TensorFlow: A Guide To Build Artificial Neural Networks Using Python

Book · December 2017

CITATION
1

READS
2,619

1 author:

Ahmed Fawzy Gad
Faculty of Computers and Information - Menoufia University
51 PUBLICATIONS **34** CITATIONS

SEE PROFILE

This guide assumes you know nothing about TensorFlow and takes you from the beginning until understanding the basics of a TensorFlow program including Variables, Placeholders, dataflow graphs, TensorFlow Core API, and TensorBoard for visualization. Because artificial neural networks (ANNs) are in the heart of deep learning models, it is recommended to start learning how they work and implemented programmatically. This guide covers in details all steps required for creating your first ANN using TensorFlow starting by reading input data then building neural networks layers (input, hidden, output) and finally making predictions. This guide helps you how neural networks parameters (e.g. weights) are updated by tracing the dataflow graph. Because the traditional way of reading data using placeholders is not appropriate for large datasets, pandas Series and DataFrame are discussed by understanding the different ways they are created. This is in addition to indexing, updating, deleting items. In addition to TensorFlow Core API, some higher level APIs are discussed including TensorFlow Estimators and train for saving time wasted by implementing some of the frequently used operations.

Ahmed Fawzy Gad is an Egyptian graduate who received the B.Sc. degree with excellent with honors in information technology in 2015 from FCI, Menoufia University, Egypt. Ahmed was selected in 2016 to work as a TA and a researcher. His research interests include AI, ML, DL, DSP, and CV.

978-620-2-07312-7

Ahmed F. Gad

TensorFlow: A Guide to Build Artificial Neural Networks using Python

Build artificial neural networks using TensorFlow library with detailed explanation of each step and line of code

LAP LAMBERT Academic Publishing

Ahmed F. Gad

TensorFlow: A Guide to Build Artificial Neural Networks using Python

Ahmed F. Gad

TensorFlow: A Guide to Build Artificial Neural Networks using Python

Build artificial neural networks using TensorFlow library with detailed explanation of each step and line of code

LAP LAMBERT Academic Publishing

Imprint
Any brand names and product names mentioned in this book are subject to trademark, brand or patent protection and are trademarks or registered trademarks of their respective holders. The use of brand names, product names, common names, trade names, product descriptions etc. even without a particular marking in this work is in no way to be construed to mean that such names may be regarded as unrestricted in respect of trademark and brand protection legislation and could thus be used by anyone.

Cover image: www.ingimage.com

Publisher:
LAP LAMBERT Academic Publishing
is a trademark of
International Book Market Service Ltd., member of OmniScriptum Publishing Group
17 Meldrum Street, Beau Bassin 71504, Mauritius

Printed at: see last page
ISBN: 978-620-2-07312-7

Table of Contents

Chapter 1

TensorFlow Overview

TensorFlow Level of Control

TensorFlow provided multiple APIs that different in the level of control over the programs. The lowest level API in TensorFlow is called TensorFlow Core that gives the programmer the ability to control every piece of code and have much better control over the created machine learning models. For example, if you want to find the summation of all elements inside an array, you can implement it manually by using loops (like TensorFlow Core API) or use an already existing method for calculating the summation (like other high-level TensorFlow APIs).
But there are also a number of higher-level APIs in TensorFlow that make things easier for programmers as it gets them off the details and just provides a simple interface for frequently used tasks. But all higher-level TensorFlow APIs are built on top of TensorFlow Core. For example, TensorFlow Estimators is a high-level API in TensorFlow that makes creating models easier than TensorFlow Core but it hides many details from the programmers. Other high-level APIs in TensorFlow are like Keras, TF-Slim, TF-Layers, and TF-Learn. For researchers, it is better to use TensorFlow Core as they will get much better control over their models.

To understand TensorFlow, get into your mind that rather than having variables and methods in previous programming tools, TensorFlow has Tensors and Operations which are regarded nodes in a graph.

Tensor

Tensor is the central data unit in TensorFlow. Tensor consists of a set of primitive data types shaped into an array of any dimensions. The Tensor primitive data types are like integer, floating point, character, string, and so on.
Number of dimensions of the array defines the tensor`s rank. Tensor rank is the tensor dimensions. Rank 0 tensor is for tensors with scalar values. Note that the rank of a tensor is different from the shape of a NumPy array. NumPy array shape returns the

number of elements within the dimension but tensor rank returns just the number of dimensions. For example, the shape of this array [1, 2, 3] is 3 where its rank is just 1. Also Tensor`s rank is just a scalar value like 5, 2, 6 that represents the number of dimensions in the tensor. But shape of the array is a tuple like (4, 3) indicating that the array has 2 dimensions where the sizes of these dimensions are 4 and 3 respectively. Here are some examples:

```
- 5 # Rank 0 tensor, scalar with [] shape
- [4, 8] # Rank 1 tensor, [2] shape
- [[3, 1, 7], [1, 5, 2]] # Rank 2 tensor, [2, 2] shape
- [[[8, 3]], [[11, 9]]]] # Rank 2 tensor, [2, 1, 2] shape
```

TensorFlow Core Tutorial

Before starting creating programs using TensorFlow you must import it. It can be imported using this simple import statement:

```
1. import tensorflow
```

Or using tf as an alias:

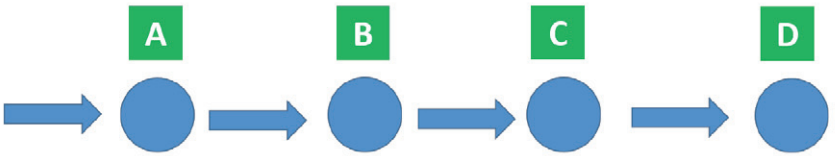
```
1. import tensorflow as tf
```

TensorFlow Core programs can be regarded as consisting of two discrete sections:

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

TensorFlow uses a dataflow graph to represent the computations in the program. This type of programming requires the creating of a dataflow graph showing the sequence of computations. After specifying the sequence of computations, **parts (not necessary the entire graph) of the graph will run** into a TensorFlow session on local or remote machines. TensorFlow session works as a connection between the program in Python and the C++ runtime environment in which some information can be stored/cached for making it easier to run the graph multiple times.

You can run selected parts of the graph and not required to run all of its parts. For example, in the next figure if you selected to run the graph until C then D won't run. If you run B then C and D won't run.



Note that working with TensorFlow Core API requires understanding dataflow graphs and sessions. Working with high-level APIs in TensorFlow like Estimators and Keras will hide the details of graphs and sessions from the user. But understanding graphs and session are useful for understanding how they are implemented.

Dataflow Graph

Dataflow is a programming model for parallel computing. In dataflow graph, nodes represent units of computation (Tensor or Operation) and the edges represent input to and output from a computation. For example, graphing the `tensorflow.matmul` operation in TensorFlow will create a graph with a single node representing the `tensorflow.matmul` method connected to two edges as input which are the two matrices to get multiplied and one edge as an output representing the result of the multiplication.

Why using dataflow graphs?

This is because there are a number of benefits from using dataflow in TensorFlow:

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

The `tensorflow.graph` contains two parts:

1. **Graph Structure**: The nodes and edges of the graphs and their connection.
2. **Graph Collection**: Data.

The computational dataflow graph in TensorFlow is a series of TensorFlow operations arranged into a graph of nodes. Each node in the computational graph accepts zero or more Tensors as input and generates one tensor as an output. But a special kind of nodes in TensorFlow is **constant node/Tensor** as it zero Tensors as input. The output that the constant node generates is a value it stores internally.
For example, the following code creates a single constant node of type float32 and prints it.

```
1. import tensorflow
```

```
node1 = tensorflow.constant(3.7, dtype=tensorflow.float32
)
print(node1)
```

The result of printing:

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

The output of printing the constant node doesn't print the value inside the constant but it prints the node/Tensor itself. The value will get printed only after evaluating the nodes.
For evaluating the node inside the computational graph, the computational graph must run within a session.

TensorFlow Session

TensorFlow uses the `tensorflow.Session` class to represent a connection between the client program--typically a Python program, and the C++ runtime environment. A `tensorflow.Session` object provides access to devices in the local machine, and remote devices using the distributed TensorFlow runtime environment. It also caches information about your `tensorflow.Graph` so that you can efficiently run the same computation multiple times.

Creating a tensorflow.Session

When using a low-level TensorFlow API like TensorFlow Core, you can create a session for the current default graph as follows:

```
#Local session
with tensorflow.Session() as sess:
          #...

#Remote session
with tensorflow.Session("grpc://example.org:2222") as ses
s:
          #...
```

Because the `tensorflow.Session` own physical resources like CPUs, GPUs, and network connections, it must free these resources after finishing execution. When the session gets created within the `with` block, it will get closed automatically after getting outside the block. But also we can create a session without using the `with` block. In this case, we have to manually exit the session using the `tensorflow.Session.close()` to free resources.

```
#Local session
sess = tensorflow.Session()
#...
#Exit session
sess.close()

#Remote session
sess = tensorflow.Session("grpc://example.org:2222")
#...
    #Exit session
    sess.close()
```

To run the session, we can use the `tensorflow.Session.run()`. This method runs the `tensorflow.Operation` and evaluates the `tensorflow.Tensor`. It accepts one or more operation or tensor.

Creating Tensor Nodes

Here is an example used to create two Tensor nodes, create `tensorflow.Session`, run it using `tensorflow.Session.run()`, prints only one tensor, and finally closes the session using `tensorflow.Session.close()`.

```
import tensorflow

node1 = tensorflow.constant(1.7, tensorflow.float16)
node2 = tensorflow.constant(8, tensorflow.float16)

sess = tensorflow.Session()
print(sess.run(node1))
sess.close()
```

To print more than one Tensor at the same time, place them within an array using two square brackets:

```
print(sess.run([node1, node2]))
```

Important note:
The use of `tensorflow.Session.run()` is not limited to just evaluating unknown values of a Tensor but it is also responsible for evaluating the Tensors in order to return the values for any Tensor even if it is a constant (its value is known). So, it is a way for evaluating and printing values within Tensors.

Creating Operation Nodes

The previous example created two Tensor nodes. We can create complex operations by combining such Tensor nodes with Operation nodes.

```
import tensorflow

node1 = tensorflow.constant(1.7, tensorflow.float16)
node2 = tensorflow.constant(8, tensorflow.float16)

node3 = tensorflow.add(node1, node2)

sess = tensorflow.Session()

    writer = tensorflow.summary.FileWriter("/tmp/log/", sess.graph)

    print("Result of sess.run(node3) : ", sess.run(node3))

    writer.close()

    sess.close()
```

Here is the picture of the previous computational graph using **TensorBoard (TB)**.

Parameterized Graph

The previous graph is static as it always accepts the same inputs and generates the same output each time it is evaluated. To be able to modify the inputs each time the program runs we can use placeholders in `tensorflow.placeholder`. In other words, for evaluating the same operations but using different inputs, you should use `tensorflow.placeholder.`

A `tensorflow.placeholder` is a way for making the program accepts external inputs.

The `tensorflow.placeholder` has this syntax with three arguments and returns a Tensor.

```
tensorflow.placeholder(
    dtype,
    shape=None,
    name=None
)
```

dtype: the data type of elements the Tensor will accept.

shape (Optional – default None): The 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.

Using Placeholders

For example, we can modify the previous example to use `tensorflow.placeholder`. When running the session, we give the `tensorflow.Session.run()` method the node to be evaluated in addition to the providing the initial values of the placeholders in a dictionary using the `feed_dict` argument of the `tensorflow.Session.run()` method.

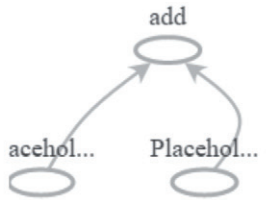
```
1. import tensorflow
2.
3. node1 = tensorflow.placeholder(dtype=tensorflow.float16)
```

```
4. node2 = tensorflow.placeholder(dtype=tensorflow.float16)

5.
6. node3 = node1 + node2 # + is identical (shortcut) to tens
   orflow.add(node1, node2)
7.
8. sess = tensorflow.Session()
9.
10.     writer = tensorflow.summary.FileWriter("/tmp/log/", s
   ess.graph)
11.
12.     print("Result of sess.run(node3) : ",  sess.run(node3
   , feed_dict={node1: 7, node2: 4}))
13.     print("Result of sess.run(node3) : ",  sess.run(node3
   , feed_dict={node1: 2.4, node2: 5.8}))
14.
15.     writer.close()
16.
17.     sess.close()
```

Note that we are able to run the same session multiple times with different values for each placeholder.

Here is the picture of the computational graph using **TensorBoard**.



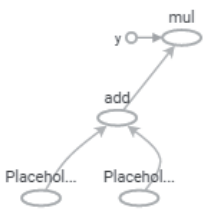
Here is the code after adding a new multiply operation node to the result of addition.

```
1. import tensorflow
2.
3. node1 = tensorflow.placeholder(dtype=tensorflow.float16)

4. node2 = tensorflow.placeholder(dtype=tensorflow.float16)
```

```
5.
6. node3 = node1 + node2 # + is identical (shortcut) to tens
   orflow.add(node1, node2)
7.
8. node4 = node3 * 5
9.
10.     sess = tensorflow.Session()
11.
12.     writer = tensorflow.summary.FileWriter("/tmp/log/", s
   ess.graph)
13.
14.     print("Result of sess.run(node4) : ",  sess.run(node4
   , feed_dict={node1: 7, node2: 4}))
15.     print("Result of sess.run(node4) : ",  sess.run(node4
   , feed_dict={node1: 2.4, node2: 5.8}))
16.
17.     writer.close()
18.
19.     sess.close()
```

Picture using TB:



Run Last Tensor in the Graph Chain

In lines 14 and 15 in the previous example, the Tensor we selected for evaluating its values and running the Session is the node4 Tensor. But why not selecting other Tensors like node3?

The answer is that we select the last Tensor in the graph chain for evaluation. This is because evaluating node4 will implicitly evaluate all other Tensors in the graph. But if we evaluated node3 then we won't evaluate node4 because node3 doesn't depend on node4 and we have nothing to do for evaluating node3. But node4 is what

depends on node3 and thus node3 and all other Tensors lower than node3 in the graph will be evaluated when selecting node4 in the run operation of the session. This is how you can run just part of the graph.

TensorFlow 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. That is if the same operation is to be applied multiple times but on different input data, then place such input data into a placeholder and run the session by assigning different values to the placeholder.
Placeholders are not initialized and get their value assigned during runtime. That is when calling the `tensorflow.Session.run()` we can give value for the placeholders.
Placeholder has an unconstrained shape which allows feeding Tensors of different shapes and data types to the placeholders. You can also constraint the shape and data type using the placeholder shape argument.
Suppose that you want to assign the training data to the placeholder and you just know that each sample get described by 35 features but you don't know how much samples you will train by. Then you can create a placeholder that will accept a Tensor with an unspecified number of samples but a specific number of features (columns) per sample. The following command did that.

```
1. node1 = tensorflow.placeholder(dtype=tensorflow.float16,
   shape=[None, 35])
```

But placeholder just accepts the value and cannot get changed after being assigned. Machine learning models have a number of **trainable parameters** that are required to get changed multiple times until reaching its final values. So, how to allow a Tensor to change its values multiple times? This is not provided by placeholders but variables.

TensorFlow variables are identical to the normal variables used in other languages. It is assigned an initial value and this value can be updated during the execution of the program based on the operations applied to it. Placeholder doesn't allow data modifications once assigned during execution time.
Variables have to be initialized when created. It can be initialized by a Tensor of any type and shape. The type and shape of such Tensor will define the type and shape of

the Variable which will still fixed and can't be changed after that. But the values will for sure eligible for changes.
Working with distributed environment, variables can be stored once and get shared across all devices. Also, variable value can be saved and restored when required.
TensorFlow variable has a state.
Also, placeholder is a function but Variable is a class and thus its name starts with uppercase.

There are many arguments for `tensorflow.Variable()` but there are two useful arguments for the Variable which are:

1. initial_value
2. dtype

Variables are created and added to the graph using the `tensorflow.Variable()` class. For example, the next line creates a variable of type float32 and get its initial value to 3.6:

```
1. variable1 = tensorflow.Variable(initial_value=[3.6], dtyp
   e=tensorflow.float32)
```

Remember that the main use of variables in TensorFlow is for holding trainable parameters to the graph like the weights of a model.

Working with constant Tensors, they values will get initialized once the `tensorflow.constant()`is called. But the values of the variables won't be initialized after calling `tensorflow.Variable()`.
There is an easy way to initialize all global variables within the program by calling this operation node and then run the program:

```
1. init = tensorflow.global_variables_initializer()
2. sess.run(init)
```

The variables won't get initialized until you call the `tensorflow.Session.run()`.

Variable Initialization

There are different ways to initialize a variable. But all variable initialization ways have a way to set both the shape and data type of the variable.

Initialize a variable based on another variable

One variable can be initialized based on the values in another variable. The initialized_value() property of a variable will return its initial value that can be used to initialize another variable.

```
1. var1 = tensorflow.Variable(initial_value=[3.6], dtype=ten
   sorflow.float32)
2. var2 = tensorflow.Variable(initial_value=var1.initialized
   _value(), dtype=tensorflow.float32)
3. var3 = tensorflow.Variable(initial_value=var1.initialized
   _value()*5, dtype=tensorflow.float32)
```

Variable Initialization Using Constants

A variable can be initialized based on a Constant Tensor. There are different operations to generate constants like:

- tensorflow.zeros
- tensorflow.ones
- tensorflow.lin_space
- tensorflow.range
- tensorflow.constant

They have the same meaning as their corresponding methods in NumPy and SciPy. All of these operations return Tensor of the specified data type and number of elements.

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

For example, we can create a `tensorflow.Variable()` whose values are initialized using `tensorflow.zeros()` returning a 1D row vector with 24 elements as follows:

```
biases = tensorflow.Variable(tensorflow.zeros([24]), name
="biases")
```

Example – Training a Linear Model

After knowing the difference between a placeholder and a variable in TensorFlow, we can make use of both to create a model.

For a simple linear model, there are input data, weights, and biases. What is the most suitable between placeholder and a variable to hold each one?

Generally, placeholder is used when we want to apply the same operation multiple times over different inputs. The inputs one by one will be assigned for the placeholder and having the operation got applied for each different input.

Variables are used for holding trainable parameters. So, input data is to be assigned to a placeholder while weights and biases are for being variables.

But remember to initialize the variables by calling the `tensorflow.global_variables_initializer()` and then run the `tensorflow.Session`.

```
weight = tensorflow.Variable([2.6], dtype=tensorflow.floa
t32)
bias = tensorflow.Variable([1], dtype=tensorflow.float32)

data = tensorflow.placeholder(dtype=tensorflow.float32)

linear_model = weights * data + bias

init = tensorflow.global_variables_initializer()
sess.run(init)
```

Because **data** is a placeholder, it will get assigned different values and each value will be fed to the program to get its result. If there are three inputs to get applied to the program, then we have to assign the placeholder each value independently of the other values and getting its output then go to the next value and so on as shown next.

```
print(sess.run(linear_model, feed_dict={data: [1]}))
print(sess.run(linear_model, feed_dict={data: [2]}))
print(sess.run(linear_model, feed_dict={data: [3]}))
```

```
print(sess.run(linear_model, feed_dict={data: [4]}))
```

But also we can assign values for the placeholder and getting the output calculated for each one simultaneously.

```
print(sess.run(linear_model, feed_dict={data: [1, 2, 3, 4]}))
```

The complete code for a very simple linear model is as follows:

```
import tensorflow

W = tensorflow.Variable([.6], dtype=tensorflow.float32) #Weight
b = tensorflow.Variable([.2], dtype=tensorflow.float32) #Bias

x = tensorflow.placeholder(dtype=tensorflow.float32) #Input Data

linear_model = W * x + b

    sess = tensorflow.Session()

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

    #Feeding the placeholder with data
    print(sess.run(linear_model, feed_dict={x: [1, 2, 3, 4]}))

    sess.close()
```

Loss Function

In the previous example, we just trained the linear model but we can't assess its accuracy. For doing that, we want to have a ground truth data which is the desired data to be generated by the model and comparing it with the actual data generated by the trained model.
To do this, we have to use a loss function. We can use the standard loss model for linear regression, which sums the squares of the deltas between the predictions of the current trained model and the provided desired data. This will generate scalar value as the overall error of the trained model.

At first, a new placeholder will be created for holding the desired outputs. Then we calculate the loss between the predicted outputs and values in such placeholder.

Here is the code after adding the loss:

```
1. import tensorflow
2.
3. #Trainable Parameters
4. W = tensorflow.Variable([.6], dtype=tensorflow.float32)
5. b = tensorflow.Variable([.2], dtype=tensorflow.float32)
6.
7. #Training Data (inputs/outputs)
8. x = tensorflow.placeholder(dtype=tensorflow.float32)
9. y = tensorflow.placeholder(dtype=tensorflow.float32)
10.
11.     #Linear Model
12.     linear_model = W * x + b
13.
14.     #Linear Regression Loss Function - sum of the squares

15.     squared_deltas = tensorflow.square(linear_model - y)

16.     loss = tensorflow.reduce_sum(squared_deltas)
17.
18.     #Creating a session
19.     sess = tensorflow.Session()
20.
```

```
    #Initializing variables
    init = tensorflow.global_variables_initializer()
    sess.run(init)

    #Print the loss
    print(sess.run(loss, feed_dict={ x: [1, 2, 3, 4],
  y: [0, 1, 2, 3]}))

    sess.close()
```

New Operations:

```
#Returns the sum of all elements across the specified
  axis. If no axis specified, all dimensions are reduced to
  return a Tensor with just one element.
reduce_sum(
    input_tensor,
    axis=None,
    keep_dims=False,
    name=None,
    reduction_indices=None
)

    #Calculates the square of x element wise.
    square(
        x,
        name=None
    )
```

Note that the Tensor used for evaluation in this modified example is the Tensor loss not linear_model because it is the last Tensor located at the top the graph.

The loss returned is 53.76. The existence of error, especially for large error, means that the parameters used must be updated.

These parameters are expected to be updated automatically but we can start updating it manually until reaching zero error.

For W=0.8 and b=0.4, the loss is 3.44
For W=1.0 and b=0.8, the loss is 12.96
For W=1.0 and b=-0.5, the loss is 1.0
For W=1.0 and b=-1.0, the loss is 0.0

Thus when W=1.0 and b=-1.0 the desired results is identical to the predicted results and thus we can't enhance the model more than this.
We reached the optimal values for the parameters but not using the optimal way. The optimal way of calculating the parameters is automatic.

There are a number of optimizers already exist in TensorFlow for making things simpler. These optimizers exist in APIs in TensorFlow like:

- tensorflow.train
- tensorflow.estimator

tensorflow.train

There are a number of optimizers that TensorFlow provides that makes the previous manual work of calculating the best values for the model parameters automatically.
The simplest optimizer is the gradient descent that changes the values of each parameter slowly until reaching the value that minimizes the loss. Gradient descent modifies each variable according to the magnitude of the derivative of loss with respect to the variable.
Because doing such operations of calculating the derivatives is complex and error-prone, TensorFlow can calculate the gradients automatically. After calculating the gradients, you need to optimize the parameters yourself.
But TensorFlow makes things easier and easier by providing optimizers that will calculate the derivatives in addition to optimizing the parameters.

`tensorflow.train` API contains a class called `GradientDescentOptimizer` that can both calculate the derivatives and optimizing the parameters.
The `GradientDescentOptimizer` class has the following constructor:

```
1. __init__(
2.     learning_rate,
3.     use_locking=False,
```

```
    name='GradientDescent'
)
```

For example, the following code shows how to minimize the loss using the `GradientDescentOptimizer`:

```
import tensorflow

#Trainable Parameters
W = tensorflow.Variable([0.3], dtype=tensorflow.float32)
b = tensorflow.Variable([-0.2], dtype=tensorflow.float32)

#Training Data (inputs/outputs)
x = tensorflow.placeholder(dtype=tensorflow.float32)
y = tensorflow.placeholder(dtype=tensorflow.float32)

    x_train = [1, 2, 3, 4]
    y_train = [0, 1, 2, 3]

    #Linear Model
    linear_model = W * x + b

    #Linear Regression Loss Function - sum of the squares
    squared_deltas = tensorflow.square(linear_model - y_train)
    loss = tensorflow.reduce_sum(squared_deltas)

    #Gradient descent optimizer
    optimizer = tensorflow.train.GradientDescentOptimizer(learning_rate=0.01)
    train = optimizer.minimize(loss=loss)

    #Creating a session
    sess = tensorflow.Session()
```

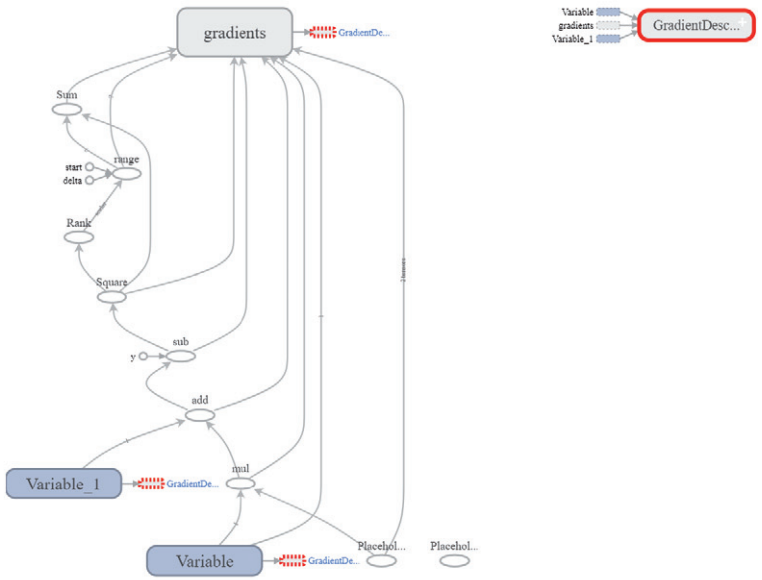
```
27.
28.     writer = tensorflow.summary.FileWriter("/tmp/log/", s
  ess.graph)
29.
30.     #Initializing variables
31.     init = tensorflow.global_variables_initializer()
32.     sess.run(init)
33.
34.     #Optimizing the parameters
35.     for i in range(1000):
36.         sess.run(train, feed_dict={x: x_train, y: y_train
  })
37.
38.     #Print the parameters and loss
39.     curr_W, curr_b, curr_loss = sess.run([W, b, loss], {x
  : x_train, y: y_train})
40.     print("W : ", curr_W, ", b : ", curr_b, ", loss : ",
  curr_loss)
41.
42.     writer.close()
43.
44.     sess.close()
```

Here is the result:

```
1. W :  [ 0.99999809] , b :  [-
  0.9999944] , loss :  2.05063e-11
```

The code for creating such simple linear regression model using TensorFlow Core API is not complex. But it won't be like that when working with much more complex models. Thus it is preferred to use the frequently used tasks from high-level APIs in TensorFlow.

Here is the graph when visualized using TB:



Important Question: How the optimizer knew the parameters that it should change?

In line 35, we run the session and asked to evaluate the train Tensor. TensorFlow will follow the chain of graph nodes to evaluate that Tensor. In line 23, TensorFlow found that to evaluate the train Tensor it should evaluate the optimizer.minimize operation. This operation will try to minimize its input arguments as much as possible. Following back, to evaluate the minimize operation it will accept one Tensor which is the loss. So, the goal now is to minimize the loss Tensor. But how to minimize the loss? It will still follow the graph back and it will find it is evaluated using the `tensorflow.reduce_sum()` operation. So, our goal now is to minimize the result of the `tensorflow.reduce_sum()` operation. Following back, this operation is evaluated using one Tensor as input which is `squared_deltas`. So, rather than having our goal to minimize the `tensorflow.reduce_sum()` operation, our goal now is to minimize the `squared_deltas` Tensor. Following the chain back, we find that the `squared_deltas` Tensor depends on the `tensorflow.square()` operation. So, we should minimize `tensorflow.square()` the operation. Minimizing that operation will ask us to minimize its input Tensors which are

`linear_model` and `y_train`. Looking for these two tensors, which one can be modified? The Tensors of type Variables can be modified. Because `y_train` is not a Variable but placeholder, then we can't modify it and thus we can modify the `linear_model` to minimize the result.

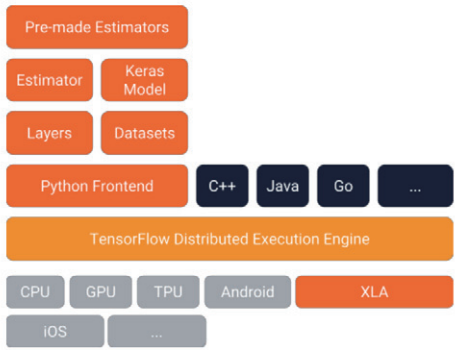
In line 15, the `linear_model` Tensor is calculated based on three inputs which are `W`, `x`, and `b`. Looking for these Tensors, only `W` and `x` can be changed because they are variables. So, our goal is to minimize these two Tensors `W` and `x`.

This is how TensorFlow deduced that to minimize the loss it should minimize the weight and bias parameters.

Chapter 2

TensorFlow Estimators

Using TensorFlow Core, we have to implement every piece of code and there are no already existing methods with your needs. Using `tensorflow.train`, we are able to do some tasks at a high level such as updating the parameters.
But also we have to implement some tasks manually that just add more overhead. Some tasks are recommended to be created using already existing methods because your custom implementation won't add any more functionality.
For such reasons, there is also another high-level TensorFlow API called `tensorflow.estimator` introduced in TensorFlow 1.3 implementing some frequently used functionalities useful when training a model reducing much of the boilerplate code. The following figures show the TensorFlow architecture after adding `tensorflow.estimator`. There also a new feature added to TensorFlow 1.3 called Datasets.
The diagram shows that Estimators themselves are based on Layers API.



Estimators are flexible as you can implement your own model or override default behavior of the models. It simplifies many tasks like:

- Running training loops
- Running evaluation loops
- Prediction

- Managing datasets

Using Estimators, there are two ways to build a model:

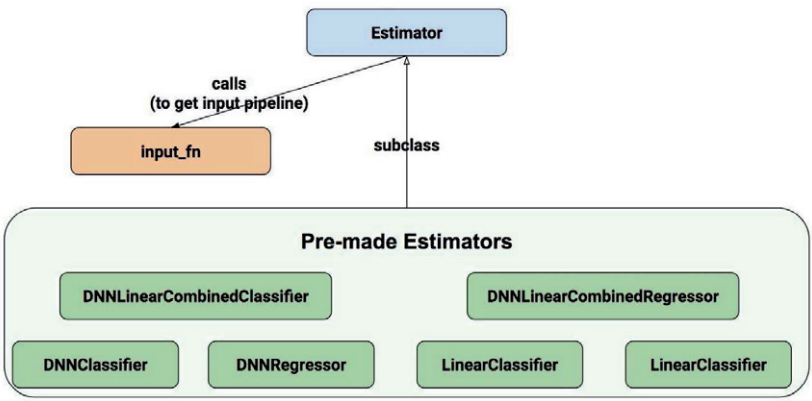
1. **Pre-Made Estimators**: There are some already existing estimators that you can use like DNNClassifier.
2. **Estimator (Base Class)**: If you want a model not existing in the pre-made Estimators or would like to change one of the pre-made Estimators, then you can create your own customized model. The Estimator class is the base class for any pre-made Estimators giving you the complete control of the model using the `model_fn` function.

The pre-made Estimators are just subclass of the Estimator class. When creating a new custom Estimator, you have to use the `tensorflow.estimator.Estimator` parent class.

Advantages of Estimators:

- Estimators build the graph for you and you don't have to build the graph.

The following figure shows the class diagram of Estimators.



When using Estimator, the data must be separated from the model. This allows using the same model with different datasets. Note that all Estimators make use of the `input_fn` function.

Pre-made Estimators

To avoid details about the base TensorFlow APIs and work from a very high conceptual level. Pre-made Estimators create and manage the Graph and Session objects for you. They also allow modifying the model architecture with minimal code.

Pre-made Estimators Program Structure

The TensorFlow program relying on pre-made Estimators has a structure with the following four steps:

1. Writing one or more dataset importing functions
2. Define the feature columns
3. Instantiate the pre-made Estimator
4. Call training, evaluation, or inference operations

1) Writing one or more dataset importing functions

Using the `input_fu` function, you can pass feature and target data to the train, evaluate, and predict operations of the Estimator. Data pre-processing can be made inside the `input_fn` function like scaling the inputs or removing bad samples.
The input functions must return the following two values (feature and labels) to be fed into your model:

1. `feature_cols`: It is a dictionary with key-value pairs mapping the feature column names (keys) and the Tensors (values) containing the corresponding feature data.
2. `labels`: This is a Tensor containing the label or target values which are the values our model should return.

Note:
Previously when reading the data into a NumPy array we were reading the entire dataset including samples for training, evaluation, and prediction. But when working with TensorFlow Estimators `input_fn`, we are working with each type of data separately. That is you have to prepare the training data separately from other data sets for evaluation and prediction. The notion of reading the entire data and dividing it within the code into parts for training, evaluation, and prediction is not easily supported.

Converting Feature Data to Tensors

Your data may be in a Python array (list), `pandas dataframes`, or a `numpy array`. To convert such data to Tensors, you will use the input function (`input_fn`) accepting such previously stored data.

Here is an example to convert a `numpy array` into Tensor using the `tensorflow.estimator.inputs.numpy_input_fn()` operation.:

```
1. import numpy as np
2. # numpy input_fn.
3. my_input_fn = tensorflow.estimator.inputs.numpy_input_fn(

4.     x={"x": np.array(x_data)},
5.     y=np.array(y_data),
6.     ...)
```

All you did is calling the input_fn according to the source of data storage (`numpy_input_fn` for `numpy array` as in this case). Then you passed the feature data to the `x` parameter and the labels to the `y` parameter.

In this example, there is only one feature column called `x` mapped to the tensor `x_data` as in this line:

```
1.     x={"x": np.array(x_data)},
```

If you have multiple feature columns, then you have to create more associations in the dict. One mapping between each feature column and its corresponding Tensor.

Passing `input_fn` Data to Model

To feed data into the mode, you can use the Model train, evaluate, or predict operations. For example, the next line feeds the data into the train operation as the value of the `input_fn` argument:

```
1. model.train(input_fn=my_input_fn)
```

Note that the `input_fn` argument of the train operation accepts a function object (`input_fn=my_input_fn`) not the return value of the function call like that (`input_fn=my_input_fn()`):

```
1. model.train(input_fn=my_input_fn())
```

This means that you won't be able to parameterize the `input_fn` to accept additional parameters. So, the following line of code will generate an error:

```
model.train(input_fn=my_input_fn(my_training_data))
```

Based on that, you have to create a separate `input_fn` for each type of data you want to use in any operation (train, evaluate, or predict). For the same data, if you want to train the model with different subsets you have to use two input functions.

But there are also other ways to parameterize the input functions. For example, you can parameterize the input functions based on:

- Wrapper function
- Lambda
- Python's `functools.partial` function

Wrapper Function

The `input_fn` argument of the train operation accepts a function object but we want to parameterize the operation. To satisfy our requirements, we can build our own parameterized function to accept our custom parameters. To satisfy the train operation requirement, we can use the previous parameterized function into a wrapper function with no parameters. This is illustrated in the following code:

```
import tensorflow

def my_input_fn(dataset, labels):
    return tensorflow.estimator.inputs.numpy_input_fn(
        x = {"f1": dataset},
        y = labels
    )

training_set = tensorflow.zeros(shape=(1, 7))
    labels = tensorflow.zeros(shape=(1, 3))
    def my_input_fn_training_set():
        return my_input_fn(dataset = training_set, labels = labels)

    model.train(input_fn = my_input_fn_training_set())
```

The usefulness of this parameterized input function is that you can create a single input function to work with multiple datasets rather than creating one input function for each data set.
For example, if you want to create a test set, previously you were to create another `input_fn` for the testing set different from the training set as follows:

```
train_set = ...
my_input_fn_training_set = tensorflow.estimator.inputs.numpy_input_fn(
    x = {"f1": train_set},
    y = labels
)

test_set = ...
my_input_fn_training_set = tensorflow.estimator.inputs.numpy_input_fn(
    x = {"f1": test_set},
        y = labels
    )
```

But using the parameterized function, just one `input_fn` is created and called multiple times with different parameters:

```
import tensorflow

def my_input_fn(dataset, labels):
    return tensorflow.estimator.inputs.numpy_input_fn(
        x = {"f1": dataset},
        y = labels
    )

training_set = tensorflow.zeros(shape=(1, 7))
    labels = tensorflow.zeros(shape=(1, 3))
    def my_input_fn_training_set():
        return my_input_fn(dataset = training_set, labels = labels)

    testing_set = tensorflow.zeros(shape=(1, 7))
```

```
    labels2 = tensorflow.zeros(shape=(1, 3))
    def my_input_fn_training_set():
        return my_input_fn(dataset = testing_set, labels
  = labels2)

    model.train(input_fn = my_input_fn_training_set)
    model.evaluate(input_fn = my_input_fn_training_set)
```

Lambda

The previous option for parameterizing the input function already uses a single input function but makes extra code for preparing the wrapping functions. We can avoid such extra code using the `lambda` expression.
We can wrap a parameterized input function inside a `lambda` statement. The result will be assigned to the `input_fn` parameter as follows:

```
def my_input_fn(dataset, labels):
    return tensorflow.estimator.inputs.numpy_input_fn(
        x = {"f1": dataset},
        y = labels
    )

model.train(input_fn = lambda: my_input_fn(training_set,
  labels))
```

The big advantage of using `lambda` is avoiding adding more code. Just the same function can be used with the train, evaluate, and predict operations by just changing the data argument as follows:

```
def my_input_fn(dataset, labels):
    return tensorflow.estimator.inputs.numpy_input_fn(
        x = {"f1": dataset},
        y = labels
    )

model.evaluate(input_fn = lambda: my_input_fn(testing_set
  , labels))
```

This approach enhances code maintainability: no need to define multiple `input_fn` (e.g. `input_fn_train, input_fn_test, input_fn_predict`) for each type of operation.

There are some more parameters to use within the input function `input_fn` inside `tensorflow.estimator.inputs` to control how the `input_fn` iterates over the data:

1. **batch_size**: Patch size to use. Useful when working with Patch Gradient Descent as it requires data to be fed into the model in patches of specific sizes.
2. **num_epochs**: Number of training epochs.
3. **shuffle**: Shuffles the samples. Either True or False. Don't set it to `True` for testing.

Here are all parameters accepted by the `numpy_input_fn`:

```
numpy_input_fn(
    x,
    y=None,
    batch_size=128,
    num_epochs=1,
    shuffle=None,
    queue_capacity=1000,
    num_threads=1
)
```

For example,

```
my_input_fn = tensorflow.estimator.inputs.numpy_input_fn(

    x = {"x": train_input},
    y = train_output,
    batch_size=3,
    num_epochs=500,
    shuffle=True
)
```

Batch Size & Iteration & Epoch

The `tensorflow.estimator.inputs.numpy_input_fn` operation has some arguments such as batch size and epoch that are better to understand their meaning.

Batch Size

Previously, all training data was used to train the model at once. But due to memory limitations, all training samples can't be held in the memory. To solve this problem, the training samples will be divided into a number of patches. Each patch will be fed to the model in one forward/backward pass. The higher the batch size, the more memory space needed.

For example, if we use 10 samples training set with batch size of 2, then there will be (10/2)=5 patches where each patch will have two sample (1-2, 3-4, 5-6, 7-8, and 9-10). Each patch will have one forward pass and one backward pass.

Because some people may say that a batch equal to the entire training samples, we may use the term mini-patch to refer to just a subset of the entire training samples.

Generally, a batch is the number of samples used in one forward/backward pass. The samples may be the complete training set or partial set of the training samples.

Epochs

One epoch is equal to one forward pass and one backward pass across all training samples. For our 10 samples, to be able to pass through the entire set of data we need 5 iterations. The here one epoch is equal to 5 iterations.

In other words, each time the entire training set is fed to the model creates one epoch.

One epoch is equal to the number of iterations to present the entire data to the model.

Iterations

One iteration is one forward pass and backward pass across a single patch. We say that one iteration is equal to one forward pass + one backward pass. We don't say that one forward only or backward only pass is an iteration.

Each time you pass a patch forward and backward through the model creates one iteration.

Example: if you have 1000 training examples, and your batch size is 500, then it will take 2 iterations to complete 1 epoch.

Example:
Say you have a dataset of 10 examples/samples. You have batch size of 2, and you've specified that the algorithm will run for 3 epochs. Therefore, in each epoch, you have 5 batches (10/2 = 5).
There will be 5 iterations per epoch. Since you've specified 3 epochs, you have a total of 15 iterations (5*3 = 15) for training.

Summary:

Pass > Iteration > Epoch

Pass is part of an iteration. An iteration is part of an epoch.

2) Define the Feature Columns

Note that **only feature column names** as string were used in the `input_fn` to create key-value mapping to their values (Tensor).
But this is not everything. We have to create feature columns by specifying their **names** again in addition to their **types** and any pre-processing for the inputs.
There a module called `tensorflow.feature_columns` that hold many functions to create different types of feature columns.
For example, if there is a numeric column in the feature vector then use the appropriate type for it which is numeric. This type is provided by the `tensorflow.feature_columns.numeric_column()` function. The parameters of this function are as follows:

```
numeric_column(
    key,
    shape=(1,),
    default_value=None,
    dtype=tf.float32,
    normalizer_fn=None
)
```

- **key**: It is a unique string identifying the column. It can has multiple uses such as column name and dictionary key in the `input_fn`.

- **shape**: Shape of the Tensor.
- **default_value**: A default value of the specified data type to be used if the value is missing in the input data.
- **dtype**: Default type of values.
- **normalizer_fn**: A function to normalize the value of the Tensor associated with that column. It can be used for any TensorFlow transformations for data. This function accepts the input Tensor and returns the output after the specified transformations in a lambda statement for example. (e.g. `lambda x: (x - 3.0) / 4.2`).

The following code creates two numeric feature columns of type `float32`:

```
1. age_feature = tensorflow.feature_column.numeric_column(ke
   y="age", dtype=tensorflow.float32)
2. height_feature = tensorflow.feature_column.numeric_column
   (key="height", dtype=tensorflow.float32)
```

Note that the feature columns are used in different positions:

- **read_csv**: feature column name only
- **input_fn**: feature column name to its value (Tensor)
- **Pre-made Estimator Call**: feature column name only

3) Instantiate the Pre-made Estimator

Instantiate the pre-made estimator. For example, we can create a very simple linear classifier that only accepts the feature columns.

Example:

```
1. estimator = tensorflow.estimator.LinearClassifier(feature
   _columns=[age_feature, height_feature])
```

4) Call Training, Evaluation, or Inference Operations

```
1. estimator.train(input_fn=my_training_set)
```

Complete LinearRegression Example:

```
import tensorflow
import numpy

#Reading data in NumPy arrays.
train_input = numpy.array([1, 2, 3, 4])
train_output = numpy.array([-1, -2, -3, -4])
evaluate_input = numpy.array([-1, -2, -3, -4])
evaluate_output = numpy.array([1, 2, 3, 4])

    #Converting training features data to Tensors
    my_input_fn = tensorflow.estimator.inputs.numpy_input_fn(
        x = {"x": train_input},
        y = train_output,
        batch_size=2,
        num_epochs=500,
        shuffle=True
    )

    #Converting evaluation features data to Tensors
    my_input_fn_evaluate = tensorflow.estimator.inputs.numpy_input_fn(
        x = {"x": evaluate_input},
        y = evaluate_output,
        shuffle=False
    )

    #Preparing the feature columns
    x = tensorflow.feature_column.numeric_column("x")

    #Intializing the Estimator
    estimator = tensorflow.estimator.LinearRegressor(feature_columns=[x])

    #Training the Estimator
```

```
33.     estimator.train(my_input_fn, steps=1000)
34.
35.     #Evaluating the Estimator
36.     train_metrics = estimator.evaluate(my_input_fn)
37.     evaluate_metrics = estimator.evaluate(my_input_fn_eva
  luate)
38.
39.     #Printing Loss of Evaluation
40.     print(train_metrics)
41.     print(evaluate_metrics)
```

The return is:

```
• {'average_loss': 1.451981e-05, 'loss': 2.903962e-
  05, 'global_step': 1000}
• {'average_loss': 0.00031257086, 'loss': 0.0012502834, 'gl
  obal_step': 1000}
```

Important:

There must be a match between the feature column names used as keys in the features dictionary of the `input_fn` and the feature columns names used to train the `Estimator` as in lines 12 and 27 of the previous example.

If you tried to use a feature column name not existing in the features dictionary, there will be an error returned. For example, the following code changes the name of the feature column used in line 27 from x to k:

```
26.     #Preparing the feature columns
27.     x = tensorflow.feature_column.numeric_column("k")
```

Here is the error message:

```
• ValueError: Feature k is not in features dictionary.
```

Chapter 3

pandas

Previously, we read small amounts of data that are very simple to be stored in a NumPy arrays. But what about large amounts of data? For working easily with larger data, there is a Python library called pandas which is an open source high-level library for data analysis and manipulation.
Its development began by Wes McKinney in April 2008 as part of the PyData project. It became open-source by the end of 2009. pandas allows flexible data manipulation through its data structures (Series [1D], DataFrames [2D], and Panels [3D & 4D]). DataFrame is a container for Series and Panel is a container for DataFrame objects (big data structure can hold small data structure). All of them are built on top of NumPy and this guarantees their speed. pandas data structures allow to easily insert, update, and delete objects in a way similar to dictionary.

Benefits of using pandas

Data Orientation

Working with the NDarrays adds more overhead over the user to consider the data orientation. One goal of pandas is to reduce the mental effort wasted thinking about data orientation.
Working with ndarray is like a graph that has no labels for the X and Y axes. You don't know what X and Y axes represent. This is ambiguous.
But using pandas, there will be a label printed to make you sure what rows and columns represent to remove the ambiguity.

Data Visualization

pandas created more readable data than in NumPy arrays. You can print the pandas data structures and you will find it easy to understand due to the labels printed across the rows and columns.

Mutability

All pandas data structures are value-mutable and that means we can change the values stored but they are not always size-mutable. For example, pandas Series are not size-mutable meaning their size can't be changed after created. But in pandas DataFrame, a new column can be added or removed.

Series

pandas Series is a one-dimensional labeled array that can hold any data type (integers, floating-point numbers, strings, objects, and more). The labels of the axis are called **index**.
There are different ways to create a pandas Series but the most basic way is using the pandas.Series method. Note that pandas must be imported before getting used.

```
1. import pandas
2. s = pandas.Series(data=None, index=None, dtype=None, copy
   =False)
```

The data parameter here can be different things:

- Python dict
- ndarray
- Scalar Value
- Another pandas Series

The index is a list of axis labels.
If the dtype is None then its type is inferred from the data.

Series by ndarray

If data parameter is a ndarray, then index must be the same length as data. If no index passed, then an index will be created in this range [0, … ,len(data)-1].

```
1. import pandas
2. import numpy
3. s = pandas.Series(data=numpy.array((0, 1, 2, 3, 4)), inde
   x=['a', 'b', 'c', 'd', 'e'])
```

When printing the Series:

```
In [6]: s
Out[6]:
a    0
b    1
c    2
d    3
e    4
dtype: int32
```

You can also use other methods that return a Python list or an `ndarray`.

```
4. s = pandas.Series(data=numpy.arange(5), index=['a', 'b',
   'c', 'd', 'e'])
5. s = pandas.Series(data=numpy.rand(5), index=['a', 'b', 'c
   ', 'd', 'e'])
6. s = pandas.Series(data=[0, 1, 2, 3,
   4], index=['a', 'b', 'c', 'd', 'e'])
```

The index can be accessed using the `index` attribute:

```
1. s.index
```

```
In [7]: s.index
Out[7]: Index(['a', 'b', 'c', 'd', 'e'], dtype='object')
```

When creating the `pandas Series` without explicitly creating index, a default index is created:

```
1. s = pandas.Series(data=numpy.arange(5))
```

```
In [9]: s
Out[9]:
0    0
1    1
2    2
3    3
4    4
dtype: int32
```

Series by dict

When pandas `Series` data parameter is assigned using a dict without passing an `index`, then the keys of the dict will be the indices in the same order as in the dict.

```
1. import pandas
2.
3. d = {'a': 0, 'b': 1, 'c': 2, 'd': 3, 'e': 4}
4. s = pandas.Series(data=d)
```

```
In [11]: s
Out[11]:
a    0
b    1
c    2
d    3
e    4
dtype: int64
```

We can also reorder the keys by explicitly passing the index with the desired order:

```
1. import pandas
2.
3. d = {'a': 0, 'b': 1, 'c': 2, 'd': 3, 'e': 4}
4. s = pandas.Series(data=d, index=['b', 'e', 'c', 'a', 'd']
   )
```

```
In [13]: s
Out[13]:
b    1
e    4
c    2
a    0
d    3
dtype: int64
```

What if one of the inserted indices was not existing as a key in the dictionary? It will be added but its value will be NaN meaning there is a missing value here.

```
1. import pandas
2.
3. d = {'a': 0, 'b': 1, 'c': 2, 'd': 3, 'e': 4}
```

```
4. s = pandas.Series(data=d, index=['b', 'e', 'c', 'a', 'd',
   'g'])
```

```
In [15]: s
Out[15]:
b    1.0
e    4.0
c    2.0
a    0.0
d    3.0
g    NaN
dtype: float64
```

Series by scalar

If a scalar value was provided for the data parameter, then the **index must be given**. The scalar will be repeated to match the length of the index.

```
1. s = pandas.Series(data=4.6, index=['b', 'e', 'c', 'a', 'd
   ', 'g'])
2. s = pandas.Series(data=4.6, index=numpy.arange(7))
```

```
In [17]: s
Out[17]:
b    4.6
e    4.6
c    4.6
a    4.6
d    4.6
g    4.6
dtype: float64
```

Series is ndarray-like

`Series` is very like to `ndarray` and may be a valid argument in methods accepting `ndarray`. But slicing the `Series` also slices the `index`.

```
In [27]: s[0]
Out[27]: 0

In [28]: s[:3]
Out[28]:
a    0
b    1
c    2
dtype: int32
```

We can also use multiple `indices` and get them returned in the specified order.

```
In [29]: s[[4, 2, 0]]
Out[29]:
e    4
c    2
a    0
dtype: int32
```

Similar to using `ndarray`, we can apply operations over the `Series` data elements and slices them based on a condition.
We don't have to make a loop through the `Series` to apply operations. Operations on `Series` can be applied using the same `NumPy` methods.

```
In [32]: numpy.power(s, 2)
Out[32]:
a     0
b     1
c     4
d     9
e    16
dtype: int32

In [33]: s[s >= 2]
Out[33]:
c    2
d    3
e    4
dtype: int32
```

```
In [53]: s*4
Out[53]:
a     0
b     4
c    60
d    72
e    16
dtype: int32

In [54]: s+s
Out[54]:
a     0
b     2
c    30
d    36
e     8
dtype: int32
```

Operations between `Series` are the union of their indices labels. If a label is not existing in one `index` but in the other one, then it will be returned in the result but with a value of `NaN` meaning that it was missing in one Series.

```
In [55]: s[2:]+s[3:]
Out[55]:
c     NaN
d    36.0
e     8.0
dtype: float64
```

The Series elements can have their value queried and changed like `ndarray` using the `index` order.

```
In [38]: s[2] = 15

In [39]: s
Out[39]:
a     0
b     1
c    15
d     3
e     4
dtype: int32
```

Series is dict-like

Like dict, we can query and edit a `Series` using the `index` label. We can also check if a label exists or not in the Series.

```
In [40]: s['a']
Out[40]: 0

In [41]: s['d'] = 18

In [42]: s
Out[42]:
a     0
b     1
c    15
d    18
e     4
dtype: int32

In [43]: 'f' in s
Out[43]: False

In [44]: 'u' is s
Out[44]: False
```

If we tried to index the Series in a dict-style using the index label and the label is not existing, then an exception is raised.

```
>>> s['f']
KeyError: 'f'
```

To avoid raising exceptions for missing labels, we can use the get method. We can also make it return a default value if the label is missing.

```
In [51]: s.get('t')

In [52]: s.get('t', numpy.nan)
Out[52]: nan
```

pandas DataFrame

`pandas DataFrame` is a 2-dimensional labeled data structure with columns of potentially different types. It can be regarded a SQL table or a dict of `Series` objects. It is the commonly used pandas object.

`pandas DataFrame` can be created using different ways:

- Dict of 1D `ndarrays`, lists, dicts, or `Series`
- 2D `NumPy ndarray`
- Structured or record `ndarray`
- `Series`
- Another `DataFrame`

Its general structure is as follows:

```
1. import pandas
2. d = pandas.DataFrame(
3.           data=None, index=None,
   columns=None, dtype=None, copy=False
4. )
```

In addition to the data parameter, we can pass the `index` (row labels) and `columns` (column labels) parameters.

DataFrame by dict of Series or dicts

The resultlting `index` will be the union of indices used in all `Series`. If one `index` label was existing in one `Series`/dict but not in another, then its value will be NaN in the `Series`/dict in which this label is missing.

Using Series

```
1. import pandas
2.
3. s1 = pandas.Series(data=[1, 2, 3], index=['a', 'b', 'c'])

4. s2 = pandas.Series(data=[11, 12, 13], index=['c', 'd', 'e
   '])
```

```
5.
6. d = {"one": s1,
7.      "two": s2}
8.
9. df = pandas.DataFrame(data=d)
```

OR

```
1. import pandas
2.
3. d = {"one": pandas.Series(data=[1, 2, 3], index=['a', 'b'
   , 'c']),
4.      "two": pandas.Series(data=[11, 12, 13], index=['c',
   'd', 'e'])}
5.
6. df = pandas.DataFrame(data=d)
```

Using dicts

```
1. import pandas
2.
3. d1 = {"one": {'a': 1, 'b': 2, 'c': 3}}
4. d2 = {"two": {'c': 11, 'd': 12, 'e': 13}}
5.
6. d = {"one": d1,
7.      "two": d2}
8.
9. df = pandas.DataFrame(data=d)
```

OR

```
1. import pandas
2.
3. d = {"one": {'a': 1, 'b': 2, 'c': 3},
4.      "two": {'c': 11, 'd': 12, 'e': 13}}
5.
6. df = pandas.DataFrame(data=d)
```

```
In [59]: df
Out[59]:
   one   two
a  1.0   NaN
b  2.0   NaN
c  3.0  11.0
d  NaN  12.0
e  NaN  13.0
```

The index and columns can be accessed using the index and columns attributes, respectively:

```
1. s.index
2. s.columns
```

```
In [67]: df.index
Out[67]: Index(['a', 'b', 'c', 'd', 'e'], dtype='object')

In [68]: df.columns
Out[68]: Index(['one', 'two'], dtype='object')
```

We can specify which indices to get returned by explicitly adding an `index` argument:

```
1. d = {"one": pandas.Series(data=[1, 2, 3], index=['a', 'b'
   , 'c']),
2.      "two": pandas.Series(data=[11, 12, 13], index=['c',
   'd', 'e'])}
3.
4. df = pandas.DataFrame(data=d, index=['c', 'd', 'a'])
```

```
In [70]: df
Out[70]:
   one   two
c  3.0  11.0
d  NaN  12.0
a  1.0   NaN
```

Also, we can order the columns. If a column label was not existing in of the `Series` or dicts, it will be inserted but all its values will be NaN. This is like the newly added column with label `three`.

```
df = pandas.DataFrame(data=d, index=['c', 'd', 'a']
  , columns=['two', 'one', three])
```

```
In [75]: df
Out[75]:
    two  one three
c  11.0  3.0   NaN
d  12.0  NaN   NaN
a   NaN  1.0   NaN
```

From dict of ndarrays/lists

The ndarrays must all have the same length. If two different ndarrays used with different lengths, then an error will occur.

```
import pandas
import numpy

d = {"one": numpy.ones((4)),
     "two": [4, 7, 8, 9]}

df = pandas.DataFrame(data=d)
```

```
In [83]: df
Out[83]:
   one  two
0  1.0    4
1  1.0    7
2  1.0    8
3  1.0    9
```

```
import pandas
import numpy

d = {"one": numpy.ones((4)),
     "two": [4, 7, 8, 9]}

df = pandas.DataFrame(data=d, index=['a', 'b', 'c', 'd'])
```

```
In [86]: df
Out[86]:
   one  two
a  1.0    4
b  1.0    7
c  1.0    8
d  1.0    9
```

From a list of dicts

```
1. data2 = [{'a': 1, 'b': 2}, {'a': 5, 'b': 10, 'c': 20}]
2. data2 = [{'a': 1, 'b': "Ahmed"}, {'a': 5, 'b': "Mohamed"}
   ]
3. df = pandas.DataFrame(data=d)
```

```
In [88]: df
Out[88]:
   a        b
0  1    Ahmed
1  5  Mohamed
```

DataFrame can be created using constructors

DataFrame.from_dict

It just takes a dict of dicts like done before. The only difference is that such constructor accepts a new parameter called "`orient`" to make the dict keys (`columns` labels) as row keys. Its default value is `columns` in which the `columns` labels will be at the columns and the `index` labels will be in rows. As a summary, the columns labels will be set according to the input to the orient argument. If `orient='index'`, then `index` (rows) labels will be at columns.

```
1. import pandas
2.
3. data2 = {"one": {'a': 1, 'b': "Ahmed"},
4.          "two": {'a': 5, 'b': "Mohamed"}}
5.
6. df = pandas.DataFrame.from_dict(data=data2, orient="colum
   ns")
```

```
In [92]: df
Out[92]:
      one       two
a       1         5
b   Ahmed   Mohamed
```

from_items

This constructor accepts a sequence of the form (key, value) pairs. The keys are the column labels in cases that the `orient` argument is set to columns (`orient='columns'`). The default is `orient='columns'` like `pandas.DataFrame.from_dicts`.
If `orient='index'`, then keys are for rows.
The columns parameter in the `from_items` constructor can only be used if `orient='index'`.

```
1. import pandas
2.
3. df = pandas.DataFrame.from_items(items=[('A', [1, 2, 3]),
   ('B', [5, 6, 7])])
```

```
In [96]: df
Out[96]:
   A  B
0  1  5
1  2  6
2  3  7
```

In case that `orient='index'`, then the keys will be labels for rows and columns will not have any label. In this case, we must use the columns argument.

```
1. import pandas
2.
3. df = pandas.DataFrame.from_items(items=[('A', [1, 2, 3]),
   ('B', [5, 6, 7])], orient="index", columns=["One", "Two",
   "Three"])
```

```
In [6]: df
Out[6]:
   One  Two  Three
A    1    2      3
B    5    6      7
```

Column selection, addition, and deletion

Column Selection

pandas `DataFrame` can be treated like a dict or indexed Series objects. Setting, getting, and deleting columns is the same as dict.

```
In [18]: df['One']['A']
Out[18]: 1

In [19]: df['One']
Out[19]:
A    1
B    5
Name: One, dtype: int64

In [20]: df['Three'] = df['One'] + df['Two']

In [21]: df
Out[21]:
   One  Two  Three
A    1    2      3
B    5    6     11
```

```
In [22]: df['Four'] = df['Three'] + 6

In [23]: df
Out[23]:
   One  Two  Three  Four
A    1    2      3     9
B    5    6     11    17
```

Delete Columns

```
In [24]: del df['Three']

In [25]: df.pop("One")
Out[25]:
A    1
B    5
Name: One, dtype: int64

In [26]: df
Out[26]:
   Two  Four
A    2     9
B    6    17
```

Accessing Items

```
In [29]: df['Two']['A']
Out[29]: 2


In [32]: df['Two'][:2]
Out[32]:
A    7.9
B    7.9
Name: Two, dtype: float64
```

Assigning a value to a column will change the entire values within that column to the new value.

```
In [30]: df['Two'] = 7.9

In [31]: df
Out[31]:
   Two  Four
A  7.9     9
B  7.9    17
```

Previously, we are able to index the columns by the label of column within the square brackets `[]`. For example:

```
1. df['One']
```

Regarding rows, the `[]` won't be able to index rows by index labels. They can only work for slicing the rows not by the index label but using the index label location. For example, the next line returns the rows with the index label locations `0` and `1`.

```
1. df[:2]
```

But we can't index rows using `[]`. The next line will give an error:

```
1. df[1]
```

Also, we can't index the row by its index label using `[]`. So, the next line will get an error:

```
1. df['A']
```

The previous line is used for column indexing by label. It searches for a column with name `'A'` and when failed it returns an error.

DataFrame Row Indexing

But what if we want to index the row by its index label or by its index label location? We have to use methods for that task:

- `iloc`: Supports row integer indexing (row index label location).
- `loc`: Supports row index label indexing.

iloc

`.iloc[]` is an integer-location based indexing for selection by position (from 0 to length-1 of the axis).
Allowed inputs are:

- An integer, e.g. 5. `df.iloc[2]`
- A list or array of integers, e.g. [4, 3, 0]. `df.iloc[[0, 1, 2]]`
- A slice object with ints, e.g. 1:7. `df.iloc[0:2]`
- A boolean array. `df.iloc[[False, True]]. Return the second row only.`

For the following `DataFrame`:

```
In [34]: df
Out[34]:
       a        b
one    1    Ahmed
two    5  Mohamed
```

Row integer indexing using `iloc`:

```
In [48]: df.iloc[[0]]
Out[48]:
       a      b
one    1  Ahmed
```

Accessing individual elements:

```
In [83]: a = df.iloc[0]

In [84]: a[0]
Out[84]: 1

In [85]: a[1]
Out[85]: 'Ahmed'
```

loc

`loc` is similar to `iloc` but it is label-based, not location-based.
Allowed inputs are:

- A single label, e.g. 5 or `'a'`, (note that 5 is interpreted as a label of the index, and never as an integer position along the index). **`df.loc[5]`** OR **`df.loc['a']`**
- A list or array of labels, e.g. `['a', 'b', 'c']`. **`df.loc[['a', 'b', 'c']]`**
- A slice object with labels, e.g. `'a':'f'` (note that contrary to usual python slices, both the start and the stop are included!). **`df.loc['a':'f']`**
- A boolean array. **`df.loc[[True, False]]`**

Row label indexing using `loc`:

```
In [52]: df.loc[['one']]
Out[52]:
      a       b
one   1   Ahmed
```

Accessing individual values:

```
In [79]: a = df.loc['one']

In [80]: a[0]
Out[80]: 1

In [81]: a[1]
Out[81]: 'Ahmed'
```

Insert new row

We can also use `loc` to insert a new row:

```
In [97]: df.loc['three'] = [2, 'Gad']

In [98]: df
Out[98]:
       a        b
one    1    Ahmed
two    5  Mohamed
three  2      Gad
```

Slicing Rows

Slicing by row index label location using `[]`:

```
In [40]: df[:1]
Out[40]:
     a      b
one  1  Ahmed
```

Indexing by column label:

```
In [43]: df['a']
Out[43]:
one    1
two    5
Name: a, dtype: int64
```

Inserting Columns

By default, columns get inserted at the end. The insert function is available to insert at a particular location in the columns.
General structure of the inset method is as follows:

```
1. DataFrame.insert(loc, column, value, allow_duplicates=Fal
   se)
```

For example:

```
1. df.insert(1, 'bar', df['Two'])
```

The value of the new column labeled `'bar'` will be the same as column labeled `'Two'`.
The new column will be inserted at index 1.

```
In [36]: df.insert(1, 'bar', df['Two'])

In [37]: df
Out[37]:
   Two  bar  Four
A  7.9  7.9     9
B  7.9  7.9    17
```

Comma-Separated Values (CSV) Files

A pandas DataFrame can be saved externally as a CSV file using the `to_csv()` method.

```
1. DataFrame.to_csv(path_or_buf=None, sep=', ', …)
```

From the important parameters:

- `path_or_buf`: The path including file name of the CSV file to save the DataFrame into.
- `sep`: The separator between the columns within the single row. It is by default comma.

Example:

```
1. import pandas
2.
3. data2 = {"one": {'a': 1, 'b': "Ahmed"},
4.          "two": {'a': 5, 'b': "Mohamed"}}
5.
6. df = pandas.DataFrame.from_dict(data=data2, orient="index
   ")
7. df.to_csv("test.csv")
```

Here is the data in the CSV file:

```
,a,b
one,1,Ahmed
two,5,Mohamed
```

Here is the DataFrame after being printed:

```
In [111]: df
Out[111]:
   a          b
0  1      Ahmed
1  2   Mohamed
```

Note that we can create a CSV formatted data manually but take into regard the following:

- Use comma `','` between each two columns.
- Add columns label in the first row inserted.
- Use \n to move to the next row.

Here is a string formatted in the CSV file format and converted into a pandas DataFrame.

```
1. import pandas
2. import io
3.
4. data = "a,b\n1, Ahmed\n2, Mohamed"
5.
6. df = pandas.read_csv(io.StringIO(data))
7. print(df)
```

The `io.StringIO` encodes the String so that `pandas.read_csv` can read it. `io.StringIO` uses `'\n'` as its default new line separator.

```
io.StringIO(initial_value='', newline='\n')
```

Reading CSV File

To read a previously created CSV file, use the `pandas.read_csv` method.

```
1. import pandas
2.
3. df = pandas.read_csv("test.csv")
4. print(df)
```

```
In [115]: df
Out[115]:
   a          b
0  1      Ahmed
1  5   Mohamed
```

To return the columns and rows labels:

```
1. df.columns
2. df.index
```

To get a specific value, use indexing as follows:

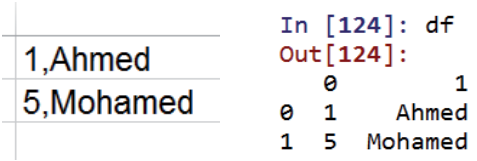
```
1. df.columns[2]
2. df.index[7]
```

Read CSV with no Header

The previously read CSV file uses the first row as the columns labels assuming that the file has a columns labels inserted as the first row. If you have not saved the CSV file with columns labels inserted as the first row in the file and would like to tell pandas that the first row is an actual row data and not a header for columns, use the `header` argument with value None.

```
1. df = pd.read_csv('test.csv', header=None)
```

This will make pandas interpret the first row in the CSV file as actual data. In this case, the columns labels will be numeric values starting from 0.



Reading Data using pandas DataFrame

After being able to create, index, and update Series and DaraFrame data structures in pandas, next is to use them in TensorFlow.

Previously, `tensorflow.estimator.inputs.numpy_input_fn()` operation was used for reading data from a NumPy array. Using pandas DataFrame rather than NumPy array for reading data is very simple. The only difference is using the `tf.estimator.inputs.pandas_input_fn()` operation that works with pandas DataFrame. Its structure is as follows:

```
pandas_input_fn(
    x,
    y=None,
    batch_size=128,
    num_epochs=1,
    shuffle=None,
    queue_capacity=1000,
    num_threads=1,
    target_column='target'
     )
```

Chapter 4

Build FFNN using TensorFlow

In this article, two basic feed-forward neural networks (FFNNs) will be created using TensorFlow deep learning library in Python. The reader should have basic understanding of how neural networks work and its concepts in order to apply them programmatically.
This article will take you through all steps required to build a simple feed-forward neural network in TensorFlow by explaining each step in details. Before actual building of the neural network, some preliminary steps are recommended to be discussed.

The summarized steps are as follows:

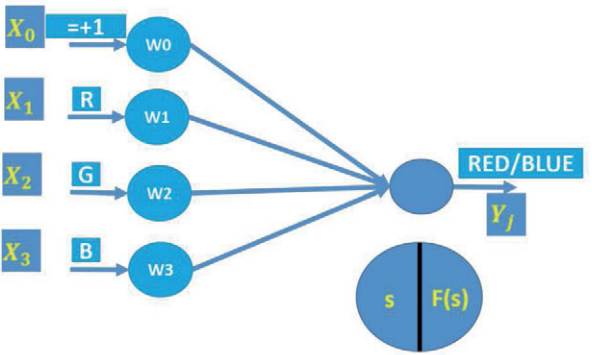
1. Reading the training data (inputs and outputs)
2. Building and connect the neural networks layers (this included preparing weights, biases, and activation function of each layer)
3. Building a loss function to assess the prediction error
4. Create a training loop for training the network and updating its parameters
5. Applying some testing data to assess the network prediction accuracy

Here is the classification problem that we are to solve using neural network.

	R (RED)	G (GREEN)	B (BLUE)
RED	255	0	0
	248	80	68
BLUE	0	0	255
	67	15	210

It is a binary classification problem to classify colors into either red or blue based on the three RGB color channels. It can be solved linearly and thus we don't have to use hidden layers. Just input and output layers are to be used. There will be a single

neuron in the output layer with an activation function. The network architecture is shown in the following figure (**Figure 1**):



Where $X_0 = 1$ is the bias and W_0 is its weight. W_1, W_2, and W_3 are the weights for the three inputs R (Red), G (Green), and B (Blue).

Here is the complete code of the neural network solving that problem to be discussed later. For easy access, this code is called **CodeSample1**.

```
1. import tensorflow
2.
3. # Preparing training data (inputs-outputs)
4. training_inputs = tensorflow.placeholder(shape=[None, 3],
   dtype=tensorflow.float32)
5. training_outputs = tensorflow.placeholder(shape=[None, 1]
   , dtype=tensorflow.float32) #Desired outputs for each inp
   ut
6.
7. # Preparing neural network parameters (weights and bias)
   using TensorFlow Variables
8. weights = tensorflow.Variable(initial_value=[[.3], [.1],
   [.8]], dtype=tensorflow.float32)
9. bias = tensorflow.Variable(initial_value=[[1]], dtype=ten
   sorflow.float32)
10.
11.     # Preparing inputs of the activation function
12.     af_input = tensorflow.matmul(training_inputs, weights
   ) + bias
```

```
13.
14.     # Activation function of the output layer neuron
15.     predictions = tensorflow.nn.sigmoid(af_input)
16.
17.     # Measuring the prediction error of the network after
  being trained
18.     prediction_error = tensorflow.reduce_sum(training_out
  puts - predictions)
19.
20.     # Minimizing the prediction error using gradient desc
  ent optimizer
21.     train_op = tensorflow.train.GradientDescentOptimizer(
  learning_rate=0.05).minimize(prediction_error)
22.
23.     # Creating a TensorFlow Session
24.     sess = tensorflow.Session()
25.
26.     # Initializing the TensorFlow Variables (weights and
  bias)
27.     sess.run(tensorflow.global_variables_initializer())
28.
29.     # Training data inputs
30.     training_inputs_data = [[255, 0, 0],
31.                             [248, 80, 68],
32.                             [0, 0, 255],
33.                             [67, 15, 210]]
34.
35.     # Training data desired outputs
36.     training_outputs_data = [[1],
37.                              [1],
38.                              [0],
39.                              [0]]
40.
41.     # Training loop of the neural network
42.     for step in range(10000):
43.         sess.run(fetches=[train_op], feed_dict={
44.
  training_inputs: training_inputs_data,
```

```
45.                                         training_outputs:
  training_outputs_data})
46.
47.     # Class scores of some testing data
48.     print("Expected
  Scores : ", sess.run(fetches=predictions, feed_dict={trai
  ning_inputs: [[248, 80, 68],
                                [0, 0, 255]]}))
49.
50.     # Closing the TensorFlow Session to free resources
51.     sess.close()
```

Reading the Training Data

The data is read in the previous code in lines 4 and 5 using something called placeholder. But what is a placeholder? Why have we not just used a NumPy array for preparing the data? To answer these questions, we can explore a simpler example that reads some inputs and prints it to the console as follows:

```
1. import tensorflow
2.
3. # Creating a NumPy array holding the input data
4. numpy_inputs = [[5, 2, 13],
5.                 [7, 9, 0]]
6.
7. # Converting the NumPy array to a TensorFlow Tensor
8. # convert_to_tensor() doc: https://www.tensorflow.org/api
  _docs/python/tf/convert_to_tensor
9. training_inputs = tensorflow.convert_to_tensor(value=nump
  y_inputs, dtype=tensorflow.int8)
10.
11.     # Creating a TensorFlow Session
12.     sess = tensorflow.Session()
13.
14.     # Running the session for evaluating the previously c
  reated Tensor
```

```
15.     print("Output is : ", sess.run(fetches=training_input
  s))
16.
17.     # Closing the TensorFlow Session
18.     sess.close()
```

The input is read into a NumPy array away from TensorFlow as in line 4. But TensorFlow just knows Tensors and just we have to convert the NumPy array into a Tensor. The tensorflow.convert_to_tensor() TensorFlow operation does that conversion as in line 9. To be able to print the contents of a Tensor, we must at first create a Session using the tensorflow.Session() class as in line 12. In line 15, the session runs in order evaluate the Tensor training_inputs and get its values printed. Finally, the session got closed in line 18. The result of printing is as follows:

```
• Output is :  [[ 5  2 13]
•               [ 7  9  0]]
```

This example doesn't use placeholders. So, what is the use of a TensorFlow placeholder? Assume that we want to run the session with another input. To do that, we have to modify the numpy_input Python variable each time a new input is applied.

```
• numpy_inputs = [[83, 49, 92],
•                 [31, 78, 60]]
```

It is not a good way to modify the code in order to get different inputs. A better way for doing that is to just create the Tensor and then modify its value without modifying it in the code. This is the job of the TensorFlow placeholder. Placeholder in TensorFlow is a way for accepting the input data. It is created in the code and modified multiple times in the Session running time. The following code modifies the previous code to use placeholders:

```
1. import tensorflow
2.
3. # Create a placeholder with data type int8 and shape 2x3.

4. training_inputs = tensorflow.placeholder(dtype=tensorflow
   .int8, shape=(2, 3))
```

```
5.
6. # Creating a TensorFlow Session
7. sess = tensorflow.Session()
8.
9. # Running the session for evaluating assigning a value to
   the placeholder
10.     print("Output is : ", sess.run(fetches=training_input
   s,
11.                             feed_dict={training_inputs: [[5
   , 2, 13],
12.                                                          [7
   , 9, 0]]}))
13.
14.     # Closing the TensorFlow Session
15.     sess.close()
```

This code prints the same outputs as before but it uses a placeholder as in line 4. The `placeholder` is created by specifying the data type and the shape of the data it will accept. The shape can be specified to restrict the input data to be of specific size. If no shape specified, then different inputs with different shapes can be assigned to the placeholder. The placeholder is assigned a value when running the `Session` using the `feed_dict` argument of the `run` operation. `feed_dict` is a dictionary used to initialize the placeholders.

But assume there is a feature vector of 50 feature and we have a dataset of 100 samples. Assume we want to train a model two times with a different number of samples, say 30 and 40. Here the size of the training set has one dimension fixed (number of features=number of columns) and another dimension (number of rows=number of training samples) of variable size. Setting its size to 30, then we restrict the input to be of size `(30, 50)` and thus we won't be able to re-train the model with 40 samples. The same holds for using 40 as the number of rows. The solution is to just set the number of columns but leave the number of rows unspecified by setting it to None as follows:

```
• # Create a placeholder with data type int8 and shape Rx3.

• training_inputs = tensorflow.placeholder(dtype=tensorflow
  .int8, shape=(None, 50))
```

One benefit of using placeholder is that its value is modified easily. You have not to modify the program in order the use different inputs. It is similar to a variable in Java, C++, or Python but it is not exactly a variable in TensorFlow. We can run the session multiple times with different values for the placeholder:

```
# Running the session for evaluating assigning a value to the placeholder
print("Output is : ", sess.run(fetches=training_inputs,
                       feed_dict={training_inputs: [[5, 2, 13],
                                                    [7, 9, 0]]}))
print("Output is : ", sess.run(fetches=training_inputs,
                       feed_dict={training_inputs: [[1, 2, 3],
                                                    [4, 5, 6]]}))
print("Output is : ", sess.run(fetches=training_inputs,
                       feed_dict={training_inputs: [[12, 13, 14],
                                                    [15, 16, 17]]}))
```

To do that using NumPy arrays we have to create a new Python array for each new input we are to run the program with.
This is why we are using placeholders for feeding the data. For every input, there should be a separate placeholder. In out neural network, there are two inputs which are training inputs and training outputs and thus there should be two placeholders one for each as in lines 4 and 5 in **CodeSample1**.

```
# Preparing training data (inputs-outputs)
training_inputs = tensorflow.placeholder(shape=[None, 3], dtype=tensorflow.float32)
training_outputs = tensorflow.placeholder(shape=[None, 1], dtype=tensorflow.float32) #Desired outputs for each input
```

Note that the size of these placeholders is not fixed to allow a variable number of training samples to be used with the code unchanged. But both placeholders of inputs and outputs training data must have the same number of rows. For example, according to our currently presented training data, `training_inputs` should have a `shape=(4, 2)` and `training_outputs` should be of `shape=(4, 1).`

Preparing the Neural Network Layers and their Parameters

Our example is very simple that has no hidden layers and just there is a single neuron in the output layer. So, we are going to explain how to create such layer and prepare its parameters and create its neuron's activation function which is sigmoid in our case.

There is a problem in placeholders. Placeholder value can't be changed once assigned. After it is given a value then placeholder can be regarded a constant. Thus it is not the suitable option for trainable parameters like ones used in this example (weight and bias). The trainable parameter is assigned an initial value and that value got changed until reaching the best value making the underlying model produce least errors.
For our neural network, there are two trainable parameters which are weight and bias. These parameters are not suitable for being stored in placeholders as we want to update them until getting their best values. This is why there is something in TensorFlow called Variable.
A `TensorFlow Variable` is very similar to variables in `Java`, `C++`, `Python`, and any language with the concept of variables. You can assign an initial value to a `TensorFlow Variable` and that value can get changed multiple times. To create a `TensorFlow Variable`, you must specify its data type and shape. The following example shows how to create a variable rather replacing the previous example`s placeholder:

```
1. import tensorflow
2.
3. # Create a Variable with data type int8 and shape 2x3.
4. training_inputs = tensorflow.Variable(initial_value=[[5,
   2, 13],
5.                                                      [7,
   9, 0]], dtype=tensorflow.int8)
6.
7. # Creating a TensorFlow Session
```

```
8. sess = tensorflow.Session()
9.
10.     # Initialize all Variables
11.     sess.run(tensorflow.global_variables_initializer())
12.
13.     # Running the session for evaluating assigning a valu
  e to the placeholder
14.     print("Output is : ", sess.run(fetches=training_input
  s))
15.
16.     # Closing the TensorFlow Session
17.     sess.close()
```

The changes compared to the previous placeholder code are in lines 4, 11, and 14. The variable is created in line 4 by specifying its initial value and data type. The `dtype` argument is not required but the `initial_value` argument must be specified. The `initial_value` argument specifies both the size and the data type (if `dtype` is missing). Note that `Variable` is class but `placeholder` is an operation.
Line 14 prints the value of the variable. Note that the `Variable` won't be initialized until calling the `global_variables_initializer()` operation as in line 11 in order to make the variable actually initialized. Trying to use the variable without calling this operation will return an error.
In **CodeSample1**, there are two variables created for our two parameters: weight (line 8) and bias (line 9).

```
• # Preparing neural network parameters (weights and bias)
  using TensorFlow Variables
• weights = tensorflow.Variable(initial_value=[[.3], [.1],
  [.8]], dtype=tensorflow.float32)
• bias = tensorflow.Variable(initial_value=[[1]], dtype=ten
  sorflow.float32)
```

Parameters Initial Values

Note that they both weight and bias have initial values. We are using fixed initial values for them. The initial values are set randomly and there is no rule used for generating them. You may use any random number generation operation in TensorFlow for doing that such as `tensorflow.truncated_normal()`. But note

that the initial values are critical for creating a robust model able to predict the right class after being trained. Bad initial values for weights and bias of a neural network can make its neurons to die. This is why there are many techniques used to generate such initial values.

As a summary, a `placeholder` is used to store input data that are to be initialized once and used multiple times. But `Variable` is used for trainable parameters that are to be changed multiple times after being initialized.

Activation Function

After preparing all parameters required by the output layer's neuron, we are ready for using the activation function as in line 15 of **SampleCode1**.

```
# Activation function of the output layer neuron
predictions = tensorflow.nn.sigmoid(af_input)
```

Our activation function is used to merge all inputs, weights, and bias into a single value describing the expected class score of each input.
Normally, there is a weight for each input. Each input is multiplied by its corresponding weight. We don't have to make element-by-element multiplications and matrix multiplication can be a good solution as in line 13.

```
# Preparing inputs of the activation function
af_input = tensorflow.matmul(training_inputs, weights) +
bias
```

Just prepare a matrix for inputs and another one for weights and multiply these two matrices. The `tensorflow.matmul()` operation make that for you.
Then `bias` is added to the summation of individual input-weight multiplications as in line 13. The result, `af_input` in our example, is then applied to the activation function as in line 15. We are not going to create the function manually as it is already supported by `tensorflow.nn` API. There are different types of activation functions and `sigmoid` is sufficient for our case. In our case, the result returned by the sigmoid activation function represents the expected class score of the current input.

Prediction Error

Evaluating the model is an essential step after it has been trained. This is why there is a loss function in line 18 in **CodeSample1**.

```
# Measuring the prediction error of the network after being trained
prediction_error = tensorflow.reduce_sum(training_outputs - predictions)
```

It is very simple but at least does well for our case. Just find the difference between each desired output and its corresponding predicted output by the model. The goal is to measure how far the predicted outputs of the trained neural network from their corresponding desired outputs. To find a single value representing the overall error of the network, the summation of individual differences is calculated using the `tensorflow.reduce_sum()` operation.

Updating Network Parameters

The prediction error of the model may not be zero from the first trial and it may be very high. This is why there must be a way for automatically updating and optimizing the model parameters to get the least possible error. One of the common optimizers is gradient descent (GD). GD tries to find the relationship between each parameter and the prediction error to know how each parameter affects the error. This is by first trying the initial parameters. If they didn't do well, then GD will try to change the parameters values and moving in the direction that minimizes the error. The GD optimizer is applied in our example in line 20 to minimize the previously calculated prediction error. The `learning_rate` of the `tensorflow.train.GradientDescentOptimizer` is just a hyper-parameter.

```
# Minimizing the prediction error using gradient descent optimizer
train_op = tensorflow.train.GradientDescentOptimizer(learning_rate=0.05).minimize(prediction_error)
```

Training Loop

Previously, we prepared the steps to follow from accepting the inputs to generating the prediction error of the model. Moreover, we described how the model parameters are to be updated. The remaining step is to go into a loop that updates the parameters automatically.

There are some works to be done before going into the loop including:

- Creating the Session as in line 24.
- Initializing all Variables as in line 27.

```
# Creating a TensorFlow Session
sess = tensorflow.Session()

# Initializing the TensorFlow Variables (weights and bias)
sess.run(tensorflow.global_variables_initializer())
```

After that, we can run the training loop as in lines 42 and 43. Note that the only parameter specified to be fetched is the Tensor returned by `tensorflow.train.GradientDescentOptimizer` which is `train_op`. This is because fetching train_op will cause all parameters to be updated.

The loop will last for 10,000 iterations and at each iteration, the GD optimizer will generate new values for the parameters that decrease the error.

```
# Training loop of the neural network
for step in range(10000):
    sess.run(fetches=[train_op], feed_dict={

training_inputs: training_inputs_data,
                                    training_outputs: tra
ining_outputs_data})
```

Because data are stored into placeholders, then these placeholders must be initialized using the `feed_dict` argument of the `tensorflow.Session.run()` operation. The weights and bias placeholders are initialized using a previously created NumPy arrays as in lines 30 and 36.

We can also avoid creating a separate NumPy arrays and make assign the data to the placeholders within the `run()` operation as follows:

```
# Training loop of the neural network
for step in range(10000):
    sess.run(fetches=[train_op], feed_dict={

            training_inputs: [[255, 0, 0],
                              [248, 80, 68],
                             [0, 0, 255],
                             [67, 15, 210]]  ,
            training_outputs: [[1],
                             [1],
                             [0],
                             [0]]})
```

But for code clarity, the NumPy arrays are created separately from the `run()` operation.

Testing the Trained Neural Network

After getting out of the training loop, the neural network will be trained and ready for predicting unknown samples. In line 48, two new samples were used for testing the network accuracy.

```
# Predicted classes of some testing data
print("Expected Class
Scores : ", sess.run(fetches=predictions,
feed_dict={training_inputs: [[255, 100, 50],
                                                    [30
, 50, 255]]}))
```

The expected class scores are as follows:

```
Expected class scores :  [[ 1.]
                          [ 1.]]
```

This is not accurate result because it says that the expected class score of the two samples is indexed 1 which is the BLUE class. The first sample is of class indexed 0 which is RED.
What is the reason for such weak prediction for such very simple example? The reason is the bad use of the initial values for the network parameters (weights and bias). We can try using different initial values and see how the results get changed. For example, using the `tensorflow.truncated_normal()` operation, we can generate the initial values for both weights and bias as follows:

```
# Preparing neural network parameters (weights and bias) using TensorFlow Variables
weights = tensorflow.Variable(tensorflow.truncated_normal(shape=[3, 1], dtype=tensorflow.float32))
bias = tensorflow.Variable(tensorflow.truncated_normal(shape=[1, 1], dtype=tensorflow.float32))
```

Then we can train the network again using the newly used initial values and predict the results again. The result of expectation is as follows:

```
Expected class score :  [[  3.23404099e-23]
                         [  1.00000000e+00]]
```

The new values for the weights and bias can be printed as follows:

```
# Printing weights initially generated using tf.truncated_normal()
print("Weights : ", sess.run(weights))

# Printing bias initially generated using tf.truncated_normal()
print("Bias : ", sess.run(bias))
```

Here is the result:

```
Weights :  [[-0.20039153]
            [-0.91293597]
            [ 0.44236434]]

```

- `Bias : [[-0.685884]]`

The results enhanced so much and the error is now `0.0` compared to `1.0` in the previous fixed initial values. This proves the importance of initializing the weights of the neural networks well.

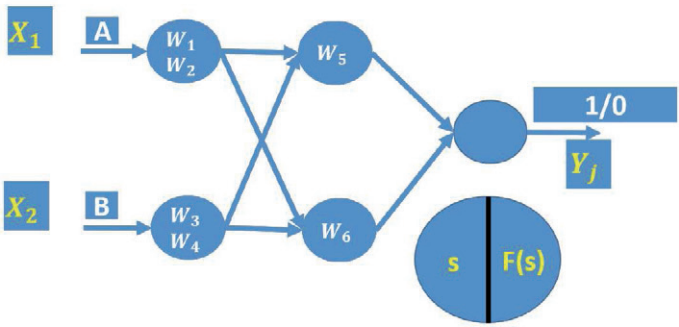
This is the end for our simple example. Next, we will explore another example that creates the XOR logic gate using a neural network with one hidden layer containing two neurons.

XOR Logic Gate using Feed-Forward Neural Network (FFNN)

The same concepts applied previously will also hold in all neural networks. There are some changes such as adding more layers or more neurons, changing the type of activation function, or using different loss function.
The data used as input are as follows:

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

The network architecture to be created using TensorFlow as a FFNN with one hidden layer containing two neurons is as follows:



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 the inputs to the output layer which produces the final expected class scores of the input data.

```
1. import tensorflow
2. import numpy
3.
4. # Preparing training data (inputs-outputs)
5. training_inputs = tensorflow.placeholder(shape=[None, 2],
   dtype=tensorflow.float32)
6. training_outputs = tensorflow.placeholder(shape=[None, 1]
   , dtype=tensorflow.float32) #Desired outputs for each inp
   ut
7.
8. """
9. Hidden layer with two neurons
10.     """
11.
12.     # Preparing neural network parameters (weights and bi
   as) using TensorFlow Variables
13.     weights_hidden = tensorflow.Variable(tensorflow.trunc
   ated_normal(shape=[2, 2], dtype=tensorflow.float32))
14.     bias_hidden = tensorflow.Variable(tensorflow.truncate
   d_normal(shape=[1, 2], dtype=tensorflow.float32))
15.
```

```
16.     # Preparing inputs of the activation function
17.     af_input_hidden = tensorflow.matmul(training_inputs,
  weights_hidden) + bias_hidden
18.
19.     # Activation function of the output layer neuron
20.     hidden_layer_output = tensorflow.nn.sigmoid(af_input_
  hidden)
21.
22.
23.     """
24.     Output layer with one neuron
25.     """
26.
27.     # Preparing neural network parameters (weights and bi
  as) using TensorFlow Variables
28.     weights_output = tensorflow.Variable(tensorflow.trunc
  ated_normal(shape=[2, 1], dtype=tensorflow.float32))
29.     bias_output = tensorflow.Variable(tensorflow.truncate
  d_normal(shape=[1, 1], dtype=tensorflow.float32))
30.
31.     # Preparing inputs of the activation function
32.     af_input_output = tensorflow.matmul(hidden_layer_outp
  ut, weights_output) + bias_output
33.
34.     # Activation function of the output layer neuron
35.     predictions = tensorflow.nn.sigmoid(af_input_output)

36.
37.
38.     #-----------------------------------
39.
40.     # Measuring the prediction error of the network after
  being trained
41.     prediction_error = 0.5 * tensorflow.reduce_sum(tensor
  flow.subtract(predictions, training_outputs) * tensorflow
  .subtract(predictions, training_outputs))
42.
```

```
43.     # Minimizing the prediction error using gradient desc
  ent optimizer
44.     train_op = tensorflow.train.GradientDescentOptimizer(
  0.05).minimize(prediction_error)
45.
46.     # Creating a TensorFlow Session
47.     sess = tensorflow.Session()
48.
49.     # Initializing the TensorFlow Variables (weights and
  bias)
50.     sess.run(tensorflow.global_variables_initializer())
51.
52.     # Training data inputs
53.     training_inputs_data = [[1.0, 0.0],
54.                             [1.0, 1.0],
55.                             [0.0, 1.0],
56.                             [0.0, 0.0]]
57.
58.     # Training data desired outputs
59.     training_outputs_data = [[1.0],
60.                              [1.0],
61.                              [0.0],
62.                              [0.0]]
63.
64.     # Training loop of the neural network
65.     for step in range(10000):
66.         op, err, p = sess.run(fetches=[train_op, predicti
  on_error, predictions],
67.                               feed_dict={training_inputs:
  training_inputs_data,
68.                                          training_outputs
  : training_outputs_data})
69.         print(str(step), ": ", err)
70.
71.     # Class scores of some testing data
72.     print("Expected class
  scroes : ", sess.run(predictions, feed_dict={training_inp
  uts: training_inputs_data}))
```

```
73.
74.     # Printing hidden layer weights
  initially generated using tf.truncated_normal()
75.     print("Hidden layer initial weights : ", sess.run(wei
  ghts_hidden))
76.
77.     # Printing hidden layer bias
  initially generated using tf.truncated_normal()
78.     print("Hidden layer initial weights : ", sess.run(bia
  s_hidden))
79.
80.     # Printing output layer weights initially
  generated using tf.truncated_normal()
81.     print("Output layer initial weights : ", sess.run(wei
  ghts_output))
82.
83.     # Printing output layer bias
  initially generated using tf.truncated_normal()
84.     print("Output layer initial weights : ", sess.run(bia
  s_output))
85.
86.     # Closing the TensorFlow Session to free resources
87.     sess.close()
```

Here is the expected class scores of the test data and the weights and biases for both hidden and output layer.

```
• Expected class scores :  [[ 0.75373638]
•                          [ 0.94796741]
•                          [ 0.25110725]
•                          [ 0.03870015]]
•
• Hidden layer weights :  [[ 1.68877864 -3.25296354]
•                  [ 1.36028981 -1.6849252 ]]
•
• Hidden layer weights :  [[-1.27290058  2.33101916]]
•
```

```
Output layer weights :  [[ 2.42446136]
                    [-5.42509556]]

Output layer weights :  [[ 1.20168602]]
```

Chapter 5

Visualizing Graphs in TensorBoard

Because the computations done using TensorFlow can be complex like training a deep neural network with large amounts of data, TensorFlow created a suite of visualization tools called TensorBoard (TB for short) to make understanding and debugging TensorFlow programs easier.

Steps to follow for visualizing graphs in TensorBoard:

1. Generate the graph
2. Save the graph in a directory
3. Launch TensorBoard within the previous directory
4. Access TensorBoard from a Web browser
5. Load the graph file

1) Generating the Graph

Just create a simple program in TensorFlow that has a number of nodes and a number of edges like the following code and your graph will be saved in the current default graph:

```
1. import tensorflow as tf
2.
3. a = tf.add(1, 2,)
4. b = tf.multiply(a, 3)
5. c = tf.add(4, 5,)
6. d = tf.multiply(c, 6,)
7. e = tf.multiply(4, 5,)
8. f = tf.div(c, 6,)
9. g = tf.add(b, d)
10.     h = tf.multiply(g, f)
11.
12.     with tf.Session() as sess:
13.         print(sess.run(h))
```

2) Save the Graph in a Directory

Using the tensorflow.summary.FileWriter, we can write the graph into a specific directory. This directory is important because you will use it later for loading the graph in TensorBoard.

```
1. with tf.Session() as sess:
2.     writer = tf.summary.FileWriter("/tmp/log/", sess.grap
   h)
3.     print(sess.run(h))
4.     writer.close()
```

I use PyCharm as an IDE for writing TensorFlow programs and PyCharm saves the log files under a directory called log in the project. For example, the previous code generated a code in this path:

C:\Users\Dell\PycharmProjects\test\log

3) Launch TensorBoard within the previous directory

To launch TensorBoard, you need to follow these steps:

1. Activate the TensorFlow environment.
2. Launch TensorBoard

3.1) Activating TensorFlow

From CMD:

C:\activate <tf-env-name>

If you named the tensorflow environment tensorflow, then issue this command:

C:\activate tensorflow

3.2) Launching TensorBoard

To launch TensorBoard, use this command in CMD:

C:\tensorboard --logdir=<path/to/log-directory>

This will launch TensorBoard in the entered logging directory where graph files exist. For example, to launch TensorBoard in /tmp/log/ directory, use this command:

C:\tensorboard --logdir=/tmp/log/

To make sure that the file was exported successfully from the Python code, use this CMD command before launching the TB:

C:\tensorboard --inspect --logdir /tmp/log/

4) Accessing TensorBoard from Web Browser

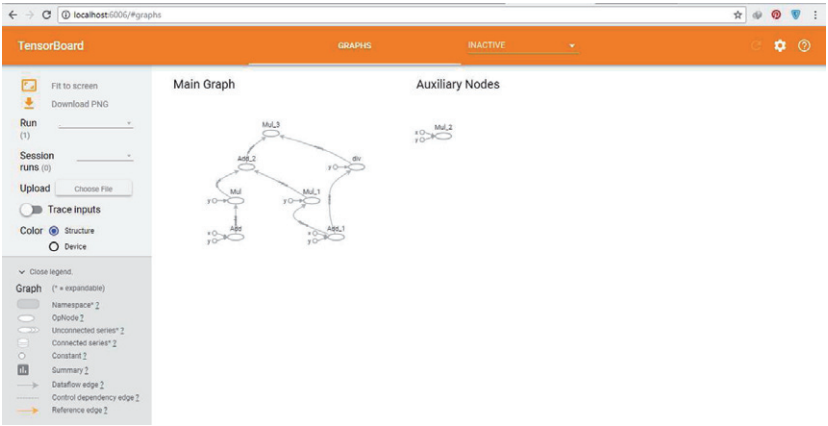
Finally, view TB in the Web browser by going to:

Localhost:6006

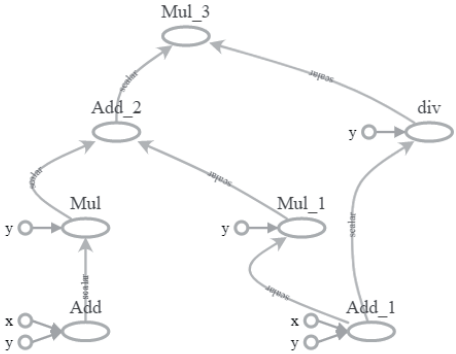
This will open TB.

5) Loading the Graph File

Navigate to the Graphs page. The graphs in the entered directory in the CMD command and all sub-directories will be shown in the page.
The page is shown below.



Here is the graph of the program created above:


Mul_3
Add_2
div
Mul
Mul_1
Add
Add_1
x
y


Mul_2
x
y

For More Information

https://developers.googleblog.com/2017/09/introducing-tensorflow-datasets.html
https://www.tensorflow.org/versions/r1.0/programmers_guide/variables
https://www.tensorflow.org/extend/estimators
https://www.tensorflow.org/get_started/estimator
https://www.tensorflow.org/programmers_guide/estimators
https://www.tensorflow.org/api_docs/python/tf/estimator/inputs
https://www.tensorflow.org/api_docs/python/tf/estimator/inputs/numpy_input_fn
https://www.tensorflow.org/api_docs/python/tf/feature_column
https://www.tensorflow.org/api_docs/python/tf/estimator/LinearClassifier
https://pandas.pydata.org/pandas-docs/stable/dsintro.html
http://pandas.pydata.org/pandas-docs/stable/indexing.html
https://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.to_csv.html
https://www.tensorflow.org/get_started/mnist/mechanics