simple, we can plot it as in Figure 6-11 to know whether the classes are linearly or nonlinearly separable.

Table 6-3. *The Truth Table of Two-Input XOR Gate*

Output	A	B
1	1	0
	0	1
0	0	0
	1	1

Based on the graph, it is obvious that the classes are nonlinearly separable. Thus, we have to use hidden layers. According to the first example in section **Designing ANN** of Chapter 3, we know that just a single hidden layer with two neurons is sufficient.

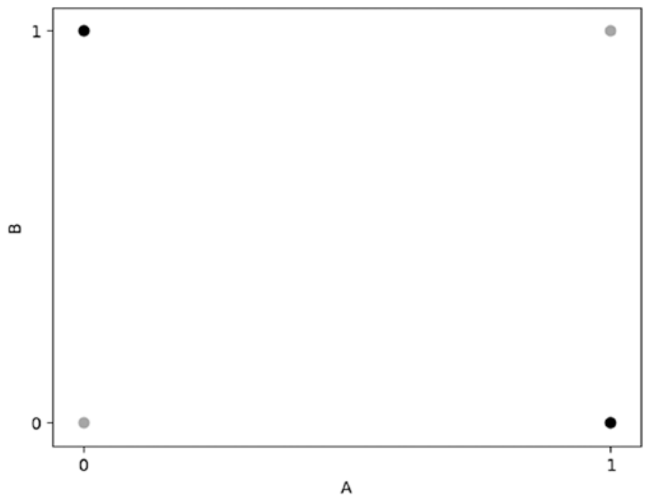

Figure 6-11. *Graph of two-input XOR gate*

The network architecture is in Figure 6-12. That hidden layer accepts the inputs from the input layer. Based on its weights and biases, its two activation functions will produce two outputs. The outputs of the hidden layer will be regarded as the inputs to the output layer. Using its activation function, the output layer produces the final expected class of the input sample.

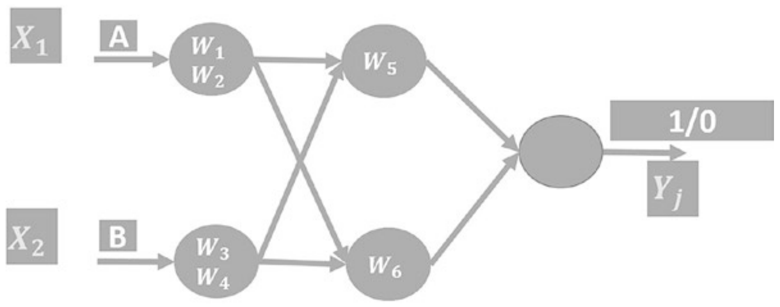

Figure 6-12. *Network architecture for XOR gate with two inputs*

The complete code is in Listing 6-24. There are some changes compared to the previous example. The initial parameters are randomly generated using "tensorflow.truncated_normal()" operation. The output tensor of the hidden layer "hidden_sigmoid" is used as input to the output layer. The output tensor of the output layer is the predicted outputs. The remaining code is similar to the previous example.

Listing 6-24. The Complete Code for ANN Simulating XOR Gate with Two Inputs

```
import tensorflow

# Preparing a placeholder for the training data inputs of shape (N, 3)
training_inputs = tensorflow.placeholder(shape=[4, 2], dtype=tensorflow.
float32, name="Inputs")

# Preparing a placeholder for the training data outputs of shape (N, 1)
training_outputs = tensorflow.placeholder(shape=[4, 1], dtype=tensorflow.
float32, name="Outputs")

# Initializing the weights of the hidden layer of shape (2, 2)
hidden_weights = tensorflow.Variable(initial_value=tensorflow.truncated_
normal(shape=(2,2), name="HiddenRandomWeights"), dtype=tensorflow.float32,
name="HiddenWeights")
```

```
# Initializing the bias of the hidden layer of shape (1,2)
hidden_bias = tensorflow.Variable(initial_value=tensorflow.truncated_
normal(shape=(1,2), name="HiddenRandomBias"), dtype=tensorflow.float32,
name="HiddenBias")

# Calculating the SOPs by multiplying the weights matrix of the hidden
layer by the data inputs matrix
hidden_sop = tensorflow.matmul(a=training_inputs, b=hidden_weights,
name="HiddenSOPs")

# Adding the bias to the SOPs of the hidden layer
hidden_sop_bias = tensorflow.add(x=hidden_sop, y=hidden_bias,
name="HiddenAddBias")

# Sigmoid activation function of the hidden layer outputs
hidden_sigmoid = tensorflow.nn.sigmoid(x=hidden_sop_bias,
name="HiddenSigmoid")

# Initializing the weights of the output layer of shape (2, 1)
output_weights = tensorflow.Variable(initial_value=tensorflow.truncated_
normal(shape=(2,1), name="OutputRandomWeights"), dtype=tensorflow.float32,
name="OutputWeights")

# Initializing the bias of the output layer of shape (1,1)
output_bias = tensorflow.Variable(initial_value=tensorflow.truncated_
normal(shape=(1,1), name="OutputRandomBias"), dtype=tensorflow.float32,
name="OutputBias")

# Calculating the SOPs by multiplying the weights matrix of the hidden
layer by the outputs of the hidden layer
output_sop = tensorflow.matmul(a=hidden_sigmoid, b=output_weights,
name="Output_SOPs")

# Adding the bias to the SOPs of the hidden layer
output_sop_bias = tensorflow.add(x=output_sop, y=output_bias,
name="OutputAddBias")

# Sigmoid activation function of the output layer outputs. These are the
predictions.
```

```
predictions = tensorflow.nn.sigmoid(x=output_sop_bias,
name="OutputSigmoid")

# Calculating the difference (error) between the ANN predictions and the
correct outputs
error = tensorflow.subtract(x=training_outputs, y=predictions,
name="Error")

# Square error.
square_error = tensorflow.square(x=error, name="SquareError")

# Measuring the prediction error of the network after being trained
loss = tensorflow.reduce_sum(square_error, name="Loss")

# Minimizing the prediction error using gradient descent optimizer
train_optim = tensorflow.train.GradientDescentOptimizer(learning_rate=0.01,
name="GradientDescent")
minimizer = train_optim.minimize(loss, name="Minimizer")

# Training data inputs of shape (4, 2)
training_inputs_data = [[1, 0],
                        [0, 1],
                        [0, 0],
                        [1, 1]]

# Training data desired outputs
training_outputs_data = [[1.0],
                         [1.0],
                         [0.0],
                         [0.0]]

# Creating a TensorFlow Session
with tensorflow.Session() as sess:
    writer = tensorflow.summary.FileWriter(logdir="\\AhmedGad\\
   TensorBoard\\", graph=sess.graph)
    # Initializing the TensorFlow Variables (weights and bias)
    init = tensorflow.global_variables_initializer()
    sess.run(init)
```

```
# Training loop of the neural network
for step in range(100000):
    print(sess.run(fetches=minimizer, feed_dict={training_inputs:
    training_inputs_data, training_outputs: training_outputs_data}))

# Class scores of training data
print("Expected Outputs for Train Data:\n", sess.
run(fetches=[predictions, hidden_weights, output_weights, hidden_bias,
output_bias], feed_dict={training_inputs: training_inputs_data}))

writer.close()
```

After completing the training process, the samples are correctly classified. Here are the predicted outputs:

```
[[0.96982265],
 [0.96998841],
 [0.0275135],
 [0.0380362]]
```

The parameters of the network after training are as follows:

- Hidden layer weights: [–6.27943468, –4.30125761], [–6.38489389, –4.31706429]]
- Hidden layer bias: [[–8.8601017], [8.70441246]]
- Output layer weights: [[2.49879336, 6.37831974]]
- Output layer bias: [[–4.06760359]]

Figure 6-13 visualizes the graph of Listing 6-24.

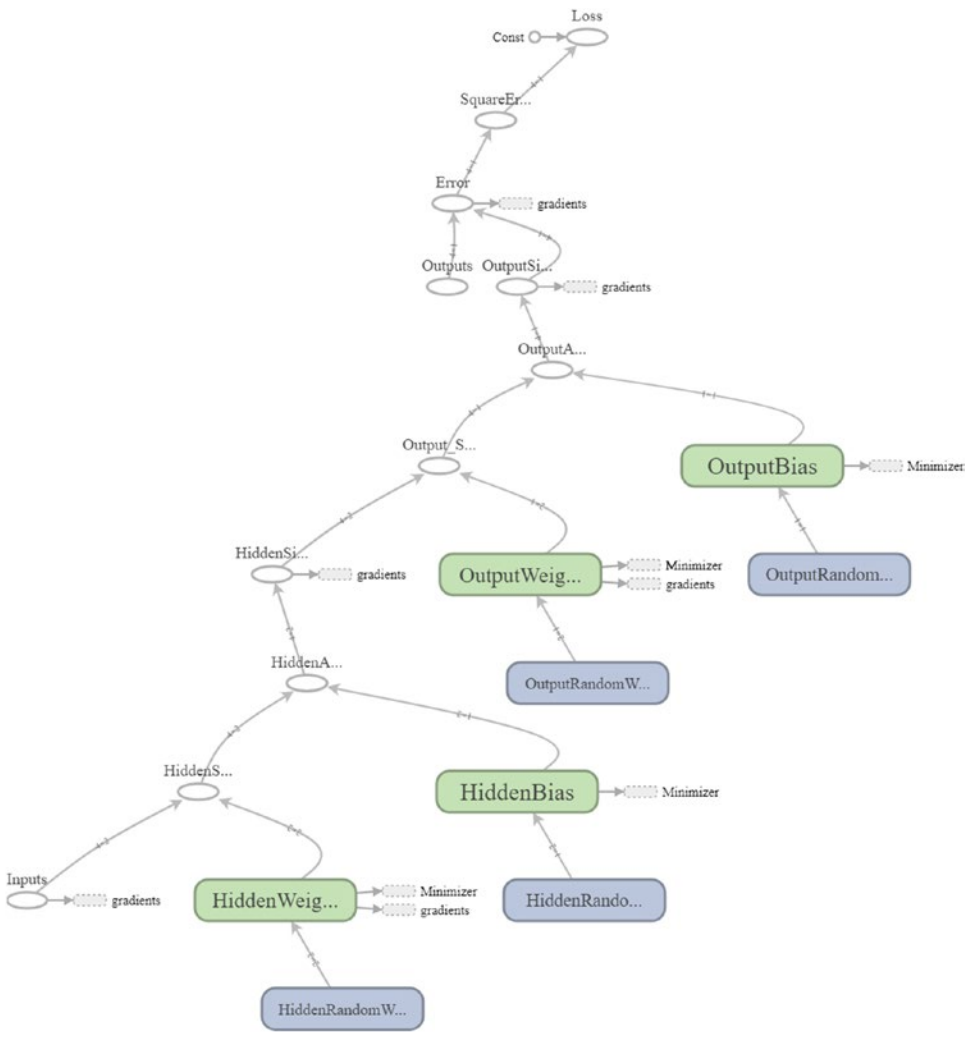

Figure 6-13. *Graph of ANN simulating XOR gate with two inputs*

CIFAR10 Recognition Using CNN

The previous examples we discussed help us learn the basics of TF and build good knowledge. This section extends this knowledge by using TF to build a CNN to recognize images from the CIFAR10 dataset.

Preparing Training Data

The binary data of the CIFAR10 dataset is available for download for Python from this page: www.cs.toronto.edu/~kriz/cifar.html. The dataset has 60,000 images split into training and testing sets. There are five binary files containing the training data, where each file has 10,000 images. The images are RGB of size 32×32×3. The training files are named “data_batch_1”, “data_batch_2”, and so on. There is a single file for the test data named “test_batch” with 10,000 images. A metadata file named “batches.meta” is available, giving details about the dataset such as the class labels, which are airplane, automobile, bird, cat, deer, dog, frog, horse, ship, and truck.

Because each file in the dataset is binary, we have to decode it in order to retrieve the actual image data. To do this job, a function called “unpickle_patch” is created, as defined in Listing 6-25.

Listing 6-25. Decoding the CIFAR10 Binary Data

```
def unpickle_patch(file):
    patch_bin_file = open(file, 'rb')#Reading the binary file.
    patch_dict = pickle.load(patch_bin_file, encoding='bytes')#Loading the
    details of the binary file into a dictionary.
    return patch_dict#Returning the dictionary.
```

The method accepts the binary file path and returns the details about this file into the “patch_dict” dictionary. The dictionary has the image data for all 10,000 samples within the file in addition to their class labels.

There are five training data files. In order to decode the entire training data, a new function called “get_dataset_images” is created as in Listing 6-26. That function accepts the dataset path and decodes the data of just the five training files. Firstly, it lists all files under the dataset directory using the “os.listdir()” function. All file names are returned into the “files_names” list.

Because all train and test files are located within the same directory, this function filters the files under this path to just return the training files. The function uses an “if” statement to just return files starting with “data_batch_” as it is discriminative to the train file names. Note that the test data is prepared later after building and training the CNN.

Listing 6-26. Decoding All Training Files

```
def get_dataset_images(dataset_path, im_dim=32, num_channels=3):
    num_files = 5#Number of training binary files in the CIFAR10 dataset.
    images_per_file = 10000#Number of samples within each binary file.
    files_names = os.listdir(patches_dir)#Listing the binary files in the
    dataset path.

    dataset_array = numpy.zeros(shape=(num_files * images_per_file, im_dim,
    im_dim, num_channels))
    dataset_labels = numpy.zeros(shape=(num_files * images_per_file),
    dtype=numpy.uint8)

    index = 0#Index variable to count number of training binary files being
    processed.
    for file_name in files_names:
        if file_name[0:len(file_name) - 1] == "data_batch_":
            print("Working on : ", file_name)
            data_dict = unpickle_patch(dataset_path+file_name)

            images_data = data_dict[b"data"]
            #Reshaping all samples in the current binary file to be of
            32x32x3 shape.
            images_data_reshaped = numpy.reshape(images_data,
            newshape=(len(images_data), im_dim, im_dim, num_channels))
            #Appending the data of the current file after being reshaped.
            dataset_array[index * images_per_file:(index + 1) * images_per_
            file, :, :, :] = images_data_reshaped
            #Appending the labels of the current file.
            dataset_labels[index * images_per_file:(index + 1) * images_
            per_file] = data_dict[b"labels"]
            index = index + 1#Incrementing the counter of the processed
            training files by 1 to accept new file.
    return dataset_array, dataset_labels#Returning the training input data
    and output labels.
```

Each training file is decoded by calling the “unpickle_patch” function, and its image data and their labels are returned into the “data_dict” dictionary. There are five training files, and thus there are five classes to such a function, where each call returns a dictionary.

Based on the dictionary returned by this function, the “get_dataset_images” function concatenates the details (image data and class labels) of all files into a NumPy array. The image data could be retrieved from that dictionary using the “data” key and stored into the “dataset_array” NumPy array, which stores all decoded images across all training files. Class labels are retrieved using the “labels” key and returned into the “dataset_labels” NumPy array, which stores all labels across all images in the training data. The “dataset_array” and “dataset_labels” are returned by the function.

When decoded, the data of each image returns as a 1D vector of length 32×32×3=3,072 pixels. This vector should be reshaped of the original shape with three dimensions. This is because CNN layers created in TF accepts the images of this shape. For this reason, the “get_dataset_images” function has arguments for accepting the size of each dimension for the dataset images. The first one is “im_dim” representing the number of rows/columns (they are equal) in addition to the “num_channels” representing the number of channels.

After preparing the training data, we can build and train the CNN model using TF.

Building the CNN

The dataflow graph of the CNN is created inside a function called “create_CNN” as in Listing 6-27. It creates a stack of convolution (conv), ReLU, max pooling, dropout, and FC layers. The architecture of the CNN is illustrated in Figure 6-14. It has three conv-relu-pool groups followed by a dropout layer and finally two FC layers.

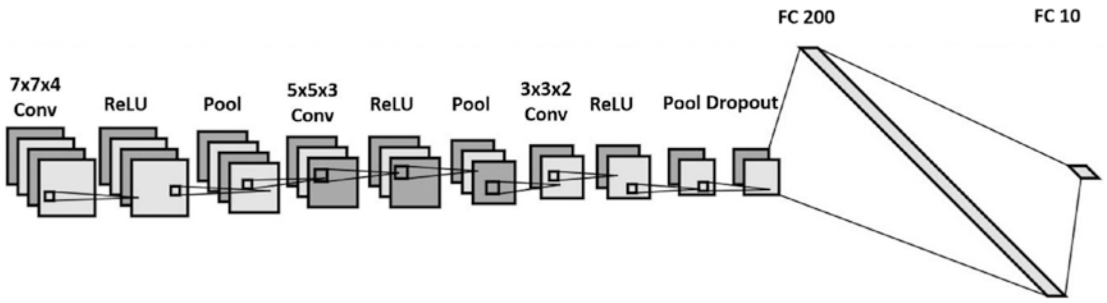

Figure 6-14. *CNN architecture*

The function returns the results of the last FC layer. As regularly, the output of each layer is the input to the next layer. This requires consistency between the sizes of the outputs and inputs of neighboring layers. Note that for each conv, ReLU, and max pooling layer, there are some parameters to get specified, such as strides across each dimension and padding.

Listing 6-27. Building the CNN Structure

```
def create_CNN(input_data, num_classes, keep_prop):
    filters1, conv_layer1 = create_conv_layer(input_data=input_data,
    filter_size=7, num_filters=4)
    relu_layer1 = tensorflow.nn.relu(conv_layer1)
    max_pooling_layer1 = tensorflow.nn.max_pool(value=relu_layer1,
                                                ksize=[1, 2, 2, 1],
                                                strides=[1, 1, 1, 1],
                                                padding="VALID")

    filters2, conv_layer2 = create_conv_layer(input_data=max_pooling_
    layer1, filter_size=5, num_filters=3)
    relu_layer2 = tensorflow.nn.relu(conv_layer2)
    max_pooling_layer2 = tensorflow.nn.max_pool(value=relu_layer2,
                                                ksize=[1, 2, 2, 1],
                                                strides=[1, 1, 1, 1],
                                                padding="VALID")

    filters3, conv_layer3 = create_conv_layer(input_data=max_pooling_
    layer2, filter_size=3, num_filters=2)
    relu_layer3 = tensorflow.nn.relu(conv_layer3)
    max_pooling_layer3 = tensorflow.nn.max_pool(value=relu_layer3,
                                                ksize=[1, 2, 2, 1],
                                                strides=[1, 1, 1, 1],
                                                padding="VALID")

    flattened_layer = dropout_flatten_layer(previous_layer=max_pooling_
    layer3, keep_prop=keep_prop)
    fc_result1 = fc_layer(flattened_layer=flattened_layer, num_
    inputs=flattened_layer.get_shape()[1:].num_elements(),
                          num_outputs=200)

    fc_result2 = fc_layer(flattened_layer=fc_result1, num_inputs=fc_
    result1.get_shape()[1:].num_elements(),
                          num_outputs=num_classes)
    print("Fully connected layer results : ", fc_result2)
    return fc_result2#Returning the result of the last FC layer.
```

The first layer in the CNN works directly on the input data. Thus, the "create_CNN" function accepts the input data as an input argument called "input_data". This data is what returned by the "get_dataset_images" function. The first layer is a convolution layer, which is created using the "create_conv_layer" function according to Listing 6-28.

The "create_conv_layer" function accepts the input data, filter size, and the number of filters. It returns the result of convolving the input data with the set of filters. The filters in the set have their depth set according to the number of channels of the input data. Because the number of channels is the last element in a NumPy array, index –1 is used to return the number of channels. The set of filters are returned into the "filters" variable.

Listing 6-28. Building Convolution Layer

```
def create_conv_layer(input_data, filter_size, num_filters):
    filters = tensorflow.Variable(tensorflow.truncated_
    normal(shape=(filter_size, filter_size, tensorflow.cast(input_data.
    shape[-1], dtype=tensorflow.int32), num_filters), stddev=0.05))

    conv_layer = tensorflow.nn.conv2d(input=input_data,
                                      filter=filters,
                                      strides=[1, 1, 1, 1],
                                      padding="VALID")
    return filters, conv_layer#Returning the filters and the convolution
    layer result.
```

The convolution layer is built by specifying the input data, filters, and strides along each of the four dimensions, and the padding to the "tensorflow.nn.conv2D" operation. A padding value of "VALID" means that some borders of the input image will be lost in the result, based on the filter size.

The result of any conv layer is fed into a ReLU layer created using the "tensorflow.nn.relu" operation. It accepts the conv layer output and returns a tensor of the same number of features after applying the ReLU activation function. Remember that activation functions help to create a nonlinear relationship between the inputs and the outputs. The result of the ReLU layer is then fed to a max pooling layer created using the "tensorflow.nn.max_pool" operation. Remember that the goal of pooling layers is to make the recognition translation invariant.

The "create_CNN" function accepts an argument named "keep_prop" representing the probability of keeping neurons in the dropout layer, which helps to avoid overfitting. The dropout layer is implemented using the "dropout_flatten_layer" function, as in Listing 6-29. This function returns a flattened array that is used as the input to the FC layers.

Listing 6-29. Building Dropout Layer

```
def dropout_flatten_layer(previous_layer, keep_prop):
    dropout = tensorflow.nn.dropout(x=previous_layer, keep_prob=keep_prop)
    num_features = dropout.get_shape()[1:].num_elements()
    layer = tensorflow.reshape(previous_layer, shape=(-1, num_
    features))#Flattening the results.
    return layer
```

Because the last FC layer should have a number of output neurons equal to the number of dataset classes, the number of dataset classes is used as another input argument named "num_classes" to the "create_CNN" function. The FC layer is created using the "fc_layer" function, defined according to Listing 6-30. This function accepts the flattened result of the dropout layer, the number of features in the flattened result, and the number of output neurons from the FC layer. Based on the number of inputs and outputs, a tensor named "fc_weights" represents the weights for the FC layer that is created. It gets multiplied by the flattened layer to get the returned result of the FC layer.

Listing 6-30. Building FC Layer

```
def fc_layer(flattened_layer, num_inputs, num_outputs):
    fc_weights = tensorflow.Variable(tensorflow.truncated_
    normal(shape=(num_inputs, num_outputs), stddev=0.05))
```

```
fc_result1 = tensorflow.matmul(flattened_layer, fc_weights)
return fc_result1#Output of the FC layer (result of matrix
multiplication).
```

The computational graph after being visualized using TB is shown in Figure 6-15. Part a gives the architecture of the CNN until the final max pooling layer, while part b shows the remaining steps.

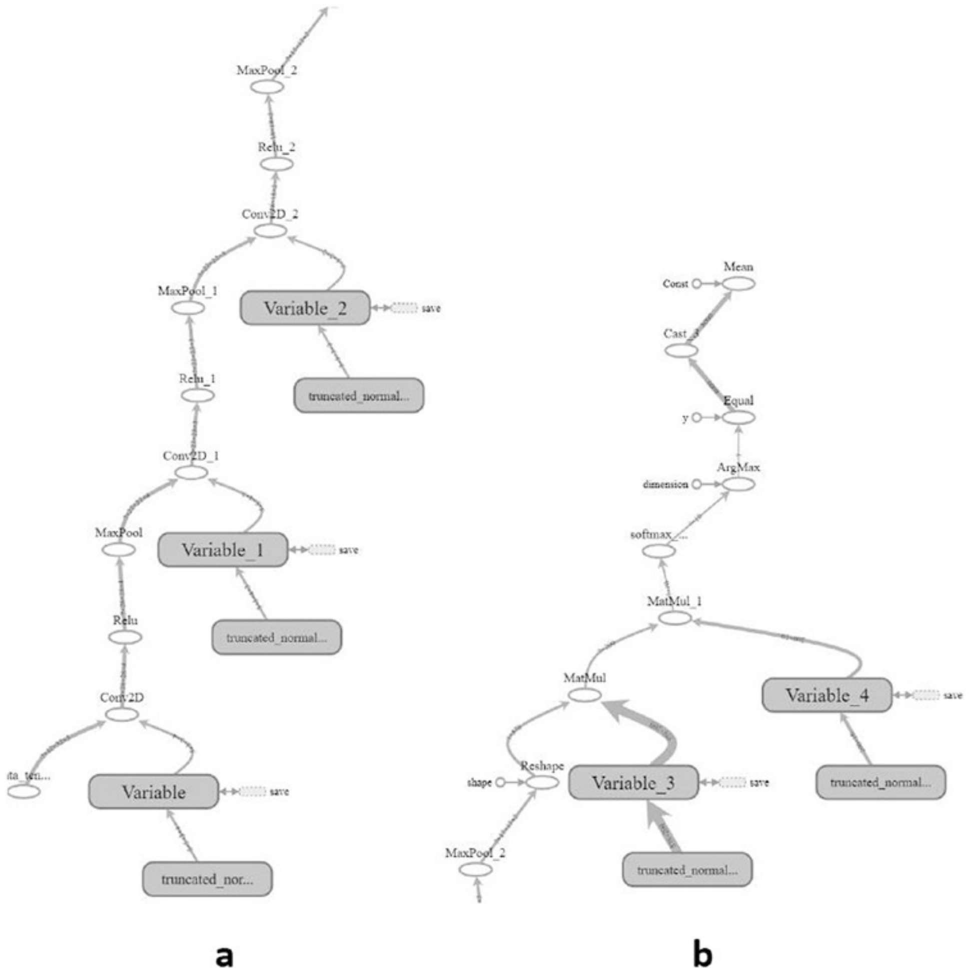

Figure 6-15. *Graph of the CNN used to classify the CIFAR10 dataset*

Training CNN

After building the computational graph of the CNN, next is to train it against the previously prepared training data. The training is done according to Listing 6-31. The code starts by preparing the path of the dataset and the data placeholders. Note that the path should be changed to be suitable for your system. Then it calls the previously discussed functions. The predictions of the trained CNN are used to measure the cost of the network, which is to be minimized using the GD optimizer. Some of the tensors have descriptive names to make it easier to retrieve them later when testing the CNN.

Listing 6-31. Training CNN

```
#Number of classes in the dataset. Used to specify the number of outputs in
the last fully connected layer.
num_dataset_classes = 10
#Number of rows & columns in each input image. The image is expected to be
rectangular Used to reshape the images and specify the input tensor shape.
im_dim = 32
#Number of channels in each input image. Used to reshape the images and
specify the input tensor shape.
num_channels = 3

#Directory at which the training binary files of the CIFAR10 dataset are
saved.
patches_dir = "\\AhmedGad\\cifar-10-python\\cifar-10-batches-py\\"

#Reading the CIFAR10 training binary files and returning the input data and
output labels. Output labels are used to test the CNN prediction accuracy.
dataset_array, dataset_labels = get_dataset_images(dataset_path=patches_
dir, im_dim=im_dim, num_channels=num_channels)
print("Size of data : ", dataset_array.shape)

# Input tensor to hold the data read in the preceding. It is the entry
point of the computational graph.
# The given name of 'data_tensor' is useful for retrieving it when
restoring the trained model graph for testing.
```

```
data_tensor = tensorflow.placeholder(tensorflow.float32, shape=[None, im_
dim, im_dim, num_channels], name='data_tensor')

# Tensor to hold the outputs label.
# The name "label_tensor" is used for accessing the tensor when testing the
saved trained model after being restored.

label_tensor = tensorflow.placeholder(tensorflow.float32, shape=[None],
name='label_tensor')

#The probability of dropping neurons in the dropout layer. It is given a
name for accessing it later.
keep_prop = tensorflow.Variable(initial_value=0.5, name="keep_prop")

#Building the CNN architecture and returning the last layer which is the
fully connected layer.
fc_result2 = create_CNN(input_data=data_tensor, num_classes=num_dataset_
classes, keep_prop=keep_prop)

# Predictions propabilities of the CNN for each training sample.
# Each sample has a probability for each of the 10 classes in the dataset.
# Such a tensor is given a name for accessing it later.

softmax_propabilities = tensorflow.nn.softmax(fc_result2, name="softmax_
probs")

# Predictions labels of the CNN for each training sample.
# The input sample is classified as the class of the highest probability.
# axis=1 indicates that maximum of values in the second axis is to be
returned. This returns that maximum class probability of each sample.

softmax_predictions = tensorflow.argmax(softmax_propabilities, axis=1)

#Cross entropy of the CNN based on its calculated propabilities.
cross_entropy = tensorflow.nn.softmax_cross_entropy_with_
logits(logits=tensorflow.reduce_max(input_tensor=softmax_propabilities,
reduction_indices=[1]), labels=label_tensor)
```

```
#Summarizing the cross entropy into a single value (cost) to be minimized
by the learning algorithm.
cost = tensorflow.reduce_mean(cross_entropy)
#Minimizing the network cost using the Gradient Descent optimizer with a
learning rate is 0.01.
error = tensorflow.train.GradientDescentOptimizer(learning_rate=.01).
minimize(cost)

#Creating a new TensorFlow Session to process the computational graph.
sess = tensorflow.Session()
#Writing summary of the graph to visualize it using TensorBoard.
tensorflow.summary.FileWriter(logdir="\\AhmedGad\\TensorBoard\\",
graph=sess.graph)
#Initializing the variables of the graph.
sess.run(tensorflow.global_variables_initializer())

# Because it may be impossible to feed the complete data to the CNN on
normal machines, it is recommended to split the data into a number of
patches.
# A subset of the training samples is used to create each path. Samples for
each path can be randomly selected.

num_patches = 5#Number of patches
for patch_num in numpy.arange(num_patches):
    print("Patch : ", str(patch_num))
    percent = 80 #percent of samples to be included in each path.

    #Getting the input-output data of the current path.
    shuffled_data, shuffled_labels = get_patch(data=dataset_array,
    labels=dataset_labels, percent=percent)

    #Data required for cnn operation. 1)Input Images, 2)Output Labels, and
    3)Dropout probability
    cnn_feed_dict = {data_tensor: shuffled_data,
                     label_tensor: shuffled_labels,
                     keep_prop: 0.5}
```

```
# Training the CNN based on the current patch.
# CNN error is used as input in the run to minimize it.
# SoftMax predictions are returned to compute the classification accuracy.

    softmax_predictions_, _ = sess.run([softmax_predictions, error], feed_
    dict=cnn_feed_dict)
    #Calculating number of correctly classified samples.
    correct = numpy.array(numpy.where(softmax_predictions_ == shuffled_
    labels))
    correct = correct.size
    print("Correct predictions/", str(percent * 50000/100), ' : ', correct)

#Closing the session
sess.close()
```

Rather than feeding the entire training data to the CNN, just a subset of the data is returned. This helps adjust the data to the amount of memory available. The subset is returned using the "get_patch" function according to Listing 6-32. This function accepts the input data, labels, and percentage of samples to be returned from the data. It then returns a subset of the data according to the specified percentage.

Listing 6-32. Splitting Dataset into Patches

```
def get_patch(data, labels, percent=70):
    num_elements = numpy.uint32(percent*data.shape[0]/100)
    shuffled_labels = labels#Temporary variable to hold the data after
    being shuffled.
    numpy.random.shuffle(shuffled_labels)#Randomly reordering the labels.

    return data[shuffled_labels[:num_elements], :, :, :], shuffled_
    labels[:num_elements]
```

Saving the Trained Model

After training the CNN, the model is saved for reuse later for testing according to Listing 6-33. You should also change the path where the model is saved to be suitable for your system.

Listing 6-33. Saving the Trained CNN Model

```
#Saving the model after being trained.
saver = tensorflow.train.Saver()
save_model_path = "\\AhmedGad\\model\\"
save_path = saver.save(sess=sess, save_path=save_model_path+"model.ckpt")
print("Model saved in : ", save_path)
```

Complete Code to Build and Train CNN

After going through all parts of the project from reading the data until saving the trained model, a summary of the steps is given in Figure 6-16. Listing 6-34 gives the complete code for training the CNN. After saving the trained model, it will be used to predict the class labels of the test data.

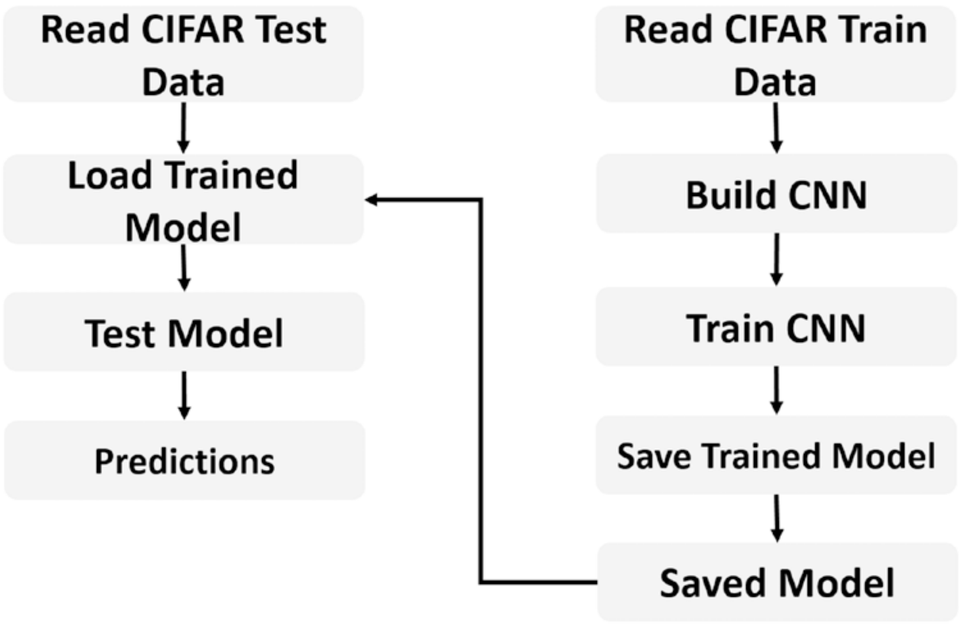

Figure 6-16. *Summary of steps for building a CNN trained using CIFAR10 dataset*

Listing 6-34. Complete Code to Train CNN for CIFAR10 Dataset

```
import pickle
import tensorflow
import numpy
import matplotlib.pyplot
import scipy.misc
import os

def get_dataset_images(dataset_path, im_dim=32, num_channels=3):
    """
    This function accepts the dataset path, reads the data, and returns it
    after being reshaped to match the requirements of the CNN.
    :param dataset_path:Path of the CIFAR10 dataset binary files.
    :param im_dim:Number of rows and columns in each image. The image is
    expected to be rectangular.
    :param num_channels:Number of color channels in the image.
    :return:Returns the input data after being reshaped and output labels.
    """
    num_files = 5#Number of training binary files in the CIFAR10 dataset.
    images_per_file = 10000#Number of samples within each binary file.
    files_names = os.listdir(patches_dir)#Listing the binary files in the
    dataset path.
# Creating an empty array to hold the entire training data after being
reshaped. The dataset has 5 binary files holding the data. Each binary
file has 10,000 samples. Total number of samples in the dataset is
5*10,000=50,000.
# Each sample has a total of 3,072 pixels. These pixels are reshaped to
form a RGB image of shape 32x32x3.
# Finally, the entire dataset has 50,000 samples and each sample of shape
32x32x3 (50,000x32x32x3).
    dataset_array = numpy.zeros(shape=(num_files * images_per_file, im_dim,
    im_dim, num_channels))
    #Creating an empty array to hold the labels of each input sample. Its
    size is 50,000 to hold the label of each sample in the dataset.
    dataset_labels = numpy.zeros(shape=(num_files * images_per_file),
    dtype=numpy.uint8)
    index = 0#Index variable to count number of training binary files being
    processed.
    for file_name in files_names:
# Because the CIFAR10 directory does not only contain the desired training
files and has some  other files, it is required to filter the required
files. Training files start by 'data_batch_' which is used to test whether
the file is for training or not.
        if file_name[0:len(file_name) - 1] == "data_batch_":
            print("Working on : ", file_name)
# Appending the path of the binary files to the name of the current file.
# Then the complete path of the binary file is used to decoded the file and
return the actual pixels values.
            data_dict = unpickle_patch(dataset_path+file_name)
# Returning the data using its key 'data' in the dictionary.
# Character b is used before the key to tell it is binary string.
            images_data = data_dict[b"data"]
            #Reshaping all samples in the current binary file to be of
            32x32x3 shape.
            images_data_reshaped = numpy.reshape(images_data,
            newshape=(len(images_data), im_dim, im_dim, num_channels))
            #Appending the data of the current file after being reshaped.
            dataset_array[index * images_per_file:(index + 1) * images_per_
            file, :, :, :] = images_data_reshaped
            #Appending the labels of the current file.
            dataset_labels[index * images_per_file:(index + 1) * images_
            per_file] = data_dict[b"labels"]
            index = index + 1#Incrementing the counter of the processed
            training files by 1 to accept new file.
    return dataset_array, dataset_labels#Returning the training input data
    and output labels.

def unpickle_patch(file):
    """
    Decoding the binary file.
    :param file:File path to decode its data.
    :return: Dictionary of the file holding details including input data
    and output labels.
    """
    patch_bin_file = open(file, 'rb')#Reading the binary file.
    patch_dict = pickle.load(patch_bin_file, encoding='bytes')#Loading the
    details of the binary file into a dictionary.
    return patch_dict#Returning the dictionary.

def get_patch(data, labels, percent=70):
    """
    Returning patch to train the CNN.
    :param data: Complete input data after being encoded and reshaped.
    :param labels: Labels of the entire dataset.
    :param percent: Percent of samples to get returned in each patch.
    :return: Subset of the data (patch) to train the CNN model.
    """
    #Using the percent of samples per patch to return the actual number of
    samples to get returned.
    num_elements = numpy.uint32(percent*data.shape[0]/100)
    shuffled_labels = labels#Temporary variable to hold the data after
    being shuffled.
    numpy.random.shuffle(shuffled_labels)#Randomly reordering the labels.
# The previously specified percent of the data is returned starting from
the beginning until meeting the required number of samples.
# The labels indices are also used to return their corresponding input
images samples.
    return data[shuffled_labels[:num_elements], :, :, :], shuffled_
    labels[:num_elements]

def create_conv_layer(input_data, filter_size, num_filters):
    """
    Builds the CNN convolution (conv) layer.
    :param input_data:patch data to be processed.
    :param filter_size:#Number of rows and columns of each filter. It is
    expected to have a rectangular filter.
    :param num_filters:Number of filters.
    :return:The last fully connected layer of the network.
    """
# Preparing the filters of the conv layer by specifying  its shape.
# Number of channels in both input image and each filter must match.
# Because number of channels is specified in the shape of the input image
as the last value, index of -1 works fine.
    filters = tensorflow.Variable(tensorflow.truncated_
    normal(shape=(filter_size, filter_size, tensorflow.cast(input_data.
    shape[-1], dtype=tensorflow.int32), num_filters), stddev=0.05))
    print("Size of conv filters bank : ", filters.shape)

# Building the convolution layer by specifying the input data, filters,
strides along each of the 4 dimensions, and the padding.
# Padding value of 'VALID' means the some borders of the input image will
be lost in the result based on the filter size.
    conv_layer = tensorflow.nn.conv2d(input=input_data,
                                      filter=filters,
                                      strides=[1, 1, 1, 1],
                                      padding="VALID")
    print("Size of conv result : ", conv_layer.shape)
    return filters, conv_layer#Returning the filters and the convolution
    layer result.

def create_CNN(input_data, num_classes, keep_prop):
    """
    Builds the CNN architecture by stacking conv, relu, pool, dropout, and
    fully connected layers.
    :param input_data:patch data to be processed.
    :param num_classes:Number of classes in the dataset. It helps to
    determine the number of outputs in the last fully connected layer.
    :param keep_prop:probability of keeping neurons in the dropout layer.
    :return: last fully connected layer.
    """
    #Preparing the first convolution layer.
    filters1, conv_layer1 = create_conv_layer(input_data=input_data,
    filter_size=7, num_filters=4)
# Applying ReLU activation function over the conv layer output.
# It returns a new array of the same shape as the input array.
    relu_layer1 = tensorflow.nn.relu(conv_layer1)
    print("Size of relu1 result : ", relu_layer1.shape)
# Max-pooling is applied to the ReLU layer result to achieve translation
invariance. It returns a new array of a different shape from the input
array relative to the strides and kernel size used.
    max_pooling_layer1 = tensorflow.nn.max_pool(value=relu_layer1,
                                                ksize=[1, 2, 2, 1],
                                                strides=[1, 1, 1, 1],
                                                padding="VALID")
    print("Size of maxpool1 result : ", max_pooling_layer1.shape)

    #Similar to the previous conv-relu-pool layers, new layers are just
    stacked to complete the CNN architecture.
    #Conv layer with 3 filters and each filter is of size 5x5.
    filters2, conv_layer2 = create_conv_layer(input_data=max_pooling_
    layer1, filter_size=5, num_filters=3)
    relu_layer2 = tensorflow.nn.relu(conv_layer2)
    print("Size of relu2 result : ", relu_layer2.shape)
    max_pooling_layer2 = tensorflow.nn.max_pool(value=relu_layer2,
                                                ksize=[1, 2, 2, 1],
                                                strides=[1, 1, 1, 1],
                                                padding="VALID")
    print("Size of maxpool2 result : ", max_pooling_layer2.shape)

    #Conv layer with 2 filters and a filter size of 5x5.
    filters3, conv_layer3 = create_conv_layer(input_data=max_pooling_
    layer2, filter_size=3, num_filters=2)
    relu_layer3 = tensorflow.nn.relu(conv_layer3)
    print("Size of relu3 result : ", relu_layer3.shape)
    max_pooling_layer3 = tensorflow.nn.max_pool(value=relu_layer3,
                                                ksize=[1, 2, 2, 1],
                                                strides=[1, 1, 1, 1],
                                                padding="VALID")
    print("Size of maxpool3 result : ", max_pooling_layer3.shape)

    #Adding dropout layer before the fully connected layers to avoid
    overfitting.
    flattened_layer = dropout_flatten_layer(previous_layer=max_pooling_
    layer3, keep_prop=keep_prop)

    #First fully connected (FC) layer. It accepts the result of the dropout
    layer after being flattened (1D).
    fc_result1 = fc_layer(flattened_layer=flattened_layer, num_
    inputs=flattened_layer.get_shape()[1:].num_elements(),
                          num_outputs=200)
    #Second fully connected layer accepting the output of the previous
    fully connected layer. Number of outputs is equal to the number of
    dataset classes.
    fc_result2 = fc_layer(flattened_layer=fc_result1, num_inputs=fc_
    result1.get_shape()[1:].num_elements(),
                          num_outputs=num_classes)
    print("Fully connected layer results : ", fc_result2)
    return fc_result2#Returning the result of the last FC layer.

def dropout_flatten_layer(previous_layer, keep_prop):
    """
    Applying the dropout layer.
    :param previous_layer: Result of the previous layer to the dropout
    layer.
    :param keep_prop: Probability of keeping neurons.
    :return: flattened array.
    """
    dropout = tensorflow.nn.dropout(x=previous_layer, keep_prob=keep_prop)
    num_features = dropout.get_shape()[1:].num_elements()
    layer = tensorflow.reshape(previous_layer, shape=(-1, num_features))
    #Flattening the results.
    return layer

def fc_layer(flattened_layer, num_inputs, num_outputs):
    """
    building a fully connected (FC) layer.
    :param flattened_layer: Previous layer after being flattened.
    :param num_inputs: Number of inputs in the previous layer.
    :param num_outputs: Number of outputs to be returned in such FC layer.
    :return:
    """
    #Preparing the set of weights for the FC layer. It depends on the
    number of inputs and number of outputs.
    fc_weights = tensorflow.Variable(tensorflow.truncated_
    normal(shape=(num_inputs, num_outputs), stddev=0.05))
    #Matrix multiplication between the flattened array and the set of
    weights.
    fc_result1 = tensorflow.matmul(flattened_layer, fc_weights)
    return fc_result1#Output of the FC layer (result of matrix
    multiplication).

#***********************************************************
#Number of classes in the dataset. Used to specify number of outputs in the
last FC layer.
num_dataset_classes = 10
#Number of rows & columns in each input image. The image is expected to be
rectangular Used to reshape the images and specify the input tensor shape.
im_dim = 32
# Number of channels in each input image. Used to reshape the images and
specify the input tensor shape.
num_channels = 3

#Directory at which the training binary files of the CIFAR10 dataset are
saved.
patches_dir = "\\AhmedGad\\cifar-10-python\\cifar-10-batches-py\\"
```

```
#Reading the CIFAR10 training binary files and returning the input data and
output labels. Output labels are used to test the CNN prediction accuracy.
dataset_array, dataset_labels = get_dataset_images(dataset_path=patches_
dir, im_dim=im_dim, num_channels=num_channels)
print("Size of data : ", dataset_array.shape)

# Input tensor to hold the data read in the preceding. It is the entry
point of the computational graph.
# The given name of 'data_tensor' is useful for retrieving it when
restoring the trained model graph for testing.
data_tensor = tensorflow.placeholder(tensorflow.float32, shape=[None, im_
dim, im_dim, num_channels], name='data_tensor')

# Tensor to hold the outputs label.
# The name "label_tensor" is used for accessing the tensor when testing the
saved trained model after being restored.
label_tensor = tensorflow.placeholder(tensorflow.float32, shape=[None],
name='label_tensor')

#The probability of dropping neurons in the dropout layer. It is given a
name for accessing it later.
keep_prop = tensorflow.Variable(initial_value=0.5, name="keep_prop")

#Building the CNN architecture and returning the last layer which is the FC
layer.
fc_result2 = create_CNN(input_data=data_tensor, num_classes=num_dataset_
classes, keep_prop=keep_prop)

# Predictions propabilities of the CNN for each training sample.
# Each sample has a probability for each of the 10 classes in the dataset.
# Such a tensor is given a name for accessing it later.
softmax_propabilities = tensorflow.nn.softmax(fc_result2, name="softmax_
probs")

# Predictions labels of the CNN for each training sample.
# The input sample is classified as the class of the highest probability.
# axis=1 indicates that maximum of values in the second axis is to be
returned. This returns that maximum class probability of each sample.
```

```
softmax_predictions = tensorflow.argmax(softmax_propabilities, axis=1)

#Cross entropy of the CNN based on its calculated propabilities.
cross_entropy = tensorflow.nn.softmax_cross_entropy_with_
logits(logits=tensorflow.reduce_max(input_tensor=softmax_propabilities,
reduction_indices=[1]),labels=label_tensor)
#Summarizing the cross entropy into a single value (cost) to be minimized
by the learning algorithm.
cost = tensorflow.reduce_mean(cross_entropy)
#Minimizing the network cost using the Gradient Descent optimizer with a
learning rate is 0.01.
ops = tensorflow.train.GradientDescentOptimizer(learning_rate=.01).
minimize(cost)

#Creating a new TensorFlow Session to process the computational graph.
sess = tensorflow.Session()
#Writing summary of the graph to visualize it using TensorBoard.
tensorflow.summary.FileWriter(logdir="\\AhmedGad\\TensorBoard\\",
graph=sess.graph)
#Initializing the variables of the graph.
sess.run(tensorflow.global_variables_initializer())

# Because it may be impossible to feed the complete data to the CNN on
normal machines, it is recommended to split the data into a number of
patches. A subset of the training samples is used to create each path.
Samples for each path can be randomly selected.

num_patches = 5#Number of patches
for patch_num in numpy.arange(num_patches):
    print("Patch : ", str(patch_num))
    percent = 80 #percent of samples to be included in each path.
    #Getting the input-output data of the current path.
    shuffled_data, shuffled_labels = get_patch(data=dataset_array,
    labels=dataset_labels, percent=percent)
    #Data required for cnn operation. 1)Input Images, 2)Output Labels,
    and 3)Dropout probability
    cnn_feed_dict = {data_tensor: shuffled_data,
                     label_tensor: shuffled_labels,
                     keep_prop: 0.5}

# Training the CNN based on the current patch.
# CNN error is used as input in the run to minimize it.
# SoftMax predictions are returned to compute the classification accuracy.

    softmax_predictions_, _ = sess.run([softmax_predictions, ops], feed_
    dict=cnn_feed_dict)
    #Calculating number of correctly classified samples.
    correct = numpy.array(numpy.where(softmax_predictions_ == shuffled_
    labels))
    correct = correct.size
    print("Correct predictions/", str(percent * 50000/100), ' : ', correct)

#Closing the session
sess.close()

#Saving the model after being trained.
saver = tensorflow.train.Saver()
save_model_path = " \\AhmedGad\\model\\"
save_path = saver.save(sess=sess, save_path=save_model_path+"model.ckpt")
print("Model saved in : ", save_path)
```

Preparing Test Data

Before testing the trained model, it is required to prepare the test data and restore the previously trained model. Test data preparation is similar to what happened with the training data except that there is just a single binary file to be decoded. The test file is decoded according to the modified "get_dataset_images" function according to Listing 6-35. Note that it has the same name as the function used to decode the training data because it is assumed that there are two separate scripts, one for training and another for testing. This function calls the "unpickle_patch" function exactly as done before with training data.

Listing 6-35. Saving the Trained CNN Model

```
def get_dataset_images(test_path_path, im_dim=32, num_channels=3):
    data_dict = unpickle_patch(test_path_path)
    images_data = data_dict[b"data"]
    dataset_array = numpy.reshape(images_data, newshape=(len(images_data),
    im_dim, im_dim, num_channels))
    return dataset_array, data_dict[b"labels"]
```

Testing the Trained CNN Model

According to Figure 6-16, the saved model will be used to predict the labels for the test data. After preparing the test data and restoring the trained model, we can start testing the model according to Listing 6-36. It's worth mentioning that when training the CNN, the session runs to minimize the cost. In testing, we are not interested in minimizing the cost anymore and just we would like to return the predictions for the data samples. This is why the TF session runs to return just the predictions by fetching the "softmax_propabilities" and "softmax_predictions" tensors.

When the graph is restored, the tensor named "data_tensor" in the training phase will be assigned the testing data, while the tensor named "label_tensor" will be assigned the sample labels.

Another interesting point is that the keep probability "keep_prop" of the dropout layer is now set to 1.0. That means do not drop any neuron (i.e., use all neurons). This is because we are just using the pretrained model after settling on what neurons to drop. Now we just use what the model did before and are not interested in making any modifications.

Listing 6-36. Testing the Trained CNN

```
#Dataset path containing the testing binary file to be decoded.
patches_dir = "\\AhmedGad\\cifar-10-python\\cifar-10-batches-py\\"
dataset_array, dataset_labels = get_dataset_images(test_path_path=patches_
dir + "test_batch", im_dim=32, num_channels=3)
print("Size of data : ", dataset_array.shape)

sess = tensorflow.Session()
```

```
#Restoring the previously saved trained model.
saved_model_path = '\\AhmedGad\\model\\'
saver = tensorflow.train.import_meta_graph(saved_model_path+'model.ckpt.meta')
saver.restore(sess=sess, save_path=saved_model_path+'model.ckpt')

#Initializing the variables.
sess.run(tensorflow.global_variables_initializer())

graph = tensorflow.get_default_graph()


softmax_propabilities = graph.get_tensor_by_name(name="softmax_probs:0")
softmax_predictions = tensorflow.argmax(softmax_propabilities, axis=1)
data_tensor = graph.get_tensor_by_name(name="data_tensor:0")
label_tensor = graph.get_tensor_by_name(name="label_tensor:0")
keep_prop = graph.get_tensor_by_name(name="keep_prop:0")

#keep_prop is equal to 1 because there is no more interest to remove
neurons in the testing phase.
feed_dict_testing = {data_tensor: dataset_array,
                     label_tensor: dataset_labels,
                     keep_prop: 1.0}

#Running the session to predict the outcomes of the testing samples.
softmax_propabilities_, softmax_predictions_ = sess.run([softmax_
propabilities, softmax_predictions], feed_dict=feed_dict_testing)

#Assessing the model accuracy by counting number of correctly classified
samples.
correct = numpy.array(numpy.where(softmax_predictions_ == dataset_labels))
correct = correct.size
print("Correct predictions/10,000 : ", correct)

#Closing the session
sess.close()
```

At this point, we have successfully built the CNN model for classifying images of the CIFAR10 dataset. In the next chapter, the saved trained CNN model is deployed to a web server created using Flask for being accessed from Internet users.