5. MULTI-LAYER NEURAL NETWORKS IN TENSORFLOW

In this chapter I will program, with the reader, a simple Deep Learning neural network using the same MNIST digit recognition problem of the previous chapter.

As I have advanced, a Deep Learning neural network consists of several layers stacked on top of each other. Specifically, in this chapter we will build a convolutional network, this is, an archetypal example of Deep Learning. Convolution neural networks were introduced and popularized in 1998 by Yann LeCunn and others. These convolutional networks have recently led state of the art performance in image recognition; for example: in our case of digit recognition they achieve an accuracy higher than 99%.

In the rest of this chapter, I will use an example code as the backbone, alongside which I will explain the two most important concepts of these networks: convolutions and pooling without entering in the details of the parameters, given the introductory nature of this book. However, the reader will be able to run all the code and I hope that it will allow you understand to global ideas behind convolutional networks.

Convolutional Neural Networks

Convolutional Neural Nets (also known as CNN’s or CovNets) are a particular case of Deep Learning and have had a significant impact in the area of computer vision.

A typical feature of CNN’s is that they nearly always have images as inputs, this allows for more efficient implementation and a reduction in the number of required parameters. Let’s have a look at our MNIST digit recognition example: after reading in the MNIST data and defining the placeholders using TensorFlow as we did in the previous example:

import input\_data

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

import tensorflow as tf

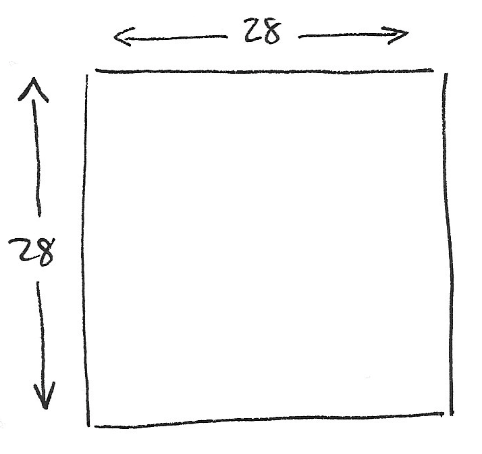
x = tf.placeholder("float", shape=[None, 784])

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

We can reconstruct the original shape of the images of the input data. We can do this as follows:

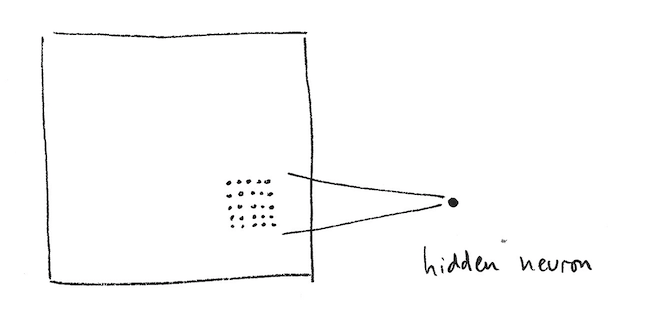
x\_image = tf.reshape(x, [-1,28,28,1])

Here we changed the input shape to a 4D tensor, the second and third dimension correspond to the width and the height of the image while the last dimension corresponding number of color channels, 1 in this case.  
This way we can think of the input to our neural network being a 2 dimensional space of neurons with size of 28×28 as depicted in the figure



There are two basic principles that define convolution neural networks: the filters and the characteristic maps. These principles can be expressed as groups of specialized neurons, as we will see shortly. But first, we will give a short description of these two principles given their importance in CNN’s.

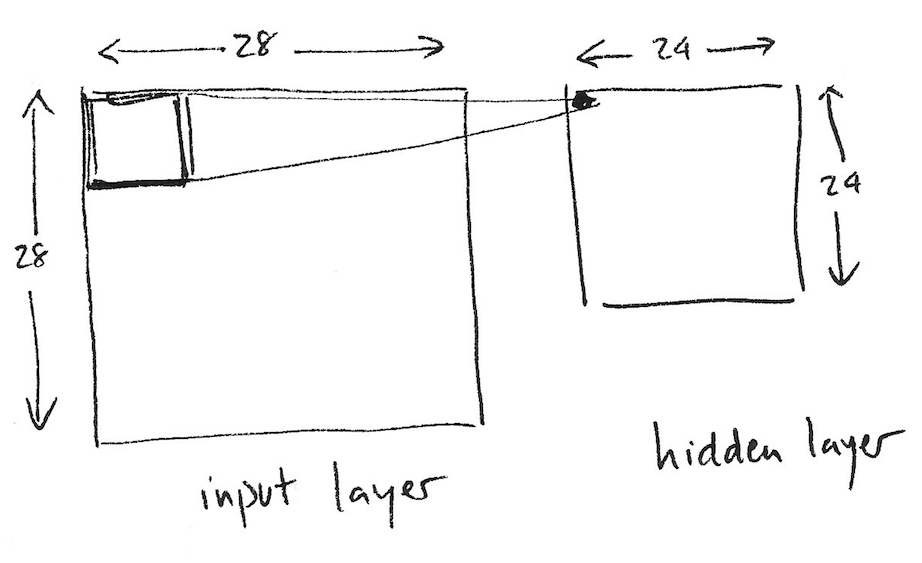
Intuitively, we could say that the main purpose of a convolutional layer is to detect characteristics or visual features in the images, think of edges, lines, blobs of color, etc. This is taken care of by a hidden layer that is connected by to input layer that we just discussed. In the case of CNN’s, in which we are interested, the input data is not fully connected to the neurons of the first hidden layer; this only happens in a small localized space in the input neurons that store the pixels values of the image. This can be visualized as follows:



To be more precise, in the given example each neuron of our hidden layer is connected with a small 5×5 region (hence 25 neurons) of the input layer.

We can think of this being a window of size 5×5 that passes over the entire input layer of size 28×28 that contains the input image. The window slides over the entire layer of neurons. For each position of the window there is a neuron in the hidden layer that processes that information.

We can visualize this by assuming that the window starts in the top left corner of the image; this provides the information to the first neuron of the hidden layer.  The window is then slid right by one pixel; we connect this 5×5 region with the second neuron in the hidden layer. We continue like this until the entire space from top to bottom and from left to right has been covered by the window.



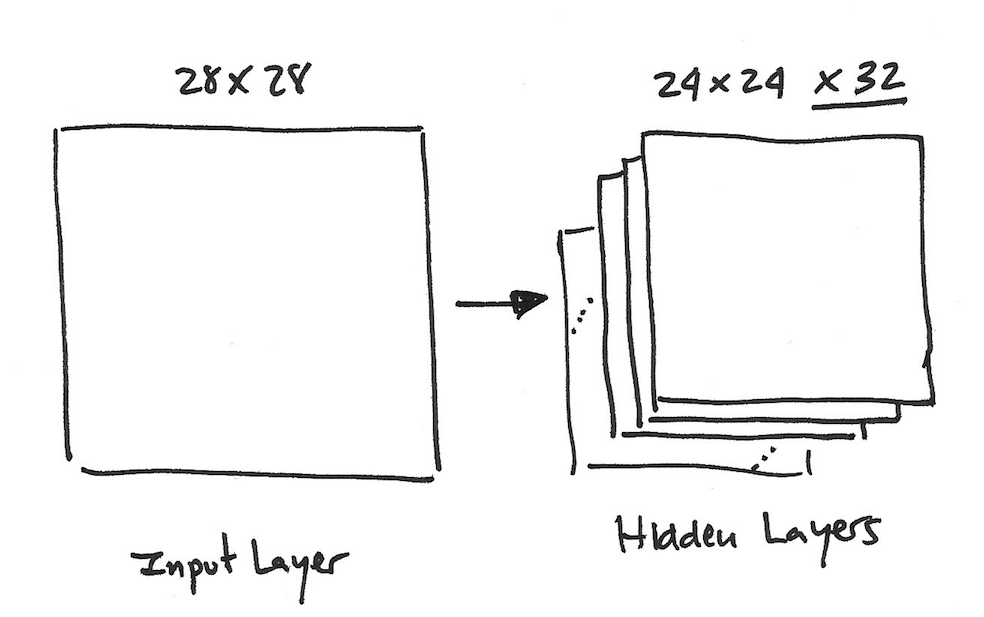
Analyzing the concrete case that we have proposed, we observe that given an input image of size 28×28 and a window of size 5×5 leads to a 24×24 space of neurons in the first hidden layer due to the fact we can only move the window 23 times down and 23 times to the right before hitting the bottom right edge of the input image. This assumes that window is moved just 1 pixel each time, so the new window overlaps with the old one expect in line that has just advanced.

It is, however, possible to move more than 1 pixel at a time in a convolution layer, this parameter is called the ‘stride’ length. Another extension is to pad the edges with zeroes (or other values) so that the window can slide over the edge of the image, which may lead to better results. The parameter to control this feature is known as padding [[39]](http://www.jorditorres.org/first-contact-with-tensorflow/" \l "_ftn39), with which you can determine the size of the padding. Given the introductory nature of the book we will not go into further detail about these two parameters.

Given our case of study, and following the formalism of the previous chapter, we will need a bias value *b* and a 5×5 weight matrix *W* to connect the neurons of the hidden layer with the input layer. A key feature of a CNN is that this weight matrix *W* and bias *b* are shared between all the neurons in the hidden layer; we use the same *W* and *b* for neurons in the hidden layer. In our case that is 24×24 (576) neurons. The reader should be able to see that this drastically reduces the amount weight parameters that one needs when compared to a fully connected neural network. To be specific, this reduces from 14000 (5x5x24x24) to just 25 (5×5) due to the sharing of the weight matrix *W*.

This shared matrix *W* and the bias *b* are usually called a kernel or filter in the context of CNN’s. These filters are similar to those used image processing programs for retouching images, which in our case are used to find discriminating features. I recommend looking at the examples found in the GIMP [[40]](http://www.jorditorres.org/first-contact-with-tensorflow/" \l "_ftn40) manual to get a good idea on how process of convolution works.

A matrix and a bias define a kernel. A kernel only detects one certain relevant feature in the image so it is, therefore, recommended to use several kernels, one for each characteristic we would like to detect. This means that a full convolution layer in a CNN consists of several kernels.  The usual way of representing several kernels is as follows:



The first hidden layer is composed by several kernels. In our example, we use 32 kernels, each one defined by a 5×5 weight matrix W and a bias b that is also shared between the neuros of the hidden layer.

In order to simplify the code, I define the following two functions related to the weight matrix W and bias b:

def weight\_variable(shape):

initial = tf.truncated\_normal(shape, stddev=0.1)

return tf.Variable(initial)

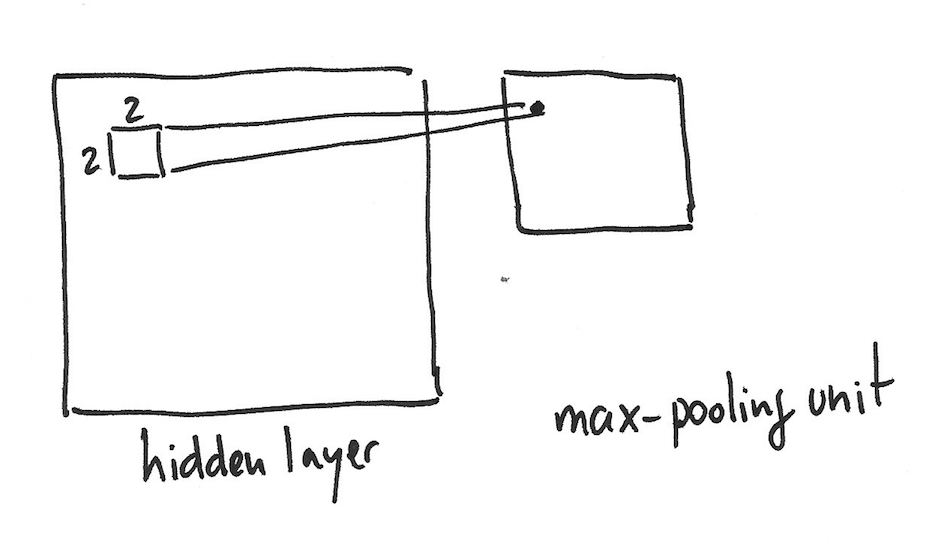
def bias\_variable(shape):

initial = tf.constant(0.1, shape=shape)

return tf.Variable(initial)

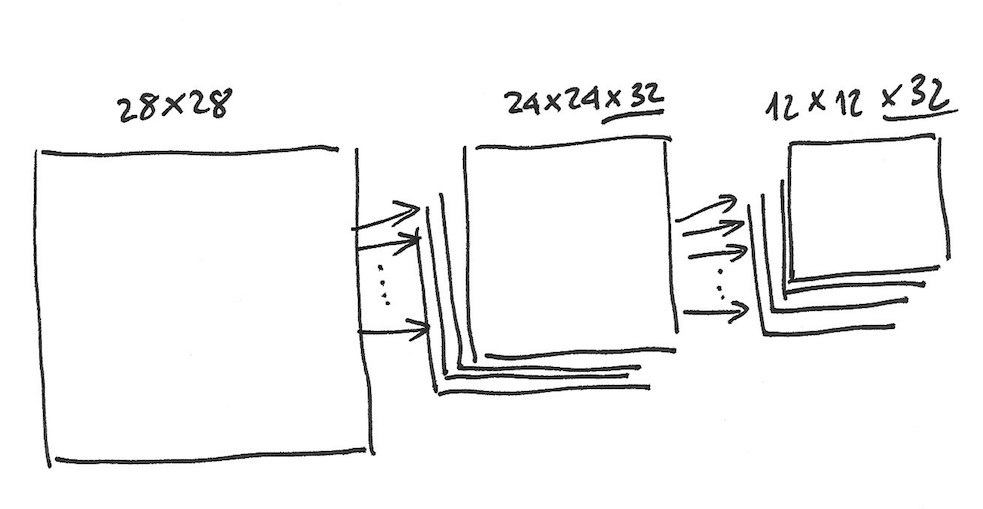
Without going into the details, it is customary to initialize the weights with some random noise and the bias values slightly positive.

In addition to the convolution layers that we just described, it is usual for the convolution layer to be followed by a so called pooling layer. The pooling layer simply condenses the output from the convolutional layer and creates a compact version of the information that have been put out by the convolutional layer. In our example, we will use a 2×2 region of the convolution layer of which we summarize the data into a single point using pooling:



There are several ways to perform pooling to condense the information; in our example we will use the method called *max-pooling*. This consists of condensing the information by just retaining the maximum value in the 2×2 region considered.

As mentioned above, the convolutional layer consists of many kernels and, therefore, we will apply *max-pooling* to each of those separately. In general, there can be many layers of pooling and convolutions:



This leads that the 24×24 convolution result is transformed to a 12×12 space by the max-pooling layer that correspond to the 12×12 tiles, of which each originates from a 2×2 region. Note that, unlike in the convolutional layer, the data is tiled and not created by a sliding window.

Intuitively, we can explain *max-pooling* as finding out if a particular feature is present anywhere in the image, the exact location of the feature is not as important as the relative location with respect to others features.

Implementation of the model

In this section, I will present the example code on how to write a CNN based on the advanced example (Deep MNIST for experts) that can be found on the TensorFlow [[41]](http://www.jorditorres.org/first-contact-with-tensorflow/" \l "_ftn41) website. As I said in the beginning, there are many details of the parameters that require a more detailed treatment and theoretical approach than the given in this book. I hence will only give an overview of the code without going into to many details of the TensorFlow parameters.

As we have already seen, there are several parameters that we have to define for the convolution and pooling layers.  We will use a stride of size 1 in each dimension (this is the step size of the sliding window) and a zero padding model. The pooling that we will apply will be a max-pooling on block of 2×2. Similar to above, I propose using the following two generic functions to be able to write a cleaner code that involves convolutions and max-pooling.

def conv2d(x, W):

return tf.nn.conv2d(x, W, strides=[1, 1, 1, 1], padding='SAME')

def max\_pool\_2x2(x):

return tf.nn.max\_pool(x, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='SAME')

Now it is time to implement the first convolutional layer followed by a pooling layer. In our example we have 32 filters, each with a window size of 5×5. We must define a tensor to hold this weight matrix W with the shape [5, 5, 1, 32]: the first two dimensions are the size of the window, and the third is the amount of channels, which is 1 in our case. The last one defines how many features we want to use. Furthermore, we will also need to define a bias for every of 32 weight matrices. Using the previously defined functions we can write this in TensorFlow as follows:

W\_conv1 = weight\_variable([5, 5, 1, 32])

b\_conv1 = bias\_variable([32])

The ReLU (Rectified Linear unit) activation function has recently become the default activation function used in the hidden layers of *deep neural networks*. This simple function consist of returning *max(0, x)*, so it return 0 for negative values and x otherwise.  In our example, we will use this activation function in the hidden layers that follow the convolution layers.

The code that we are writing will first apply the convolution to the input images *x\_image*, which returns the results of the convolution of the image in a 2D tensor *W\_conv1* and then it sums the bias to which finally the ReLU activation function is applied. As a final step, we apply *max-pooling* to the output:

h\_conv1 = tf.nn.relu(conv2d(x\_image, W\_conv1) + b\_conv1)

h\_pool1 = max\_pool\_2x2(h\_conv1)

When constructing a deep neural network, we can stack several layers on top of each other. To demonstrate how to do this, I will create a secondary convolutional layer with 64 filters with a 5×5 window. In this case we have to pass 32 as the number of channels that we need as that is the output size of the previous layer:

W\_conv2 = weight\_variable([5, 5, 32, 64])

b\_conv2 = bias\_variable([64])

h\_conv2 = tf.nn.relu(conv2d(h\_pool1, W\_conv2) + b\_conv2)

h\_pool2 = max\_pool\_2x2(h\_conv2)

The resulting output of the convolution has a dimension of 7×7 as we are applying the 5×5 window to a 12×12 space with a *stride* size of 1. The next step will be to add a fully connected layer to 7×7 output, which will then be fed to the final *softmax* layer like we did in the previous chapter.

We will use a layer of 1024 neurons, allowing us to to process the entire image. The tensors for the weights and biases are as follows:

W\_fc1 = weight\_variable([7 \* 7 \* 64, 1024])

b\_fc1 = bias\_variable([1024])

Remember that the first dimension of the *tensor* represents the 64 filters of size 7×7 from the second convolutional layer, while the second parameter is the amount of neurons in the layer and is free to be chosen by us (in our case 1024).

Now, we want to flatten the *tensor* into a vector. We saw in the previous chapter that the *softmax* needs a flattened image in the form as a vector as input. This is achieved by multiplying the weight matrix *W\_fc1*with the flattend vector, adding the bias *b\_fc1* after wich we apply the *ReLU*activation function:

h\_pool2\_flat = tf.reshape(h\_pool2, [-1, 7\*7\*64])

h\_fc1 = tf.nn.relu(tf.matmul(h\_pool2\_flat, W\_fc1) + b\_fc1)

The next step will be to reduce the amount of effective parameters in the neural network using a technique called *dropout*. This consists of removing nodes and their incoming and outgoing connections. The decision of which neurons to drop and which to keep is decided randomly.  To do this in a consistent manner we will assign a probability to the neurons being dropped or not in the code.

Without going into too many details, *dropout* reduces the risk of the model of *overfitting* the data. This can happen when the hidden layers have large amount of neurons and thus can lead to very expressive models; in this case it can happen that random noise (or error) is modelled. This is known as *overfitting*, which is more likely if the model has a lot of parameters compared to the dimension of the input. The best is to avoid this situation, as overfitted models have poor predictive performance.

In our model we apply *dropout*, which consists of using the function *dropout tf.nn.dropout* before the final *softmax* layer. To do this we construct a *placeholder* to store the probability that a neuron is maintained during dropout:

keep\_prob = tf.placeholder("float")

h\_fc1\_drop = tf.nn.dropout(h\_fc1, keep\_prob)

Finally, we add the *softmax* layer to our model like have been done in the previous chapter. Remember that *sofmax* returns the probability of the input belonging to each class (digits in our case) so that the total probability adds up to 1. The *softmax* layer code is as follows:

W\_fc2 = weight\_variable([1024, 10])

b\_fc2 = bias\_variable([10])

y\_conv=tf.nn.softmax(tf.matmul(h\_fc1\_drop, W\_fc2) + b\_fc2)

Training and evaluation of the model

We are now ready to train the model that we have just defined by adjusting all the weights in the convolution, and fully connected layers to obtain the predictions of the images for which we have a label. If we want to know how well our model performs we must follow the example set in the previous chapter.

The following code is very similar to the one in the previous chapter, with one exception: we replace the *gradient descent optimizer* with *the ADAM optimizer*, because this algorithm implements a different optimizer that offers certain advantages according to the literature [[42]](http://www.jorditorres.org/first-contact-with-tensorflow/" \l "_ftn42).

We also need to include the additional parameter *keep\_prob* in the *feed\_dict* argument, which controls the probability of the *dropou*t layer that we discussed earlier.

.

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

train\_step = tf.train.AdamOptimizer(1e-4).minimize(cross\_entropy)

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

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

sess = tf.Session()

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

for i in range(20000):

batch = mnist.train.next\_batch(50)

if i%100 == 0:

train\_accuracy = sess.run( accuracy, feed\_dict={x:batch[0], y\_: batch[1], keep\_prob: 1.0})

print("step %d, training accuracy %g"%(i, train\_accuracy))

sess.run(train\_step,feed\_dict={x: batch[0], y\_: batch[1], keep\_prob: 0.5})

print("test accuracy %g"% sess.run(accuracy, feed\_dict={ x: mnist.test.images, y\_: mnist.test.labels, keep\_prob: 1.0}))

Like in the previous models, the entire code can be found on the *Github* page of this book, one can verify that this model achieves 99.2% accuracy.

Here is when the brief introduction to building, training and evaluating *deep neural networks* using TensorFlow comes to an end. If the reader have managed to run the provided code, he or she has noticed that the training of this network took noticeably longer time than the one in the previous chapters; you can imagine that a network with much more layers will take a lot longer to train. I suggest you to read the next chapter, where is explained how to use a GPU for training, which will vaslty decrease your training time.

The code of this chapter can be found in *CNN.py* on the *github* page of this book [[43]](http://www.jorditorres.org/first-contact-with-tensorflow/" \l "_ftn43), for studying purposes the code can be found in its entirity below:

import input\_data

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

import tensorflow as tf

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

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

x\_image = tf.reshape(x, [-1,28,28,1])

print "x\_image="

print x\_image

def weight\_variable(shape):

initial = tf.truncated\_normal(shape, stddev=0.1)

return tf.Variable(initial)

def bias\_variable(shape):

initial = tf.constant(0.1, shape=shape)

return tf.Variable(initial)

def conv2d(x, W):

return tf.nn.conv2d(x, W, strides=[1, 1, 1, 1], padding='SAME')

def max\_pool\_2x2(x):

return tf.nn.max\_pool(x, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='SAME')

W\_conv1 = weight\_variable([5, 5, 1, 32])

b\_conv1 = bias\_variable([32])

h\_conv1 = tf.nn.relu(conv2d(x\_image, W\_conv1) + b\_conv1)

h\_pool1 = max\_pool\_2x2(h\_conv1)

W\_conv2 = weight\_variable([5, 5, 32, 64])

b\_conv2 = bias\_variable([64])

h\_conv2 = tf.nn.relu(conv2d(h\_pool1, W\_conv2) + b\_conv2)

h\_pool2 = max\_pool\_2x2(h\_conv2)

W\_fc1 = weight\_variable([7 \* 7 \* 64, 1024])

b\_fc1 = bias\_variable([1024])

h\_pool2\_flat = tf.reshape(h\_pool2, [-1, 7\*7\*64])

h\_fc1 = tf.nn.relu(tf.matmul(h\_pool2\_flat, W\_fc1) + b\_fc1)

keep\_prob = tf.placeholder("float")

h\_fc1\_drop = tf.nn.dropout(h\_fc1, keep\_prob)

W\_fc2 = weight\_variable([1024, 10])

b\_fc2 = bias\_variable([10])

y\_conv=tf.nn.softmax(tf.matmul(h\_fc1\_drop, W\_fc2) + b\_fc2)

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

train\_step = tf.train.AdamOptimizer(1e-4).minimize(cross\_entropy)

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

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

sess = tf.Session()

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

for i in range(200):

batch = mnist.train.next\_batch(50)

if i%10 == 0:

train\_accuracy = sess.run( accuracy, feed\_dict={ x:batch[0], y\_: batch[1], keep\_prob: 1.0})

print("step %d, training accuracy %g"%(i, train\_accuracy))

sess.run(train\_step,feed\_dict={x: batch[0], y\_: batch[1], keep\_prob: 0.5})

print("test accuracy %g"% sess.run(accuracy, feed\_dict={

x: mnist.test.images, y\_: mnist.test.labels, keep\_prob: 1.0}))

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