**FIRST CONTACT WITH TENSORFLOW**

**Get started with with Deep Learning programming**

My first code in TensorFlow

As I mentioned at the beginning, we will move in this exploration of the planet TensorFlow with little theory and lots of practice. Let’s start!

From now on, it is best to use any text editor to write python code and save it with extension “.py” (eg *test.py*). To run the code, it will be enough with the command *python test.py*.

To get a first impression of what a TensorFlow’s program is, I suggest doing a simple multiplication program; the code looks like this:

import tensorflow as tf

a = tf.placeholder("float")

b = tf.placeholder("float")

y = tf.mul(a, b)

sess = tf.Session()

print sess.run(y, feed\_dict={a: 3, b: 3})

In this code, after importing the Python module tensorflow, we define “symbolic” variables, called *placeholder* in order to manipulate them during the program execution. Then, we move these variables as a parameter in the call to the function multiply that TensorFlow offers. *tf.mul* is one of the many mathematical operations that TensorFlow offers to manipulate the *tensors*. In this moment, tensors can be considered dynamically-sized, multidimensional data arrays.

The main ones are shown in the following table:

|  |  |
| --- | --- |
| **Operation** | **Description** |
| **tf.add** | sum |
| **tf.sub** | substraction |
| **tf.mul** | multiplication |
| **tf.div** | division |
| **tf.mod** | module |
| **tf.abs** | return the absolute value |
| **tf.neg** | return negative value |
| **tf.sign** | return the sign |
| **tf.inv** | returns the inverse |
| **tf.square** | calculates the square |
| **tf.round** | returns the nearest integer |
| **tf.sqrt** | calculates the square root |
| **tf.pow** | calculates the power |
| **tf.exp** | calculates the exponential |
| **tf.log** | calculates the logarithm |
| **tf.maximum** | returns the maximum |
| **tf.minimum** | returns the minimum |
| **tf.cos** | calculates the cosine |
| **tf.sin** | calculates the sine |

TensorFlow also offers the programmer a number of functions to perform mathematical operations on matrices. Some are listed below*:*

|  |  |
| --- | --- |
| **Operation** | **Description** |
| **tf.diag** | returns a diagonal tensor with a given diagonal values |
| **tf.transpose** | returns the transposes of the argument |
| **tf.matmul** | returns a tensor product of multiplying two tensors listed as arguments |
| **tf.matrix\_determinant** | returns the determinant of the square matrix specified as an argument |
| **tf.matrix\_inverse** | returns the inverse of the square matrix specified as an argument |

The next step, one of the most important, is to create a session to evaluate the specified symbolic expression. Indeed, until now nothing has yet been executed in this TensorFlowcode. Let me emphasize that TensorFlow is both, an interface to express Machine Learning’s algorithms and an implementation to run them, and this is a good example.

Programs interact with Tensorflow libraries by creating a session with *Session()*; it is only from the creation of this session when we can call the *run()* method, and that is when it really starts to run the specified code. In this particular example, the values of the variables are introduced into the *run()* method with a *feed\_dict* argument. That’s when the associated code solves the expression and exits from the display a 9 as a result of multiplication.

With this simple example, I tried to introduce the idea that the normal way to program in TensorFlow is to specify the whole problem first, and eventually create a session to allow the running of the associated computation.

Sometimes however, we are interested in having more flexibility in order to structure the code, inserting operations to build the graph with operations running part of it. It happens when we are, for example, using interactive environments of Python such as IPython [[8]](http://www.jorditorres.org/first-contact-with-tensorflow/" \l "_ftn8). For this purpose, TesorFlow offers the *tf.InteractiveSession()* class.

The motivation for this programming model is beyond the reach of this book. However, to continue with the next chapter, we only need to know that all information is saved internally in a graph structure that contains all the information operations and data .

This graph describes mathematical computations. The nodes typically implement mathematical operations, but they can also represent points of data entry, output results, or read/write persistent variables. The edges describe the relationships between nodes with their inputs and outputs and at the same time carry tensors, the basic data structure of TensorFlow.

The representation of the information as a graph allows TensorFlow to know the dependencies between transactions and assigns operations to devices asynchronously, and in parallel, when these operations already have their associated tensors (indicated in the edges input) available.

Parallelism is therefore one of the factors that enables us to speed up the execution of some computationally expensive algorithms, but also because TensorFlow has already efficiently implemented a set of complex operations. In addition, most of these operations have associated kernels which are implementations of operations designed for specific devices such as GPUs. The following table summarizes the most important operations/kernels[[9]](http://www.jorditorres.org/first-contact-with-tensorflow/" \l "_ftn9):

|  |  |
| --- | --- |
| **Operations groups** | **Operations** |
| **Maths** | Add, Sub, Mul, Div, Exp, Log, Greater, Less, Equal |
| **Array** | Concat, Slice, Split, Constant, Rank, Shape, Shuffle |
| **Matrix** | MatMul, MatrixInverse, MatrixDeterminant |
| **Neuronal Network** | SoftMax, Sigmoid, ReLU, Convolution2D, MaxPool |
| **Checkpointing** | Save, Restore |
| **Queues and syncronizations** | Enqueue, Dequeue, MutexAcquire, MutexRelease |
| **Flow control** | Merge, Switch, Enter, Leave, NextIteration |

Display panel Tensorboard

To make it more comprehensive, TensorFlow includes functions to debug and optimize programs in a visualization tool called TensorBoard. TensorBoard can view different types of statistics about the parameters and details of any part of the graph computing graphically.

The data displayed with TensorBoard module is generated during the execution of TensorFlow and stored in trace files whose data is obtained from the *summary operations*. In the documentation page[[10]](http://www.jorditorres.org/first-contact-with-tensorflow/" \l "_ftn10) of TensorFlow, you can find detailed explanation of the Python API.

The way we can invoke it is very simple: a service with Tensorflow commands from the command line, which will include as an argument the file that contains the trace.

(tensorflow)$ tensorboard --logdir=&lt;trace file&gt;

You simply need to access the local socket 6006 from the browser[[11]](http://www.jorditorres.org/first-contact-with-tensorflow/" \l "_ftn11) with http://localhost:6006/ .

The visualization tool called TensorBoard is beyond the reach of this book. For more details about how Tensorboard works, the reader can visit the section *TensorBoard Graph Visualization*[[12]](http://www.jorditorres.org/first-contact-with-tensorflow/" \l "_ftn12)from the TensorFlow tutorial page.  
[[contents link]](http://www.jorditorres.org/first-contact-with-tensorflow/#indice)

2. LINEAR REGRESSION IN TENSORFLOW

In this chapter, I will begin exploring TensorFlow’s coding with a simple model: Linear Regression. Based on this example, I will present some code basics and, at the same time, how to call various important components in the learning process, such as the cost function or the algorithm gradient descent.

Model of relationship between variables

Linear regression is a statistical technique used to measure the relationship between variables. Its interest is that the algorithm that implements it is not conceptually complex, and can also be adapted to a wide variety of situations. For these reasons, I have found it interesting to start delving into TensorFlow with an example of linear regression.

Remember that both, in the case of two variables (simple regression) and the case of more than two variables (multiple regression), linear regression models the relationship between a dependent variable, independent variables *xi* and a random term *b*.

In this section I will create a simple example to explain how TensorFlow works assuming that our data model corresponds to a simple linear regression as *y = W \* x + b*. For this, I use a simple Python program that creates data in a two-dimensional space, and then I will ask TensorFlow to look for the line that fits the best in these points.

The first thing to do is to import the NumPy package that we will use to generate points. The code we have created is as it follows:

import numpy as np

num\_points = 1000

vectors\_set = []

for i in xrange(num\_points):

x1= np.random.normal(0.0, 0.55)

y1= x1 \* 0.1 + 0.3 + np.random.normal(0.0, 0.03)

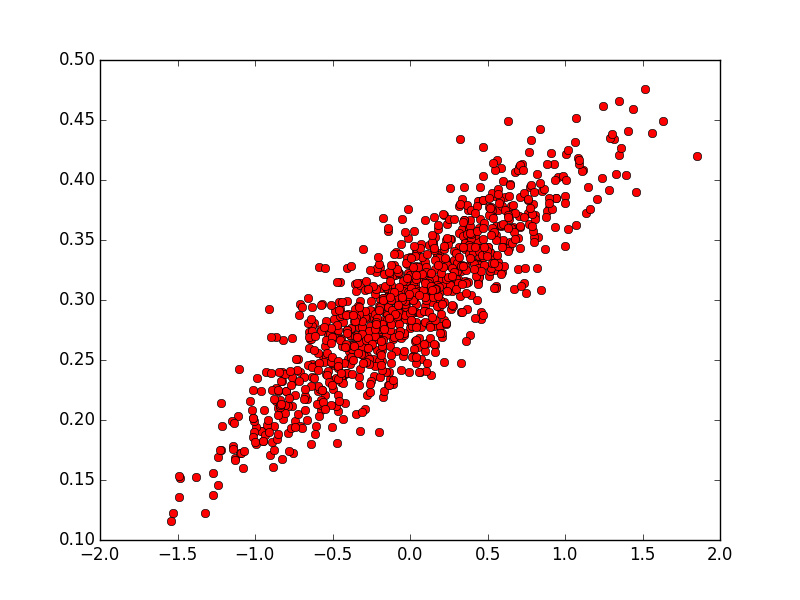
vectors\_set.append([x1, y1])

x\_data = [v[0] for v in vectors\_set]

y\_data = [v[1] for v in vectors\_set]

As you can see from the code, we have generated points following the relationship *y = 0.1 \* x + 0.3*, albeit with some variation, using a normal distribution, so the points do not fully correspond to a line, allowing us to make a more interesting example.

In our case, a display of the resulting cloud of points is:



The reader can view them with the following code (in this case, we need to import some of the functions of *matplotlib*package, running *pip install matplotlib*[[13]](http://www.jorditorres.org/first-contact-with-tensorflow/" \l "_ftn13)):

import matplotlib.pyplot as plt

plt.plot(x\_data, y\_data, 'ro', label='Original data')

plt.legend()

plt.show()

These points are the data that we will consider the training dataset for our model.

Cost function and gradient descent algorithm

The next step is to train our learning algorithm to be able to obtain output values *y*, estimated from the input data *x\_data*. In this case, as we know in advance that it is a linear regression, we can represent our model with only two parameters: *W* and *b*.

The objective is to generate a TensorFlow code that allows to find the best parameters *W* and *b,*that from input data *x\_data*, adjunct them to *y\_data* output data, in our case it will be a straight line defined by *y\_data = W \* x\_data + b*. The reader knows that *W* should be close to 0.1 and *b* to 0.3, but TensorFlow does not know and it must realize it for itself.

A standard way to solve such problems is to iterate through each value of the data set and modify the parameters *W* and *b* in order to get a more precise answer every time. To find out if we are improving in these iterations, we will define a cost function (also called “error function”) that measures how “good” (actually, as “bad”) a certain line is.

This function receives the pair of *W*and as parameters *b* and returns an error value based on how well the line fits the data. In our example we can use as a cost function the*mean squared error****[[14]](http://www.jorditorres.org/first-contact-with-tensorflow/" \l "_ftn14)***. With the *mean squared error* we get the average of the “errors” based on the distance between the real values and the estimated one on each iteration of the algorithm.

Later, I will go into more detail with the cost function and its alternatives, but for this introductory example the *mean squared error* helps us to move forward step by step.

Now it is time to program everything that I have explained with TensorFlow. To do this, first we will create three variables with the following sentences:

W = tf.Variable(tf.random\_uniform([1], -1.0, 1.0))

b = tf.Variable(tf.zeros([1]))

y = W \* x\_data + b

For now, we can move forward knowing only that the call to the method *Variable* is defining a variable that resides in the internal graph data structure of TensorFlow, of which I have spoken above. We will return with more information about the method parameters later, but for now I think that it’s better to move forward to facilitate this first approach.

Now, with these variables defined, we can express the cost function that we discussed earlier, based on the distance between each point and the calculated point with the function y= *W \* x + b*. After that, we can calculate its square, and average the sum. In TensorFlow this cost function is expressed as follows:

loss = tf.reduce\_mean(tf.square(y - y\_data))

As we see, this expression calculates the average of the squared distances between the *y\_data* point that we know, and the point *y* calculated from the input *x\_data*.

At this point, the reader might already suspects that the line that best fits our data is the one that obtains the lesser error value. Therefore, if we minimize the error function, we will find the best model for our data.

Without going into too much detail at the moment, this is what the optimization algorithm that minimizes functions known as*gradient descent****[[15]](http://www.jorditorres.org/first-contact-with-tensorflow/" \l "_ftn15)*** achieves. At a theoretical level gradient descent is an algorithm that given a function defined by a set of parameters, it starts with an initial set of parameter values and iteratively moves toward a set of values that minimize the function. This iterative minimization is achieved taking steps in the negative direction of the function *gradient*[[16]](http://www.jorditorres.org/first-contact-with-tensorflow/" \l "_ftn16). It’s conventional to square the distance to ensure that it is positive and to make the error function differentiable in order to compute the gradient.

The algorithm begins with the initial values of a set of parameters (in our case *W* and *b*), and then the algorithm is iteratively adjusting the value of those variables in a way that, in the end of the process, the values of the variables minimize the cost function.

To use this algorithm in TensorFlow, we just have to execute the following two statements:

optimizer = tf.train.GradientDescentOptimizer(0.5)

train = optimizer.minimize(loss)

Right now, this is enough to have the idea that TensorFlow has created the relevant data in its internal data structure, and it has also implemented in this structure an optimizer that may be invoked by *train*, which it is a *gradient descent* algorithm to the cost function defined. Later on, we will discuss the function parameter called *learning rate* (in our example with value 0.5).

Running the algorithm

As we have seen before, at this point in the code the calls specified to the library TensorFlow have only added information to its internal graph, and the runtime of TensorFlow has not yet run any of the algorithms. Therefore, like the example of the previous chapter, we must create a *session*, call the *run* method and passing *train* as parameter. Also, because in the code we have specified variables, we must initialize them previously with the following calls:

init = tf.initialize\_all\_variables()

sess = tf.Session()

sess.run(init)

Now we can start the iterative process that will allow us to find the values of *W* and *b,*defining the model line that best fits the points of entry. The training process continues until the model achieves a desired level of accuracy on the training data. In our particular example, if we assume that with only 8 iterations is sufficient, the code could be:

for step in xrange(8):

sess.run(train)

print step, sess.run(W), sess.run(b)

The result of running this code show that the values of *W* and *b* are close to the value that we know beforehand. In my case, the result of the *print* is:

(array([ 0.09150752], dtype=float32), array([ 0.30007562], dtype=float32))

And, if we graphically display the result with the following code:

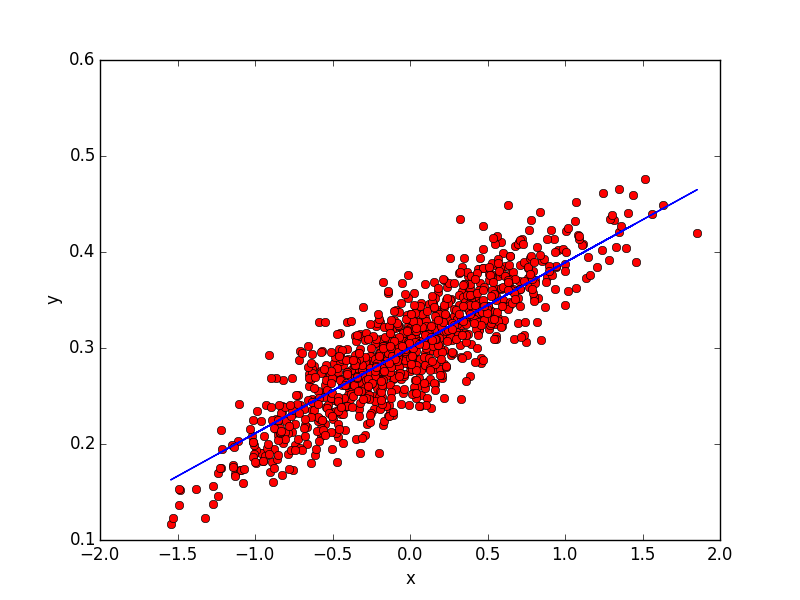
plt.plot(x\_data, y\_data, 'ro')

plt.plot(x\_data, sess.run(W) \* x\_data + sess.run(b))

plt.legend()

plt.show()

We can see graphically the line defined by parameters *W* = 0.0854 and *b* = 0.299 achieved with only 8 iterations:



Note that we have only executed eight iterations to simplify the explanation, but if we run more, the value of parameters get closer to the expected values. We can use the following sentence to print the values of *W* and *b*:

print(step, sess.run(W), sess.run(b))

In our case the print outputs are:

(0, array([-0.04841119], dtype=float32), array([ 0.29720169], dtype=float32))

(1, array([-0.00449257], dtype=float32), array([ 0.29804006], dtype=float32))

(2, array([ 0.02618564], dtype=float32), array([ 0.29869056], dtype=float32))

(3, array([ 0.04761609], dtype=float32), array([ 0.29914495], dtype=float32))

(4, array([ 0.06258646], dtype=float32), array([ 0.29946238], dtype=float32))

(5, array([ 0.07304412], dtype=float32), array([ 0.29968411], dtype=float32))

(6, array([ 0.08034936], dtype=float32), array([ 0.29983902], dtype=float32))

(7, array([ 0.08545248], dtype=float32), array([ 0.29994723], dtype=float32))

You can observe that the algorithm begins with the initial values of *W= -0.0484* and *b=0.2972* (in our case) and then the algorithm is iteratively adjusting in a way that the values of the variables minimize the cost function.

You can also check that the cost function is decreasing with

print(step, sess.run(loss))

In this case the print output is:

(0, 0.015878126)

(1, 0.0079048825)

(2, 0.0041520335)

(3, 0.0023856456)

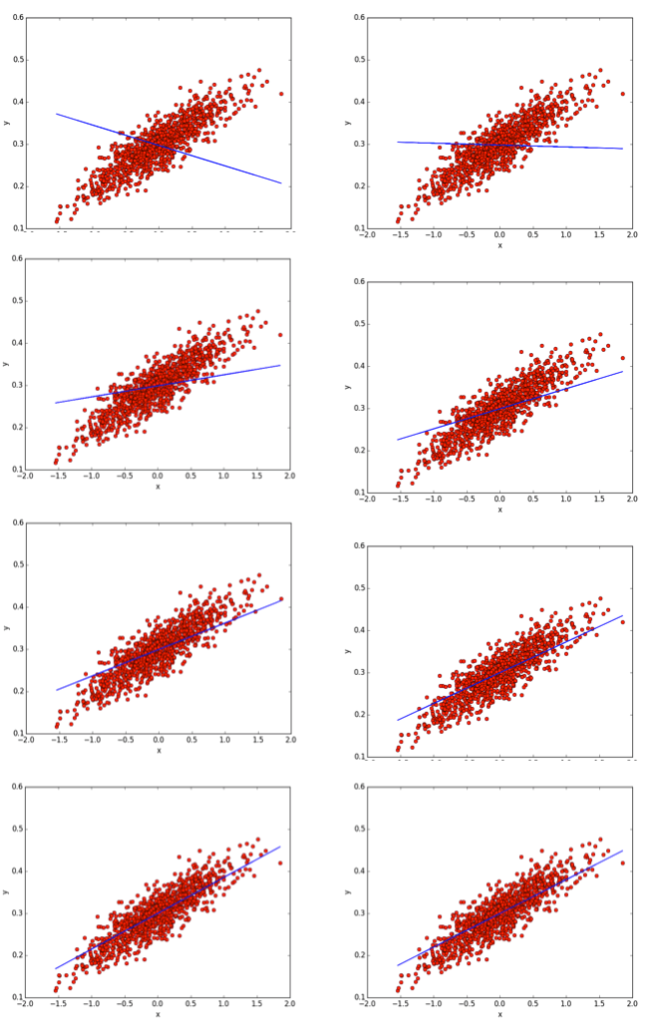
(4, 0.0015542418)

(5, 0.001162916)

(6, 0.00097872759)

(7, 0.00089203351)

I suggest that reader visualizes the plot at each iteration, allowing us to visually observe how the algorithm is adjusting the parameter values. In our case the 8 snapshots are:

  
As the reader can see, at each iteration of the algorithm the line fits better to the data. How does the gradient descent algorithm get closer to the values of the parameters that minimize the cost function?

Since our error function consists of two parameters (W and b) we can visualize it as a two-dimensional surface. Each point in this two-dimensional space represents a line. The height of the function at each point is the error value for that line. In this surface some lines yield smaller error values than others. When TensorFlow runs gradient descent search, it will start from some location on this surface (in our example the point *W= -0.04841119* and *b=0.29720169*) and move downhill to find the line with the lowest error.

To run gradient descent on this error function, TensorFlow computes its gradient. The gradient will act like a compass and always point us downhill. To compute it, TensorFlow will differentiate the error function, that in our case means that it will need to compute a partial derivative for *W* and *b* that indicates the direction to move in for each iteration.

The *learning rate* parameter mentioned before, controls how large of a step TensorFlow will take downhill during each iteration. If we introduce a parameter too large of a step, we may step over the minimum. However, if we indicate to TensorFlow to take small steps, it will require much iteration to arrive at the minimum. So using a good learning rate is crucial. There are different techniques to adapt the value of the learning rate parameter, however it is beyond the scope of this introductory book. A good way to ensure that gradient descent algorithm is working fine is to make sure that the error decreases at each iteration.

Remember that in order to facilitate the reader to test the code described in this chapter, you can download it from *Github*[[17]](http://www.jorditorres.org/first-contact-with-tensorflow/" \l "_ftn17)of the book with the name of *regression.py*. Here you will find all together for easy tracking:

import numpy as np

num\_points = 1000

vectors\_set = []

for i in xrange(num\_points):

x1= np.random.normal(0.0, 0.55)

y1= x1 \* 0.1 + 0.3 + np.random.normal(0.0, 0.03)

vectors\_set.append([x1, y1])

x\_data = [v[0] for v in vectors\_set]

y\_data = [v[1] for v in vectors\_set]

import matplotlib.pyplot as plt

#Graphic display

plt.plot(x\_data, y\_data, 'ro')

plt.legend()

plt.show()

import tensorflow as tf

W = tf.Variable(tf.random\_uniform([1], -1.0, 1.0))

b = tf.Variable(tf.zeros([1]))

y = W \* x\_data + b

loss = tf.reduce\_mean(tf.square(y - y\_data))

optimizer = tf.train.GradientDescentOptimizer(0.5)

train = optimizer.minimize(loss)

init = tf.initialize\_all\_variables()

sess = tf.Session()

sess.run(init)

for step in xrange(8):

sess.run(train)

print(step, sess.run(W), sess.run(b))

print(step, sess.run(loss))

#Graphic display

plt.plot(x\_data, y\_data, 'ro')

plt.plot(x\_data, sess.run(W) \* x\_data + sess.run(b))

plt.xlabel('x')

plt.xlim(-2,2)

plt.ylim(0.1,0.6)

plt.ylabel('y')

plt.legend()

plt.show()

In this chapter we have begun to explore the possibilities of the TensorFlow package with a first intuitive approach to two fundamental pieces: the *cost function* and *gradient descent algorithm*, using a basic linear regression algorithm for their introduction. In the next chapter we will go into more detail about the data structures used by TensorFlow package.

[[contents index]](http://www.jorditorres.org/first-contact-with-tensorflow/#indice)

3. CLUSTERING IN TENSORFLOW

Linear regression, which has been presented in the previous chapter, is a supervised learning algorithm in which we use the data and output values (or labels) to build a model that fits them. But we haven’t always tagged data, and despite this we also want analyze them in some way. In this case, we can use an unsupervised learning algorithm as clustering. The clustering method is widely used because it is often a good approach for preliminary screening data analysis.

In this chapter, I will present the clustering algorithm called *K-means*. It is surely the most popular and widely used to automatically group the data into coherent subsets so that all the elements in a subset are more similar to each other than with the rest. In this algorithm, we do not have any target or outcome variable to predict estimations.

I will also use this chapter to achieve progress in the knowledge of TensorFlow and go into more detail in the basic data structure called *tensor*. I will start by explaining what this type of data is like and present the transformations that can be performed on it. Then, I will show the use of *K-means* algorithm in a case study using *tensors*.

Basic data structure: *tensor*

TensorFlow programs use a basic data structure called *tensor* to represent all of their datum. A tensor can be considered a dynamically-sized multidimensional data arrays that have as a properties a static data type, which can be from *boolean* or *string* to a variety of numeric types. Below is a table of the main types and their equivalent in Python.

|  |  |  |
| --- | --- | --- |
| **Type in**  **TensorFlow** | **Type in**  **Python** | **Description** |
| DT\_FLOAT | tf.float32 | Floating point of 32 bits |
| DT\_INT16 | tf.int16 | Integer of 16 bits |
| DT\_INT32 | tf.int32 | Integer of 32 bits |
| DT\_INT64 | tf.int64 | Integer of 64 bits |
| DT\_STRING | tf.string | String |
| DT\_BOOL | tf.bool | Boolean |

In addition, each *tensor* has a *rank*, which is the number of its dimensions. For example, the following tensor (defined as a list in Python) has rank 2:

t = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

Tensors can have any rank. A rank 2 tensor is usually considered a matrix, and a rank 1 tensor would be a vector. Rank 0 is considered a scalar value.

TensorFlow documentation uses three types of naming conventions to describe the dimension of a tensor: *Shape*, *Rank* and *Dimension Number*. The following table shows the relationship between them in order to make easier the Tensor Flow documentation’s traking easier:

|  |  |  |
| --- | --- | --- |
| ***Shape*** | ***Rank*** | ***Dimension Number*** |
| [] | 0 | 0-D |
| [D0] | 1 | 1-D |
| [D0, D1] | 2 | 2-D |
| [D0, D1, D2] | 3 | 3-D |
| … | … | … |
| [D0, D1, … Dn] | n | n-D |

These tensors can be manipulated with a series of transformations that supply the TensorFlow package. Below, we discuss some of them in the next table.

Throughout this chapter we will go into more detail on some of them. A comprehensive list of transformations and details of each one can be found on the official website of TensorFlow, *Tensor Transformations****[[18]](http://www.jorditorres.org/first-contact-with-tensorflow/" \l "_ftn18)***.

|  |  |
| --- | --- |
| **Operation** | **Description** |
| tf.shape | To find a shape of a *tensor* |
| tf.size | To find the size of a *tensor* |
| tf.rank | To find a rank of a *tensor* |
| tf.reshape | To change the shape of a *tensor* keeping the same elements contained |
| tf.squeeze | To delete in a *tensor* dimensions of size 1 |
| tf.expand\_dims | To insert a dimension to a *tensor* |
| tf.slice | To remove a portions of a *tensor* |
| tf.split | To divide a *tensor* into several tensors along one dimension |
| tf.tile | To create a new *tensor* replicating a *tensor* multiple times |
| tf.concat | To concatenate *tensors* in one dimension |
| tf.reverse | To reverse a specific dimension of a *tensor* |
| tf.transpose | To transpose dimensions in a *tensor* |
| tf.gather | To collect portions according to an index |

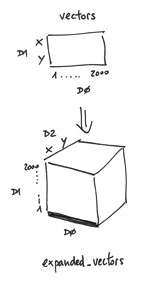
For example, suppose that you want to extend an array of 2×2000 (a 2D tensor) to a cube (3D tensor). We can use the *tf.expand\_ dims* function, which allows us to insert a dimension to a tensor:

vectors = tf.constant(conjunto\_puntos)

extended\_vectors = tf.expand\_dims(vectors, 0)

In this case, *tf.expand\_dims* inserts a dimension into a tensor in the one given in the argument (the dimensions start at zero).

Visually, the above transformation is as follows:



As you can see, we now have a 3D *tensor*, but we cannot determine the size of the new dimension D0 based on function arguments.

If we obtain the shape of this *tensor* with the *get\_shape()* operation, we can see that there is no associated size:

print expanded\_vectors.get\_shape()

It appears on the screen like:

TensorShape([Dimension(1), Dimension(2000), Dimension(2)])

Later in this chapter, we will see that, thanks to TensorFlow *shape broadcasting,*many mathematical manipulation functions of *tensors*(as presented in the first chapter), are able to discover for themselves the size in the dimension which unspecific size and assign to it this deduced value.

Data Storage in TensorFlow

Following the presentation of TensorFlow’s package, broadly speaking there are three main ways of obtaining data on a TensorFlow program:

1. From data files.
2. Data preloaded as constants or variables.
3. Those provided by Python code.

Below, I briefly describe each of them.

1. **Data files**

Usually, the initial data is downloaded from a data file. The process is not complex, and given the introductory nature of this book I invite the reader to visit the website of TensorFlow[[19]](http://www.jorditorres.org/first-contact-with-tensorflow/" \l "_ftn19) for more details on how to download data from different file types. You can also review the Python code [*input\_data.py*](https://github.com/jorditorresBCN/TutorialTensorFlow/blob/master/input_data.py)***[[20]](http://www.jorditorres.org/first-contact-with-tensorflow/" \l "_ftn20)***(available on the Github book), which loads the MNIST data from files (I will use this in the following chapters).

1. **Variables and constants**

When it comes to small sets, data can also be found pre-loaded into memory; there are two basic ways to create them, as we have seen in the previous example:

* As a constants using *constant(…)*
* As a variable using *Variable(…)*

TensorFlow package offers different operations that can be used to generate constants. In the table below you can find a summary of the most important:

|  |  |
| --- | --- |
| **Operation** | **Description** |
| tf.zeros\_like | Creates a tensor with all elements initialized to 0 |
| tf.ones\_like | Creates a tensor with all elements initialized to 1 |
| tf.fill | Creates a tensor with all elements initialized to a scalar value given as argument |
| tf.constant | Creates a tensor of constants with the elements listed as an arguments |

In TensorFlow, during the training process of the models, the parameters are maintained in the memory as variables. When a variable is created, you can use a tensor defined as a parameter of the function as an initial value, which can be a constant or a random value. TensorFlow offers a collection of operations that produce random tensors with different distributions:

|  |  |
| --- | --- |
| **Operation** | **Description** |
| tf.random\_normal | Random values with a normal distribution |
| tf.truncated\_normal | Random values with a normal distribution but eliminating those values whose magnitude is more than 2 times the standard deviation |
| tf.random\_uniform | Random values with a uniform distribution |
| tf.random\_shuffle | Randomly mixed tensor elements in the first dimension |
| tf.set\_random\_seed | Sets the random seed |

An important detail is that all of these operations require a specific shape of the tensors as the parameters of the function, and the variable that is created has the same shape. In general, the variables have a fixed shape, but TensorFlow provides mechanisms to reshape it if necessary.

When using variables, these must be explicitly initialized after the graph that has been constructed, and before any operation is executed with the *run()*function*.* As we have seen, it can be used *tf.initialize\_all\_variables()*for this purpose. Variables also can be saved onto disk during and after training model through TensorFlow *tf.train.Saver()* class, but this class is beyond the scope of this book.

1. **Provided by Python code**

Finally, we can use what we have called “symbolic variable” or *placeholder* to manipulate data during program execution. The call is *placeholder(),*which includes arguments with the type of the elements and the shape of the tensor, and optionally a name.

At the same time as making the calls to *Session.run()* or *Tensor.eval()* from the Python code, this tensor is populated with the data specified in the *feed\_dict* parameter. Remember the first code in Chapter 1:

import tensorflow as tf

a = tf.placeholder("float")

b = tf.placeholder("float")

y = tf.mul(a, b)

sess = tf.Session()

print   sess.run(y, feed\_dict={a: 3, b: 3})

In the last line of code, when the call *sess.run()* is made, it is when we pass the values of the two tensors *a* and *b* through *feed\_dict* parameter.

With this brief introduction about tensors, I hope that from now on the reader can follow the codes of the following chapters without any difficulty.

*K-means* algorithm

*K-means* is a type of unsupervised algorithm which solves the clustering problem. Its procedure follows a simple and easy way to classify a given data set through a certain number of clusters (assume k clusters). Data points inside a cluster are homogeneous and heterogeneous to peer groups, that means that all the elements in a subset are more similar to each other than with the rest.

The result of the algorithm is a set of K dots, called centroids, which are the focus of the different groups obtained, and the tag that represents the set of points that are assigned to only one of the K clusters. All the points within a cluster are closer in distance to the centroid than any of the other centroids.

Making clusters is a computationally expensive problem if we want to minimize the error function directly (what is known as an *NP-hard* problem); and therefore it some algorithms that converge rapidly in a local optimum by heuristics have been created. The most commonly used algorithm uses an iterative refinement technique, which converges in a few iterations.

Broadly speaking, this technique has three steps:

* **Initial step**(step 0): determines an initial set of K centroids.
* **Allocation step**(step 1): assigns each observation to the nearest group.
* **Update step**(step 2): calculates the new centroids for each new group.

There are several methods to determine initial K centroids. One of them is randomly choose K observations in the data set and consider them centroids; this is the one we will use in our example.

The steps of allocation (step 1) and updating (step 2) are being alternated in a loop until it is considered that the algorithm has converged, which may be for example when allocations of points to groups no longer change.

Since this is a heuristic algorithm, there is no guarantee that it converges to the global optimum, and the outcome depends on the initial groups. Therefore, as the algorithm is generally very fast, it is usual to repeat executions multiple times with different values of the initials centroides, and then weigh the result.

To start coding our example of *K-means* in TensorFlow I suggest to first generate some data as a testbed. I propose to do something simple, like generating 2,000 points in a 2D space in a random manner, following two normal distributions to draw up a space that allows us to better understand the outcome. For example, I suggest the following code:

num\_puntos = 2000

conjunto\_puntos = []

for i in xrange(num\_puntos):

if np.random.random() &gt; 0.5:

conjunto\_puntos.append([np.random.normal(0.0, 0.9), np.random.normal(0.0, 0.9)])

else:

conjunto\_puntos.append([np.random.normal(3.0, 0.5), np.random.normal(1.0, 0.5)])

As we have done in the previous chapter, we can use some Python graphic libraries to plot the data. I propose that we use *matplotlib* like before, but this time we will also use the visualization package *Seaborn* based on *matplotlib* and the data manipulation package *pandas*, which allows us to work with more complex data structures.

If you do not have these packages installed, you must do it with the *pip* value before you can run the following codes.

To display the points that have been generated randomly I suggest the following code:

import matplotlib.pyplot as plt

import pandas as pd

import seaborn as sns

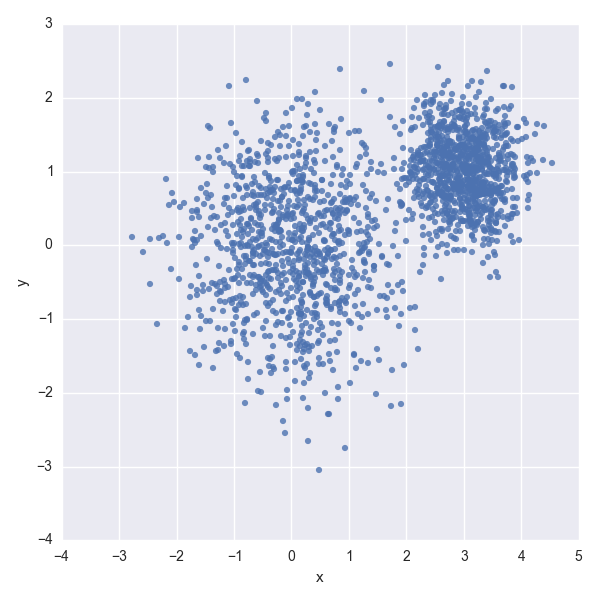
df = pd.DataFrame({"x": [v[0] for v in conjunto\_puntos],

"y": [v[1] for v in conjunto\_puntos]})

sns.lmplot("x", "y", data=df, fit\_reg=False, size=6)

plt.show()

This code generates a graph of points in a two dimensional space like the following screenshot:



A *k-means* algorithm implemented in TensorFlow to group the above points, for example in four clusters, can be as follows (based on the model proposed by Shawn Simister in his blog[[21]](http://www.jorditorres.org/first-contact-with-tensorflow/" \l "_ftn21)):

import numpy as np

vectors = tf.constant(conjunto\_puntos)

k = 4

centroides = tf.Variable(tf.slice(tf.random\_shuffle(vectors),[0,0],[k,-1]))

expanded\_vectors = tf.expand\_dims(vectors, 0)

expanded\_centroides = tf.expand\_dims(centroides, 1)

assignments = tf.argmin(tf.reduce\_sum(tf.square(tf.sub(expanded\_vectors, expanded\_centroides)), 2), 0)

means = tf.concat(0, [tf.reduce\_mean(tf.gather(vectors, tf.reshape(tf.where( tf.equal(assignments, c)),[1,-1])), reduction\_indices=[1]) for c in xrange(k)])

update\_centroides = tf.assign(centroides, means)

init\_op = tf.initialize\_all\_variables()

sess = tf.Session()

sess.run(init\_op)

for step in xrange(100):

\_, centroid\_values, assignment\_values = sess.run([update\_centroides, centroides, assignments])

I suggest the reader checks the result in the *assignment\_values* tensor with the following code, which generates a graph as above:

data = {"x": [], "y": [], "cluster": []}

for i in xrange(len(assignment\_values)):

data["x"].append(conjunto\_puntos[i][0])

data["y"].append(conjunto\_puntos[i][1])

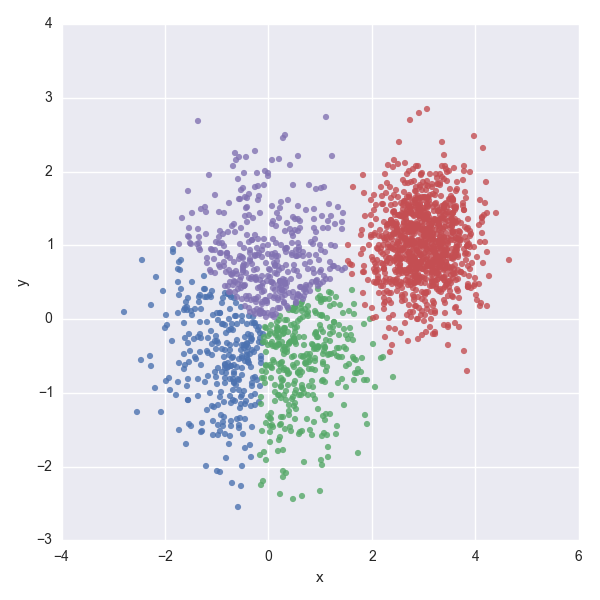
data["cluster"].append(assignment\_values[i])

df = pd.DataFrame(data)

sns.lmplot("x", "y", data=df, fit\_reg=False, size=6, hue="cluster", legend=False)

plt.show()

The screenshot with the result of the execution of my code it is shown in the following figure:

[](http://www.jorditorres.org/wp-content/uploads/2016/02/image026.png)

New groups

I assume that the reader might feel a little overwhelmed with the *K-means* code presented in the previous section. Well, I propose that we analyze this in detail, step by step, and especially watch the tensors invoved and how they are transformed during the program.

The first thing to do is move all our data to tensors. In a *constant tensor,* we keep our entry points randomly generated:

vectors = tf.constant(conjunto\_vectors)

Following the algorithm presented in the previous section, in order to start we must determine the initial centroids. As I advanced, an option may be randomly choose K observations from the input data. One way to do this is with the following code, which indicates to TensorFlow that it must shuffle randomly the entry point and choose the first K points as centroids:

k = 4

centroides = tf.Variable(tf.slice(tf.random\_shuffle(vectors),[0,0],[k,-1]))

These K points are stored in a 2D tensor. To know the shape of those tensors we can use *tf.Tensor.get\_shape()*:

print vectors.get\_shape()

print centroides.get\_shape()

TensorShape([Dimension(2000), Dimension(2)])

TensorShape([Dimension(4), Dimension(2)])

We can see that *vectors* is an array that dimension D0 contains 2000 positions, one for each, and D1 contains the position *x,y* for each point. Instead, *centroids* is a matrix of four positions in the dimension D0, one position for each centroid, and the dimension D1 is equivalent to the dimension D1 of *vectors.*

Next, the algorithm enters in a loop. The first step is to calculate, for each point, its closest centroid by the *Squared Euclidean Distance****[[22]](http://www.jorditorres.org/first-contact-with-tensorflow/" \l "_ftn22)*** (which can only be used when we want to compare distances):

image028

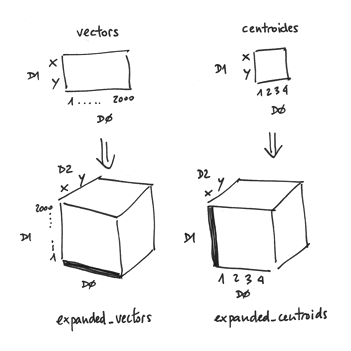
To calculate this value *tf.sub(vectors, centroides*) is used. We should note that, although the two subtract tensors have both 2 dimensions, they have different sizes in one dimension (2000 vs 4 in dimension D0), which, in fact, also represent different things.

To fix this problem we could use some of the functions discussed before, for instance *tf.expand\_dims* in order to insert a dimension in both tensors. The aim is to extend both tensors from 2 dimensions to 3 dimensions to make the sizes match in order to perform a subtraction:

expanded\_vectors = tf.expand\_dims(vectors, 0)

expanded\_centroides = tf.expand\_dims(centroides, 1)

*tf.expand\_dims* inserts one dimension in each tensor; in the first dimension (D0) of *vectors* tensor, and in the second dimension (D1) of *centroids* tensor. Graphically, we can see that in the extended tensors the dimensions have the same meaning in each of them:



It seems to be solved, but actually, if you look closely (outlined in bulk in the illustration), in each case there are dimensions that have not been able to determinate the sizes of those dimensions. Remember that the with *get\_shape()*function we can find out:

print expanded\_vectors.get\_shape()

print expanded\_centroides.get\_shape()

The output is as follows:

TensorShape([Dimension(1), Dimension(2000), Dimension(2)])

TensorShape([Dimension(4), Dimension(1), Dimension(2)])

With 1 it is indicating a no assigned size.

But I have already advanced that TensorFlow allows *broadcasting*, and therefore the *tf.sub* function is able to discover for itself how to do the subtraction of elements between the two tensors.

Intuitively, and observing the previous drawings, we see that the shape of the two tensors match, and in those cases both tensors have the same size in a certain dimension. These math, as happens in dimension D2. Instead, in the dimension D0 only has a defined size the *expanded\_centroides*.

In this case, TensorFlow assumes that the dimension D0 of *expanded\_vectors* tensor have to be the same size if we want to perform a subtraction element to element within this dimension.

And the same happens with the size of the dimension D1 of *expended\_centroides*tensor, where TensorFlow deduces the size of the dimension D1 of *expanded\_vectors* tensor.

Therefore, in the allocation step (step 1) the algorithm can be expressed in these four lines of TensorFlow´s code, which calculates the *Squared Euclidean Distance*:

diff=tf.sub(expanded\_vectors, expanded\_centroides)

sqr= tf.square(diff)

distances = tf.reduce\_sum(sqr, 2)

assignments = tf.argmin(distances, 0)

And, if we look at the shapes of tensors, we see that they are respectively for *diff, sqr, distances* and *assignments* as follows:

TensorShape([Dimension(4), Dimension(2000), Dimension(2)])

TensorShape([Dimension(4), Dimension(2000), Dimension(2)])

TensorShape([Dimension(4), Dimension(2000)])

TensorShape([Dimension(2000)])

That is, *tf.sub* function has returned the tensor *dist*, that contains the subtraction of the index values for centroids and vector (indicated in the dimension D1, and the centroid indicated in the dimension D0. For each index *x,y*are indicated in the dimension D2) .

The *sqr* tensor contains the square of those. In the *distance* tensor we can see that it has already reduced one dimension, the one indicated as a parameter in *tf.reduce\_sum* function.

I use this example to explain that TensorFlow provides several operations which can be used to perform mathematical operations that reduce dimensions of a tensor as in the case of *tf.reduce\_sum*. In the table below you can find a summary of the most important ones:

|  |  |
| --- | --- |
| **Operation** | **Description** |
| tf.reduce\_sum | Computes the sum of the elements along one dimension |
| tf.reduce\_prod | Computes the product of the elements along one dimension |
| tf.reduce\_min | Computes the minimum of the elements along one dimension |
| tf.reduce\_max | Computes the maximum of the elements along one dimension |
| tf.reduce\_mean | Computes the mean of the elements along one dimension |

Finally, the assignation is achieved with *tf.argmin*, which returns the index with the minimum value of the tensor *dimension* (in our case *D0*, which remember that was the centroid). We also have the *tf.argmax* operation:

|  |  |
| --- | --- |
| **Operation** | **Description** |
| tf.argmin | Returns the index of the element with the minimum value along *tensor* dimension |
| tf.argmax | Returns the index of the element with the maximum value of the *tensor* dimension |

In fact, the 4 instructions seen above could be summarized in only one code line, as we have seen in the previous section:

assignments = tf.argmin(tf.reduce\_sum(tf.square(tf.sub(expanded\_vectors, expanded\_centroides)), 2), 0)

But anyway, internal *tensors* and the operations that they define as nodes and execute the internal graph are like the ones we have described before.

Computation of the new centroids

Once we have created new groups on each iteration, we will have to remember that the new step of the algorithm consists in calculating the new centroids of the groups. In the code of the section before we have seen this line of code:

means = tf.concat(0, [tf.reduce\_mean(tf.gather(vectors, tf.reshape(tf.where( tf.equal(assignments, c)),[1,-1])), reduction\_indices=[1]) for c in xrange(k)])

On that piece of code, we can see that the *means*tensor is the result of the concatenation of the *k*tensors that correspond to the mean value of every point that belongs to each *k cluster*.

Next, I will comment on each of the TensorFlow operations that are involved in the calculation of the mean value of every points that belongs to each *cluster*[[23]](http://www.jorditorres.org/first-contact-with-tensorflow/" \l "_ftn23):

* With *equal* we can obtain a *boolean tensor* (*Dimension(2000))* that indicates (with *true*value) the positions where the *assignment tensor* match with the *K cluster*, which, at the time, we are calculating the average value of the points.
* With *where*is constructed a *tensor* (*Dimension(1) x Dimension(2000))* with the position where the values *true* are on the *boolean tensor* received as a parameter. i.e. a list of the position of these.
* With *reshape* is constructed a tensor*(Dimension(2000) x Dimension(1))* with the index of the points inside *vectors*tensor that belongs to this *c cluster*.
* With *gather*is constructed a *tensor (Dimension(1) x Dimension(2000))*which gathers the coordenates of the points that form the *c cluster.*
* With *reduce\_mean*it is constructed a tensor*(Dimension(1) x Dimension(2))* that contains the average value of all points that belongs to the cluster *c*.

Anyway, if the reader wants to dig deeper into the code, as I always say, you can find more info for each of these operations, with very illustrative examples, on the *TensorFlow* *API* page[[24]](http://www.jorditorres.org/first-contact-with-tensorflow/" \l "_ftn24).

Graph execution

Finally, we have to describe the part of the above code that corresponds to the loop and to the part that update the centroids with the new values of the *means*tensor.

To do this, we need to create an operator that assigns the value of the variable *means*tensor into *centroids* in a way than, when the operation *run()*is executed, the values of the updated centroids are used in the next iteration of the loop:

update\_centroides = tf.assign(centroides, means)

We also have to create an operator to initialize all of the variable before starting to run the graph:

init\_op = tf.initialize\_all\_variables()

At this point everything is ready. We can start running the graph:

sess = tf.Session()

sess.run(init\_op)

for step in xrange(num\_steps):

\_, centroid\_values, assignment\_values = sess.run([update\_centroides, centroides, assignments])

In this code, for each iteration, the centroids and the new allocation of clusters for each entry points are updated.

Notice that the code specifies three operators and it has to go look in the execution of the call *run()*, and running in this order. Since there are three values to search, *sess.run()* returns a data structure of three *numpy array* elements with the contents of the corresponding tensor during the training process.

As *update\_centroides* is an operation whose result is not the parameter that returns, the corresponding item in the return tuple contains nothing, and therefore be ruled out, indicating it with “\_” [[25]](http://www.jorditorres.org/first-contact-with-tensorflow/" \l "_ftn25).

For the other two values, the centroids and the assigning points to each cluster, we are interested in displaying them on screen once they have completed all *num\_steps* iterations.

We can use a simple print. The output is as it follows:

print centroid\_values

[[ 2.99835277e+00 9.89548564e-01]

[ -8.30736756e-01 4.07433510e-01]

[ 7.49640584e-01 4.99431938e-01]

[ 1.83571398e-03 -9.78474259e-01]]

I hope that the reader has a similar values on the screen, since this will indicate that he has successfully executed the proposed code in this chapter of the book.

I suggest that the reader tries to change any of the values in the code, before advancing. For example the num\_points, and especially the number of clustersk, and see how it changes the result in the *assignment\_values* tensor with the previous code that generates a graph.

Remember that in order to facilitate testing the code described in this chapter, it can be downloaded from Github[[26]](http://www.jorditorres.org/first-contact-with-tensorflow/" \l "_ftn26) . The name of the file that contains this code is *Kmeans.py.*

In this chapter we have advanced some knowledge of TensorFlow, especially on basic data structure *tensor,* from a code example in TensorFlow that implements a clustering algorithm *K-means.*

With this knowledge, we are ready to build a single layer neural network, step by step, with TensorFlow in the next chapter.