4. SINGLE LAYER NEURAL NETWORK IN TENSORFLOW

In the preface, I commented that one of the usual uses of Deep Learning includes pattern recognition. Given that, in the same way that beginners learn a programming language starting by printing “Hello World” on screen, in Deep Learning we start by recognizing hand-written numbers.

In this chapter I present how to build, step by step, a neural network with a single layer in TensorFlow. This neural network will recognize hand-written digits, and it’s based in one of the diverse examples of the beginner’s tutorial of Tensor-Flow[[27]](http://www.jorditorres.org/first-contact-with-tensorflow/" \l "_ftn27).

Given the introductory style of this book, I chose to guide the reader while simplifying some concepts and theoretical justifications at some steps through the example.

If the reader is interested in learn more about the theoretical concepts of this example after reading this chapter, I suggest to read *Neural Networks and Deep Learning*[[28]](http://www.jorditorres.org/first-contact-with-tensorflow/" \l "_ftn28), available online, presenting this example but going in depth with the theoretical concepts.

The MNIST data-set

The MNIST data-set is composed by a set of black and white images containing hand-written digits, containing more than 60.000 examples for training a model, and 10.000 for testing it. The MNIST data-set can be found at the *MNIST database*[[29]](http://www.jorditorres.org/first-contact-with-tensorflow/" \l "_ftn29).

This data-set is ideal for most of the people who begin with pattern recognition on real examples without having to spend time on data pre-processing or formatting, two very important steps when dealing with images but expensive in time.

The black and white images (bilevel) have been normalized into 20×20 pixel images, preserving the aspect ratio. For this case, we notice that the images contain gray pixels as a result of the [*anti-aliasing*](https://es.wikipedia.org/wiki/Antialiasing)[[30]](http://www.jorditorres.org/first-contact-with-tensorflow/" \l "_ftn30) used in the normalization algorithm (reducing the resolution of all the images to one of the lowest levels). After that, the images are centered in 28×28 pixel frames by computing the mass center and moving it into the center of the frame. The images are like the ones shown here:



Also, the kind of learning required for this example is *supervised learning*; the images are labeled with the digit they represent. This is the most common form of Machine Learning.

In this case we first collect a large data set of images of numbers, each labelled with its value. During the training, the model is shown an image and produces an output in the form of a vector of scores, one score for each category. We want the desired category to have the highest score of all categories, but this is unlikely to happen before training.

We compute an objective function that measures the error (as we did in previous chapters) between the output scores and the desired pattern of scores. The model then modifies its internal adjustable parameters , called weights, to reduce this error. In a typical Deep Learning system, there may be hundreds of millions of these adjustable weights, and hundreds of millions of labelled examples with which to train the machine. We will consider a smaller example in order to help the understanding of how this type of models work.

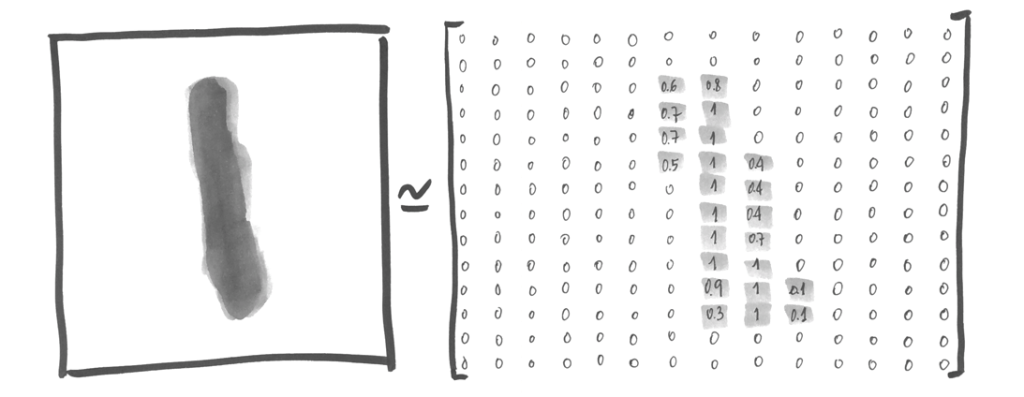
To download easily the data, you can use the script *input\_data.py*[[31]](http://www.jorditorres.org/first-contact-with-tensorflow/" \l "_ftn31), obtained from Google’s site [[32]](http://www.jorditorres.org/first-contact-with-tensorflow/" \l "_ftn32) but uploaded to the book’s *github* for your comodity. Simply download the code *input\_data.py* in the same work directory where you are programming the neural network with TensorFlow. From your application you only need to import and use in the following way:

import input\_data

mnist = input\_data.read\_data\_sets("MNIST\_data/", one\_hot=True)

After executing these two instructions you will have the full training data-set in *mnist.train* and the test data-set in *mnist.test*. As I previously said, each element is composed by an image, referenced as “*xs*”, and its corresponding label “*ys*”, to make easier to express the processing code. Remember that all data-sets, training and testing, contain “*xs*” and “*ys*”; also, the training images are referenced in *mnist.train.images* and the training labels in *mnist.train.labels*.

As previously explained, the images are formed by 28×28 pixels, and can be represented as a numerical matix. For example, one of the images of number 1 can be represented as:



Where each position indicates the level of lackness of each pixel between 0 and 1. This matrix can be represented as an array of 28×28 = 784 numbers. Actually, the image has been transformed in a bunch of points in a vectorial space of 784 dimensions. Only to mention that when we reduce the structure to 2 dimensions, we can be losing part of the information, and for some computer vision algorithms this could affect their result, but for the simplest method used in this tutorial this will not be a problem.

Summarizing, we have a *tensor* *mnist.train.images* in 2D in which calling the fuction *get\_shape()* indicates its shape:

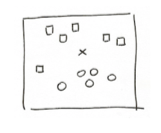
TensorShape([Dimension(60000), Dimension(784)])

The first dimension indexes each image and the second each pixel in each image. Each element of the tensor is the intensity of each pixel between 0 and 1.

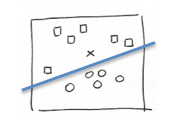
Also, we have the labels in the form of numbers between 0 and 9, indicating which digit each image represents. In this example, we are representing labels as a vector of 10 positions, which the corresponding position for the represented number contains a 1 and the rest 0. So *mnist.train.labels*es is a *tensor* shaped as *TensorShape([Dimension(60000), Dimension10)]).*

An artificial neutron

Although the book doesn’t focus on the theoretical concepts of neural netwoks, a brief and intuitive introduction of how neurons work to learn the training data will help the reader to undertand what is happening. Those readers that already know the theory and just seek how to use TensorFlow can skip this section.  
Let’s see a simple but illustrative example of how a neuron learns. Suppose a set of points in a plane labeled as “square” and “circle”. Given a new point “X”, we want to know which label corresponds to it:

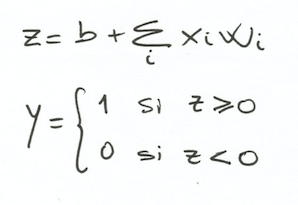


A usual approximation could be to draw a straight line dividing the two groups and use it as a classifier:



In this situation, the input data is represented by vectors shaped as (x,y) representing the coordinates in this 2-dimension space, and our function returning ‘0’ or ‘1’ (above or below the line) to know how to classify it as a “square” or “circle”. Mathematically, as we learned in the linear regression chapter, the “line” (classifier) can be expressed as *y= W\*x+b*.

Generalizing, a neuron must learn a weight *W* (with the same dimension as the input data X) and an offset *b* (called *bias* in neural networks) to learn how to classify those values. With them, the neuron will compute a weighted sum of the inputs in X using weight W, and add the offset *b*; and finally the neuron will apply an “activation” non-linear function to produce the result of “0” or “1”.  
The function of the neuron can be more formally expressed as:

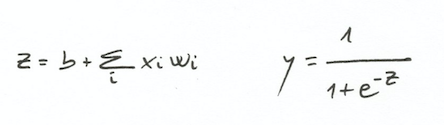


Having defined this function for our neuron, we want to know how the neuron can learn those parameters *W* and *b* from the labeled data with “squares” and “circles” in our example, to later label the new point “X”.

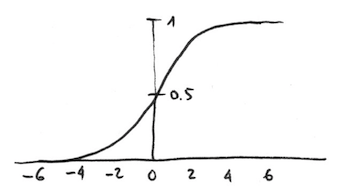
A first approach could be similar to what we did with the linear regression, this is, feed the neuron with the known labeled data, and compare the obtained result with the real one. Then, when iterating, weights in *W* and *b* are adjusted to minimize the error, as shown again in chapter 2 for the linear regression line.

Once we have the *W* and *b* parameters we can compute the weighted sum, and now we need the function to turn the result stored in *z* into a ‘0’ or ‘1’. There are several activation functions available, and for this example we can use a popular one called *sigmoid*[[33]](http://www.jorditorres.org/first-contact-with-tensorflow/" \l "_ftn33), returning a real value between 0 and 1

:

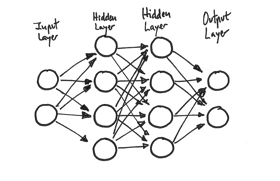


Looking at the formula we see that it will tend to return values close to 0 or 1. If input *z* is big enough and positive, “e” powered to minus z is zero and then *y* is 1. If the input *z* is big enough and negative, “e” powered to a large positive number becomes also a large positive number, so the denominator becomes large and the final *y* becomes 0. If we plot the function it would look like this:



Since here we have presented how to define a neuron, but a neural network is actually a composition of neurons connected among them in a different ways and using different activation functions. Given the scope of this book, I’ll not enter into all the extension of the neural networks universe, but I assure you that it is really exciting.

Just to mention that there is a specific case of neural networks (in which Chapter 5 is based on) where the neurons are organized in layers, in a way where the inferior layer (input layer) receives the inputs, and the top layer (output layer) produces the response values. The neural network can have several intermediate layers, called hidden layers. A visual way to represent this is:



In these networks, the neurons of a layer communicate to the neurons of the previous layer to receive information, and then communicate their results to the neurons of the next layer.

As previously said, there are more activation functions apart of the *Sigmoid*, each one with different properties. For example, when we want to classify data into more than two classes at the output layer, we can use the *Softmax*[[34]](http://www.jorditorres.org/first-contact-with-tensorflow/" \l "_ftn34) activation function, a generalization of the *sigmoid* function. *Softmax* allows obtaining the probability of each class, so their sum is 1 and the most probable result is the one with higher probability.

A easy example to start: Softmax

Remember that the problem to solve is that, given an input image, we get the probability that it belongs to a certain digit. For example, our model could predict a “9” in an image with an 80% certainty, but give a 5% of chances to be an “8” (due to a dubious lower trace), and also give certain low probabilities to be any other number. There is some uncertainty on recognizing hand-written numbers, and we can’t recognize the digits with a 100% of confidence. In this case, a probability distribution gives us a better idea of how much confidence we have in our prediction.

So, we have an output vector with the probability distribution for the different output labels, mutully exclusive. This is, a vector with 10 probability values, each one corresponding to each digit from 0 to 9, and all probabilities summing 1.

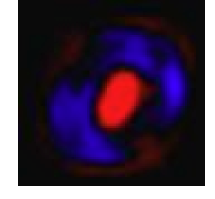
As previously said, we get to this by using an output layer with the *softmax* activation function. The output of a neuron with a *softmax* function depends on the output of the other neurons of its layer, as all of their outputs must sum 1.

The *softmax* function has two main steps: first, the “evidences” for an image belonging to a certain label are computed, and later the evidences are converted into probabilities for each possible label.

Evidence of belonging

Measuring the evidence of a certain image to belong to a specific class/label, a usual approximation is to compute the weighted sum of pixel intensities. That weight is negative when a pixel with high intensity happens to not to be in a given class, and positive if the pixel is frequent in that class.

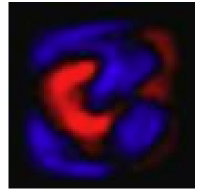
Let’s show a graphical example: suppose a learned model for the digit “0” (we will see how this is learned later). At this time, we define a model as “something” that contains information to know whether a number belongs to a specific class. In this case, we chose a model like the one below, where the red (or bright gray for the b/n edition) represents negative examples (this is, reduce the support for those pixels present in “0”), while the blue (the darker gray for b/n edition) represents the positive examples. Look at it:



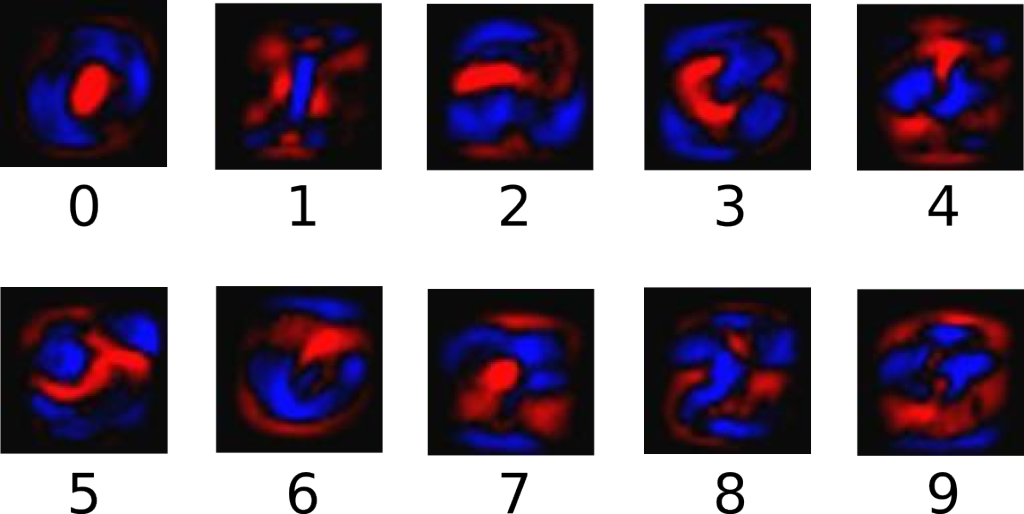
Think in a white paper sheet of 28×28 pixels and draw a “0” on it. Generally our zero would be drawn in the blue zone (remember that we left some space around the 20×20 drawing zone, centering it later).

It is clear that in those cases where our drawing crosses the red zone, it is most probably that we are not drawing a zero. So, using a metric based in rewarding those pixels, stepping on the blue region and penalizing those stepping the red zone seems reasonable.

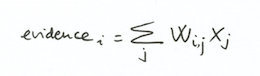
Now think of a “3”: it is clear that the red zone of our model for “0” will penalize the probabilities of it being a “0”. But if the reference model is the following one, generally the pixels forming the “3” will follow the blue region; also the drawing of a “0” would step into a red zone.



I hope that the reader, seeing those two examples, understands how the explained approximation allows us to estimate which number represent those drawings.  
The following figure shows an example of the ten different labels/classes learned from the MNIST data-set (extracted from the examples of Tensorflow[[35]](http://www.jorditorres.org/first-contact-with-tensorflow/" \l "_ftn35)). Remember that red (bright gray) represents negative weights and blue (dark gray) represents the positive ones.

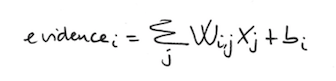


In a more formal way, we can say that the evidence for a class *i* given an input *x* is expressed as:

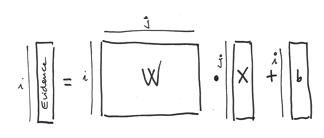


Where *i* indicates the class (in our situation, between 0 and 9), and *j* is an index to sum the indexes of our input image. Finally *Wi* represents the aforementioned weights.

Remember that, in general, the models also include an extra parameter representing the bias, adding some base uncertainty. In our situation, the formula would end like this:

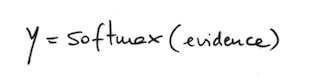


For each *i* (between 0 and 9) we have a matrix Wi of 784 elements (28×28), where each element *j* is multiplied by the corresponding component *j* of the input image, with 784 components, then added *bi*. A graphical view of the matrix calculus and indexes is this:

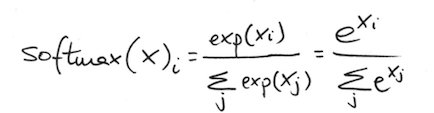


Probability of belonging

We commented that the second step consisted on computing probabilities. Specifically we turn the sum of evidences into predicted probabilities, indicated as *y*, using the *softmax* function:



Remember that the output vector must be a probability function with sum equals 1. To normalize each component, the softmax function uses the exponential value of each of its inputs and then normalizes them as follows:

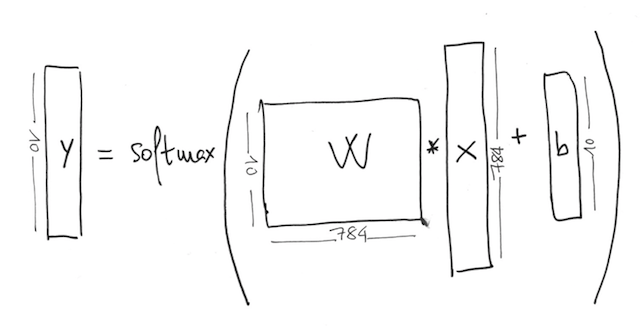


The obtained effect when using exponentials is a multiplication effect in weights. Also, when the evidence for a class is small, this class support is reduced by a fraction of its previous weight. Furthermore, *softmax* normalizes the weights making them to sum 1, creating a probability distribution.

The interesting fact of such function is that a good prediction will have one output with a value near 1, while all the other outputs will be near zero; and in a *weak* prediction, some labels may show similar support.

Programming in TensorFlow

After this brief description of what the algorithm does to recognize digits, we can implement it in TensorFlow. For this, we can make a quick look at how the *tensors* should be to store our data and the parameters for our model. For this purpose, the following schema depicts the data structures and their relations (to help the reader recall easily each piece of our problem):

First of all we create two variables to contain the weights *W* and the bias *b:*

W = tf.Variable(tf.zeros([784,10]))

b = tf.Variable(tf.zeros([10]))

Those variables are created using the *tf.Variable* function and the initial value for the variables; in this case we initialize the tensor with a constant tensor containing zeros.

We see that W is shaped as *[Dimension(784), Dimension(10)],* defined by its argument, a constant tensor *tf.zeros[784,10]* dimensioned like *W*. The same happens with the bias *b*, shaped by its argument as *[Dimension(10)]*.

Matrix W has that size because we want to multiply the image vector of 784 positions for each one of the 10 possible digits, and produce a tensor of evidences after adding *b*.

In this case of study using MNIST, we also create a tensor of two dimensions to keep the information of the *x* points, with the following line of code:

x = tf.placeholder("float", [None, 784])

The tensor *x* will be used to store the MNIST images as a vector of 784 floating point values (using *None* we indicate that the dimension can be any size; in our case it will be equal to the number of elements included in the learning process).

Now we have the tensors defined, and we can implement our model. For this, TensorFlow provides several operations, being *tf.nn.softmax(logits, name=None)* one of the available ones, implementing the previously described *softmax* function. The arguments must be a tensor, and optionally, a name. The function returns a tensor of the same kind and shape of the tensor passed as argument.

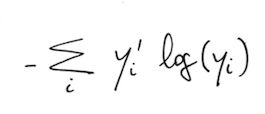
In our case, we provide to this function the resulting tensor of multiplying the image vector *x* and the weight matrix *W,* adding *b*:

y = tf.nn.softmax(tf.matmul(x,W) + b)

Once specified the model implementation, we can specify the necessary code to obtain the weights for *W* and bias *b* using an iterative training algorithm. For each iteration, the training algorithm gets the training data, applies the neural network and compares the obtained result with the expected one.

To decide when a model is good enough or not we have to define what “good enough” means. As seen in prevous chapters, the usual methodology is to define the opposite: how “bad” a model is using *cost* functions. In this case, the goal is to obtain values of *W* and *b* that minimizes the function indicating how “bad” the model is.

There are different metrics for the degree of error between resulting outputs and expected outputs on training data. One common metric is the *mean squared error* or the *squared Euclidean distance*, which have been previously seen. Even though, some research lines propose other metrics for such purpose in neural networks, like the *cross entropy error*, used in our example. This metric is computed like this:



Where *y* is the predicted distribution of probability and y’ is the real distribution, obtained from the labeling of the training data-set. We will not enter into details of the maths behind the cross-entropy and its place in neural networks, as it is far more complex than the intended scope of this book; just indicate that the minimum value is obtained when the two distributions are the same. Again, if the reader wants to learn the insights of this function, we recommend reading *Neural Networks and Deep Learning*[[36]](http://www.jorditorres.org/first-contact-with-tensorflow/" \l "_ftn36).

To implement the *cross-entropy* measurement we need a new placeholder for the correct labels:

y\_ = tf.placeholder("float", [None,10])

Using this placeholder, we can implement the *cross-entropy* with the following line of code, representing our cost function:

cross\_entropy = -tf.reduce\_sum(y\_\*tf.log(y))

First, we calculate the logarithm of each element *y* with the built-in function in TensorFlow *tf.log()*, and then we multiply it for each *y\_* element. Finally, using *tf.reduce\_sum* we sum all the elements of the tensor (later we will see that the images are visited in bundles, and in this case the value of *cross-entropy* corresponds to the bundle of images *y* and not a single image).

Iteratively, once determined the error for a sample, we have to correct the model (modifying the parameters *W* and *b*, in our case) to reduce the difference between computed and expected outputs in the next iteration.

Finally, it only remains to specify this iterative minimization process. There are several algorithms for this purpose in neural networks; we will use the *backpropagation* (backward propagation of errors) algorithm, and as its name indicates, it propagates backwards the error obtained at the outputs to recompute the weights of *W*, especially important for multi-layer neural network.

This method is used together with the previously seen gradient descent method, which using the cross-entropy cost function allows us to compute how much the parameters must change on each iteration in order to reduce the error using the available local information at each moment. In our case, intuitively it consist on changing the weights *W* a little bit on each iteration (this little bit expressed by a *learning rate* hyperparameter, indicating the speed of change) to reduce the error.

Due that in our case we only have one layer neural network we will not enter into backpropagation methods. Only remember that TensorFlow knows the entire computing graph, allowing it to apply optimization algorithms to find the proper gradients for the cost function to train the model.

So, in our example using the MNIST images, the following line of code indicates that we are using the backpropagation algorithm to minimize the *cross-entropy* using the *gradient descent* algorithm and a l*earning rate* of 0.01:

train\_step = tf.train.GradientDescentOptimizer(0.01).minimize(cross\_entropy)

Once here, we have specified all the problem and we can start the computation by instantiating *tf.Session()* in charge of executing the TensorFlow operations in the available devices on the system, CPUs or GPUs:

sess = tf.Session()

Next, we can execute the operation initializing all the variables:

sess.run(tf.initialize\_all\_variables())

From this moment on, we can start training our model. The returning parameter for *train\_step*, when executed, will apply the gradient descent to the involved parameters. So training the model can be achieved by repeating the *train\_step* execution. Let’s say that we want to iterate 1.000 times our *train\_step*; we have to indicate the following lines of code:

for i in range(1000):

batch\_xs, batch\_ys = mnist.train.next\_batch(100)

sess.run(train\_step, feed\_dict={x: batch\_xs, y\_: batch\_ys})

The first line inside the loop specifies that, for each iteration, a bundle of 100 inputs of data, randomly sampled from the training data-set, are picked. We could use all the training data on each iteration, but in order to make this first example more agile we are using a small sample each time. The second line indicates that the previously obtained inputs must feed the respective placeholders.

Finally, just mention that the Machine Learning algorithms based in gradient descent can take advantage from the capabilities of automatic differentiation of TensorFlow. A TensorFlow user only has to define the computational architecture of the predictive model, combine it with the target function, and then just add the data.

TensorFlow already manages the associated calculus of derivatives behind the learning process. When the *minimize()* method is executed, TensorFlow identifies the set of variables on which *loss* *function* depends, and computes gradients for each of these. If you are interested to know how the differentiation is implemented you can inspect the *ops/gradients.py* file [[37]](http://www.jorditorres.org/first-contact-with-tensorflow/" \l "_ftn37).

Model evaluation

A model must be evaluated after training to see how much “good” (or “bad”) is. For example, we can compute the percentage of hits and misses in our prediction, seeing which examples were correctly predicted. In previous chapters, we saw that the *tf.argmax(y,1)* function returns the index of the highest value of a tensor according a given axis. In effect, *tf.argmax(y,1)*is the label in our model with higher probability for each input, while *tf.argmax(y\_,1)* is the correct label. Using *tf.equal* method we can compare if our prediction coincides with the correct label:

correct\_prediction = tf.equal(tf.argmax(y,1), tf.argmax(y\_,1))

This instruction returns a list of Booleans. To determine which fractions of predictions are correct, we can cast the values to numeric variables (floating point) and do the following operation:

accuracy = tf.reduce\_mean(tf.cast(correct\_prediction, "float"))

For example, *[True, False, True, True]* will turn into *[1,0,1,1]* and the average will be 0.75 representing the percentage of *accuracy*. Now we can ask for the accuracy of our test data-set using the *mnist.test* as the *feed\_dict* argument:

print sess.run(accuracy, feed\_dict={x: mnist.test.images, y\_: mnist.test.labels})

I obtained a value around 91%. Are these results good? I think that they are fantastic, because this means that the reader has been able to program and execute her first neural network using TensorFlow.

Another problem is that other models may provide better accuracy, and this will be presented in the next chapter with a neural network containing more layers.

The reader will find the whole code used in this chapter in the file *RedNeuronalSimple.py*, in the book’s *github*[[38]](http://www.jorditorres.org/first-contact-with-tensorflow/" \l "_ftn38). Just to provide a global view of it, I’ll put it here together:

import input\_data

mnist = input\_data.read\_data\_sets("MNIST\_data/", one\_hot=True)

import tensorflow as tf

x = tf.placeholder("float", [None, 784])

W = tf.Variable(tf.zeros([784,10]))

b = tf.Variable(tf.zeros([10]))

matm=tf.matmul(x,W)

y = tf.nn.softmax(tf.matmul(x,W) + b)

y\_ = tf.placeholder("float", [None,10])

cross\_entropy = -tf.reduce\_sum(y\_\*tf.log(y))

train\_step = tf.train.GradientDescentOptimizer(0.01).minimize(cross\_entropy)

sess = tf.Session()

sess.run(tf.initialize\_all\_variables())

for i in range(1000):

batch\_xs, batch\_ys = mnist.train.next\_batch(100)

sess.run(train\_step, feed\_dict={x: batch\_xs, y\_: batch\_ys})

correct\_prediction = tf.equal(tf.argmax(y,1), tf.argmax(y\_,1))

accuracy = tf.reduce\_mean(tf.cast(correct\_prediction, "float"))

print sess.run(accuracy, feed\_dict={x: mnist.test.images, y\_: mnist.test.labels})

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