(https://www.bigdatauniversity.com)

CONVOLUTIONAL NEURAL NETWORK APPLICATION

Introduction

In this section, we will use the famous MNIST Dataset (http://yann.lecun.com/exdb/mnist/) to build two Neural Networks capable to perform handwritten digits classification. The first Network is a simple Multi-layer Perceptron (MLP) and the second one is a Convolutional Neural Network (CNN from now on). In other words, when given an input our algorithm will say, with some associated error, what type of digit this input represents.

Click on the links to go to the following sections:

Table of Contents

What is Deep Learning?

Brief Theory: Deep learning (also known as deep structured learning, hierarchical learning or deep machine learning) is a branch of machine learning based on a set of algorithms that attempt to model high-level abstractions in data by using multiple processing layers, with complex structures or otherwise, composed of multiple non-linear transformations.
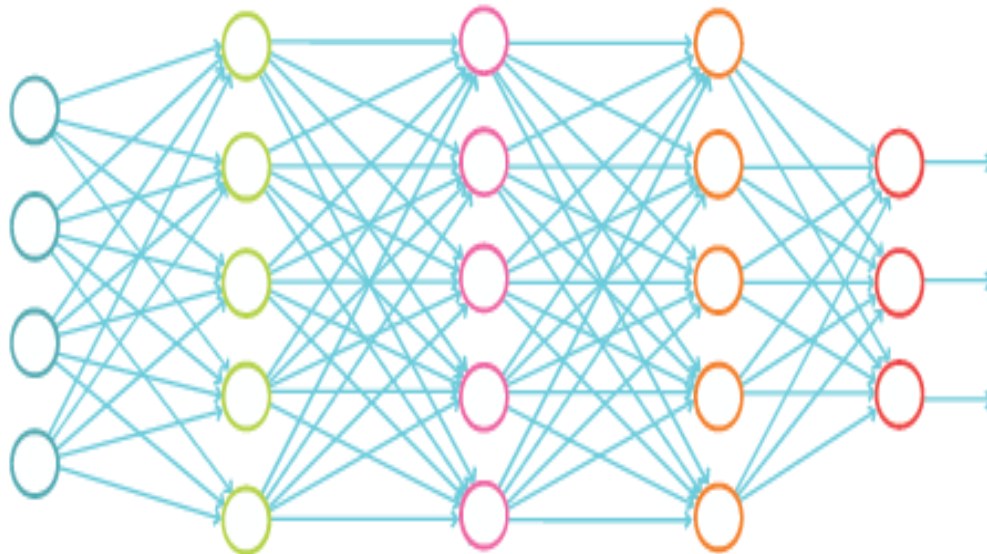
It's time for deep learning. Our brain doesn't work with only one or three layers. Why it would be different with machines?.

In Practice, defining the term "Deep": in this context, deep means that we are studying a Neural Network which has several hidden layers (more than one), no matter what type (convolutional, pooling, normalization, fully-connected etc). The most interesting part is that some papers noticed that Deep Neural Networks with the right architectures/hyper-parameters achieve better results than shallow Neural Networks with the same computational power (e.g. number of neurons or connections).

In Practice, defining "Learning": In the context of supervised learning, digits recognition in our case, the learning part consists of a target/feature which is to be predicted using a given set of observations with the already known final prediction (label). In our case, the target will be the digit (0, 1, 2, 3, 4, 5, 6, 7, 8, 9) and the observations are the intensity and relative position of the pixels. After some training, it is possible to generate a "function" that map inputs (digit image) to desired outputs(type of digit). The only problem is how well this map operation occurs. While trying to generate this "function", the training process continues until the model achieves a desired level of accuracy on the training data.

Notice: This notebook has been created with TensorFlow version 1.8, and might not work with other versions.

```
import tensorflow as tf
tf.__version__
```

```
/home/jupyterlab/conda/envs/python/lib/python3.6/site-packages/tensorfl
ow/python/framework/dtypes.py:519: FutureWarning: Passing (type, 1) or
'1type' as a synonym of type is deprecated; in a future version of nump
y, it will be understood as (type, (1,)) / '(1,)type'.
  _np_qint8 = np.dtype([("qint8", np.int8, 1)])
/home/jupyterlab/conda/envs/python/lib/python3.6/site-packages/tensorfl
ow/python/framework/dtypes.py:520: FutureWarning: Passing (type, 1) or
'1type' as a synonym of type is deprecated; in a future version of nump
y, it will be understood as (type, (1,)) / '(1,)type'.
  _np_quint8 = np.dtype([("quint8", np.uint8, 1)])
/home/jupyterlab/conda/envs/python/lib/python3.6/site-packages/tensorfl
ow/python/framework/dtypes.py:521: FutureWarning: Passing (type, 1) or
'1type' as a synonym of type is deprecated; in a future version of nump
y, it will be understood as (type, (1,)) / '(1,)type'.
  _np_qint16 = np.dtype([("qint16", np.int16, 1)])
/home/jupyterlab/conda/envs/python/lib/python3.6/site-packages/tensorfl
ow/python/framework/dtypes.py:522: FutureWarning: Passing (type, 1) or
'1type' as a synonym of type is deprecated; in a future version of nump
y, it will be understood as (type, (1,)) / '(1,)type'.
  _np_quint16 = np.dtype([("quint16", np.uint16, 1)])
/home/jupyterlab/conda/envs/python/lib/python3.6/site-packages/tensorfl
ow/python/framework/dtypes.py:523: FutureWarning: Passing (type, 1) or
'1type' as a synonym of type is deprecated; in a future version of nump
y, it will be understood as (type, (1,)) / '(1,)type'.
  _np_qint32 = np.dtype([("qint32", np.int32, 1)])
/home/jupyterlab/conda/envs/python/lib/python3.6/site-packages/tensorfl
ow/python/framework/dtypes.py:528: FutureWarning: Passing (type, 1) or
'1type' as a synonym of type is deprecated; in a future version of nump
y, it will be understood as (type, (1,)) / '(1,)type'.
  np_resource = np.dtype([("resource", np.ubyte, 1)])
```

Out[1]:

```
'1.8.0'
```

In this tutorial, we first classify MNIST using a simple Multi-layer perceptron and then, in the second part, we use deeplearning to improve the accuracy of our results.

1st part: classify MNIST using a simple model.

We are going to create a simple Multi-layer perceptron, a simple type of Neural Network, to perform classification tasks on the MNIST digits dataset. If you are not familiar with the MNIST dataset, please consider to read more about it: click here (http://yann.lecun.com/exdb/mnist/)

What is MNIST?

According to LeCun's website, the MNIST is a: "database of handwritten digits that has a training set of 60,000 examples, and a test set of 10,000 examples. It is a subset of a larger set available from NIST. The digits have been size-normalized and centered in a fixed-size image".

Import the MNIST dataset using TensorFlow built-in feature

It's very important to notice that MNIST is a high optimized data-set and it does not contain images. You will need to build your own code if you want to see the real digits. Another important side note is the effort that the authors invested on this data-set with normalization and centering operations.

```python
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)
```

```
WARNING:tensorflow:From <ipython-input-2-8bf8ae5a5303>:2: read_data_set
s (from tensorflow.contrib.learn.python.learn.datasets.mnist) is deprec
ated and will be removed in a future version.
Instructions for updating:
Please use alternatives such as official/mnist/dataset.py from tensorfl
ow/models.
WARNING:tensorflow:From /home/jupyterlab/conda/envs/python/lib/python3.
6/site-packages/tensorflow/contrib/learn/python/learn/datasets/mnist.p
y:260: maybe_download (from tensorflow.contrib.learn.python.learn.datas
ets.base) is deprecated and will be removed in a future version.
Instructions for updating:
Please write your own downloading logic.
WARNING:tensorflow:From /home/jupyterlab/conda/envs/python/lib/python3.
6/site-packages/tensorflow/contrib/learn/python/learn/datasets/base.py:
252: _internal_retry.<locals>.wrap.<locals>.wrapped_fn (from tensorflo
w.contrib.learn.python.learn.datasets.base) is deprecated and will be r
emoved in a future version.
Instructions for updating:
Please use urllib or similar directly.
Successfully downloaded train-images-idx3-ubyte.gz 9912422 bytes.
WARNING:tensorflow:From /home/jupyterlab/conda/envs/python/lib/python3.
6/site-packages/tensorflow/contrib/learn/python/learn/datasets/mnist.p
y:262: extract_images (from tensorflow.contrib.learn.python.learn.datas
ets.mnist) is deprecated and will be removed in a future version.
Instructions for updating:
Please use tf.data to implement this functionality.
Extracting MNIST_data/train-images-idx3-ubyte.gz
Successfully downloaded train-labels-idx1-ubyte.gz 28881 bytes.
WARNING:tensorflow:From /home/jupyterlab/conda/envs/python/lib/python3.
6/site-packages/tensorflow/contrib/learn/python/learn/datasets/mnist.p
y:267: extract_labels (from tensorflow.contrib.learn.python.learn.datas
ets.mnist) is deprecated and will be removed in a future version.
Instructions for updating:
Please use tf.data to implement this functionality.
Extracting MNIST_data/train-labels-idx1-ubyte.gz
WARNING:tensorflow:From /home/jupyterlab/conda/envs/python/lib/python3.
6/site-packages/tensorflow/contrib/learn/python/learn/datasets/mnist.p
y:110: dense_to_one_hot (from tensorflow.contrib.learn.python.learn.dat
asets.mnist) is deprecated and will be removed in a future version.
Instructions for updating:
Please use tf.one_hot on tensors.
Successfully downloaded t10k-images-idx3-ubyte.gz 1648877 bytes.
Extracting MNIST_data/t10k-images-idx3-ubyte.gz
Successfully downloaded t10k-labels-idx1-ubyte.gz 4542 bytes.
Extracting MNIST_data/t10k-labels-idx1-ubyte.gz
WARNING:tensorflow:From /home/jupyterlab/conda/envs/python/lib/python3.
6/site-packages/tensorflow/contrib/learn/python/learn/datasets/mnist.p
y:290: DataSet.__init__ (from tensorflow.contrib.learn.python.learn.dat
asets.mnist) is deprecated and will be removed in a future version.
Instructions for updating:
Please use alternatives such as official/mnist/dataset.py from tensorfl
ow/models.
```

The one-hot = True argument only means that, in contrast to Binary representation, the labels will be presented in a way that to represent a number N, the Nth bit is 1 while the the other bits are 0. For example, five and zero in a binary code would be:

```
Number representation:    0
Binary encoding:       [2^5]  [2^4]   [2^3]   [2^2]   [2^1]   [2^0]
Array/vector:            0      0       0       0       0       0


Number representation:    5
Binary encoding:       [2^5]  [2^4]   [2^3]   [2^2]   [2^1]   [2^0]
Array/vector:            0      0       0       1       0       1
```

Using a different notation, the same digits using one-hot vector representation can be show as:

```
Number representation:     0
One-hot encoding:       [5]    [4]     [3]     [2]     [1]     [0]
Array/vector:            0      0       0       0       0       1


Number representation:     5
One-hot encoding:       [5]    [4]     [3]     [2]     [1]     [0]
Array/vector:            1      0       0       0       0       0
```

Understanding the imported data

The imported data can be divided as follow:

- Training (mnist.train) >> Use the given dataset with inputs and related outputs for training of NN. In our case, if you give an image that you know that represents a "nine", this set will tell the neural network that we expect a "nine" as the output.

    - 55,000 data points
    - mnist.train.images for inputs
    - mnist.train.labels for outputs

- Validation (mnist.validation) >> The same as training, but now the data is used to generate model properties (classification error, for example) and from this, tune parameters like the optimal number of hidden units or determine a stopping point for the back-propagation algorithm

    - 5,000 data points
    - mnist.validation.images for inputs
    - mnist.validation.labels for outputs

- Test (mnist.test) >> the model does not have access to this informations prior to the testing phase. It is used to evaluate the performance and accuracy of the model against "real life situations". No further optimization beyond this point.

    - 10,000 data points
    - mnist.test.images for inputs
    - mnist.test.labels for outputs

Creating an interactive section

You have two basic options when using TensorFlow to run your code:

- [Build graphs and run session] Do all the set-up and THEN execute a session to evaluate tensors and run operations (ops)
- [Interactive session] create your coding and run on the fly.

For this first part, we will use the interactive session that is more suitable for environments like Jupyter notebooks.

In [3]:

```
sess = tf.InteractiveSession()
```

Creating placeholders

It iss a best practice to create placeholders before variable assignments when using TensorFlow. Here we'll create placeholders for inputs ("Xs") and outputs ("Ys").

Placeholder 'X': represents the "space" allocated input or the images.

- Each input has 784 pixels distributed by a 28 width x 28 height matrix
- The 'shape' argument defines the tensor size by its dimensions.
- 1st dimension = None. Indicates that the batch size, can be of any size.
- 2nd dimension = 784. Indicates the number of pixels on a single flattened MNIST image.

Placeholder 'Y': represents the final output or the labels.

- 10 possible classes (0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
- The 'shape' argument defines the tensor size by its dimensions.
- 1st dimension = None. Indicates that the batch size, can be of any size.
- 2nd dimension = 10. Indicates the number of targets/outcomes

dtype for both placeholders: if you not sure, use tf.float32. The limitation here is that the later presented softmax function only accepts float32 or float64 dtypes. For more dtypes, check TensorFlow's documentation here (https://www.tensorflow.org/api_docs/python/tf/DType)

In [4]:

```
x  = tf.placeholder(tf.float32, shape=[None, 784])
y_ = tf.placeholder(tf.float32, shape=[None, 10])
```

Assigning bias and weights to null tensors

Now we are going to create the weights and biases, for this purpose they will be used as arrays filled with zeros. The values that we choose here can be critical, but we'll cover a better way on the second part, instead of this type of initialization.

In [5]:

```
# Weight tensor
W = tf.Variable(tf.zeros([784, 10],tf.float32))
# Bias tensor
b = tf.Variable(tf.zeros([10],tf.float32))
```

Execute the assignment operation

Before, we assigned the weights and biases but we did not initialize them with null values. For this reason, TensorFlow need to initialize the variables that you assign.
Please notice that we're using this notation "sess.run" because we previously started an interactive session.

```
# run the op initialize_all_variables using an interactive session
sess.run(tf.global_variables_initializer())
```

Adding Weights and Biases to input

The only difference for our next operation to the picture below is that we are using the mathematical convention for what is being executed in the illustration. The tf.matmul operation performs a matrix multiplication between x (inputs) and W (weights) and after the code add biases.
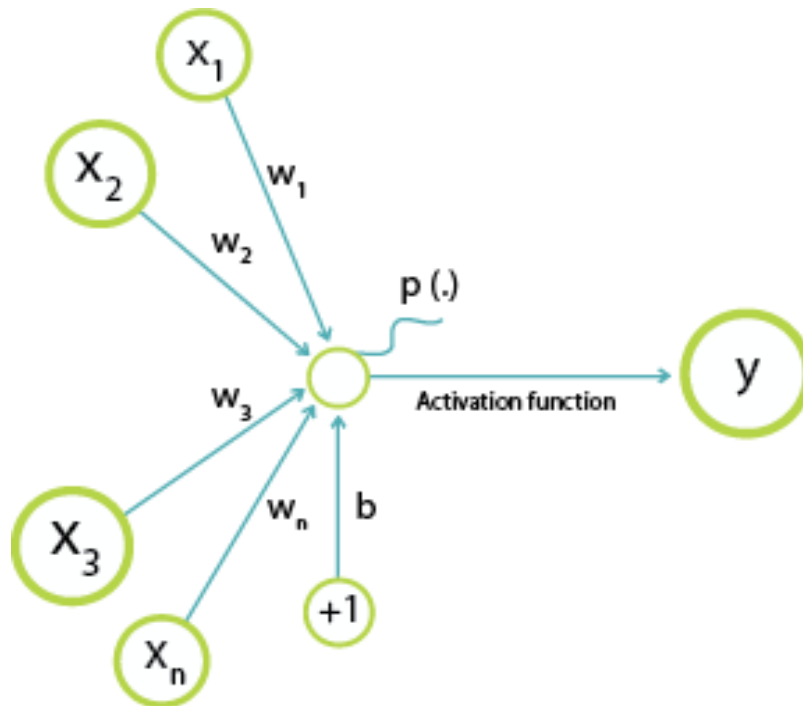

Illustration showing how weights and biases are added to neurons/nodes.

In [7]:

```
# mathematical operation to add weights and biases to the inputs
tf.matmul(x,W) + b
```

Out[7]:

```
<tf.Tensor 'add:0' shape=(?, 10) dtype=float32>
```

Softmax Regression

Softmax is an activation function that is normally used in classification problems. It generate the probabilities for the output. For example, our model will not be 100% sure that one digit is the number nine, instead, the answer will be a distribution of probabilities where, if the model is right, the nine number will have a larger probability than the other other digits.

For comparison, below is the one-hot vector for a nine digit label:

0 --> 0 1 --> 0 2 --> 0 3 --> 0 4 --> 0 5 --> 0 6 --> 0 7 --> 0 8 --> 0 9 --> 1

A machine does not have all this certainty, so we want to know what is the best guess, but we also want to understand how sure it was and what was the second better option. Below is an example of a hypothetical distribution for a nine digit:

0 -->0.01 1 -->0.02 2 -->0.03 3 -->0.02 4 -->0.12 5 -->0.01 6 -->0.03 7 -->0.06 8 -->0.1 9 -->0.6

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

Logistic function output is used for the classification between two target classes 0/1. Softmax function is generalized type of logistic function. That is, Softmax can output a multiclass categorical probability distribution.

Cost function

It is a function that is used to minimize the difference between the right answers (labels) and estimated outputs by our Network.

```python
cross_entropy = tf.reduce_mean(-tf.reduce_sum(y_ * tf.log(y), reduction_indices=[1]))
```

Type of optimization: Gradient Descent

This is the part where you configure the optimizer for your Neural Network. There are several optimizers available, in our case we will use Gradient Descent because it is a well established optimizer.

```python
train_step = tf.train.GradientDescentOptimizer(0.5).minimize(cross_entropy)
```

Training batches

Train using minibatch Gradient Descent.

In practice, Batch Gradient Descent is not often used because is too computationally expensive. The good part about this method is that you have the true gradient, but with the expensive computing task of using the whole dataset in one time. Due to this problem, Neural Networks usually use minibatch to train.

```python
#Load 50 training examples for each training iteration
for i in range(1000):
    batch = mnist.train.next_batch(50)
    train_step.run(feed_dict={x: batch[0], y_: batch[1]})
```

Test

```
correct_prediction = tf.equal(tf.argmax(y, 1), tf.argmax(y_, 1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
acc = accuracy.eval(feed_dict={x: mnist.test.images, y_: mnist.test.labels}) * 100
print("The final accuracy for the simple ANN model is: {} % ".format(acc) )
```

The final accuracy for the simple ANN model is: 91.86999797821045 %

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

Evaluating the final result

Is the final result good?

Let's check the best algorithm available out there (10th June 2016):

Result: 0.21% error (99.79% accuracy)
Reference here (http://cs.nyu.edu/~wanli/dropc/)

How to improve our model?

Several options as follow:

- Regularization of Neural Networks using DropConnect
- Multi-column Deep Neural Networks for Image Classification
- APAC: Augmented Pattern Classification with Neural Networks
- Simple Deep Neural Network with Dropout

In the next part we are going to explore the option:

- Simple Deep Neural Network with Dropout (more than 1 hidden layer)

2nd part: Deep Learning applied on MNIST

In the first part, we learned how to use a simple ANN to classify MNIST. Now we are going to expand our knowledge using a Deep Neural Network.

Architecture of our network is:

- (Input) -> [batch_size, 28, 28, 1] >> Apply 32 filter of [5x5]
- (Convolutional layer 1) -> [batch_size, 28, 28, 32]
- (ReLU 1) -> [?, 28, 28, 32]
- (Max pooling 1) -> [?, 14, 14, 32]
- (Convolutional layer 2) -> [?, 14, 14, 64]
- (ReLU 2) -> [?, 14, 14, 64]
- (Max pooling 2) -> [?, 7, 7, 64]
- [fully connected layer 3] -> [1x1024]
- [ReLU 3] -> [1x1024]
- [Drop out] -> [1x1024]
- [fully connected layer 4] -> [1x10]

The next cells will explore this new architecture.

Starting the code

In [14]:

```python
import tensorflow as tf

# finish possible remaining session
sess.close()

#Start interactive session
sess = tf.InteractiveSession()
```

The MNIST data

In [15]:

```python
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets('MNIST_data', one_hot=True)
```

Extracting MNIST_data/train-images-idx3-ubyte.gz
Extracting MNIST_data/train-labels-idx1-ubyte.gz
Extracting MNIST_data/t10k-images-idx3-ubyte.gz
Extracting MNIST_data/t10k-labels-idx1-ubyte.gz

Initial parameters

Create general parameters for the model

```
width = 28 # width of the image in pixels
height = 28 # height of the image in pixels
flat = width * height # number of pixels in one image
class_output = 10 # number of possible classifications for the problem
```

Input and output

Create place holders for inputs and outputs

```
x  = tf.placeholder(tf.float32, shape=[None, flat])
y_ = tf.placeholder(tf.float32, shape=[None, class_output])
```

Converting images of the data set to tensors

The input image is 28 pixels by 28 pixels, 1 channel (grayscale). In this case, the first dimension is the **batch number** of the image, and can be of any size (so we set it to -1). The second and third dimensions are width and height, and the last one is the image channels.

```
x_image = tf.reshape(x, [-1,28,28,1])
x_image
```

```
<tf.Tensor 'Reshape:0' shape=(?, 28, 28, 1) dtype=float32>
```
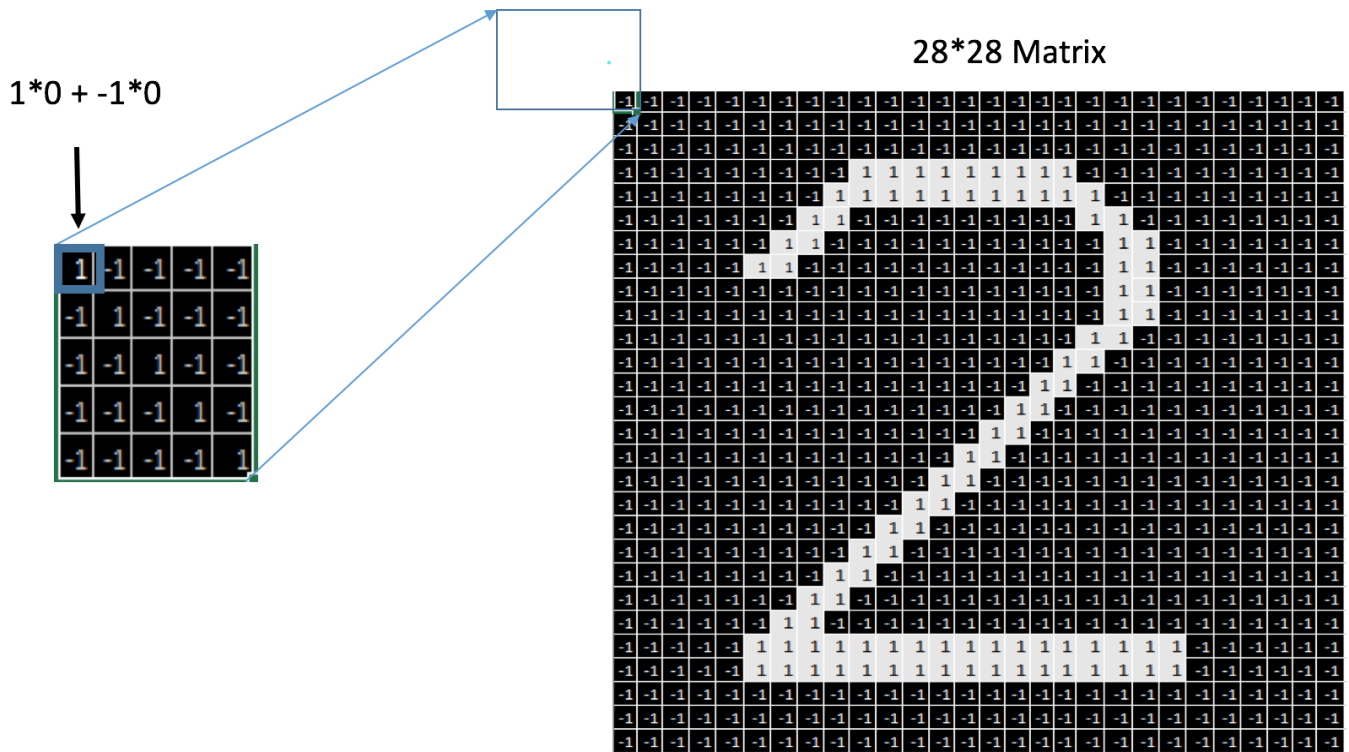
Convolutional Layer 1

Defining kernel weight and bias
We define a kernel here. The Size of the filter/kernel is 5x5; Input channels is 1 (grayscale); and we need 32 different feature maps (here, 32 feature maps means 32 different filters are applied on each image. So, the output of convolution layer would be 28x28x32). In this step, we create a filter / kernel tensor of shape `[filter_height, filter_width, in_channels, out_channels]`

```
W_conv1 = tf.Variable(tf.truncated_normal([5, 5, 1, 32], stddev=0.1))
b_conv1 = tf.Variable(tf.constant(0.1, shape=[32])) # need 32 biases for 32 outputs
```

</h4>Convolve with weight tensor and add biases.</h4>

To create convolutional layer, we use **tf.nn.conv2d**. It computes a 2-D convolution given 4-D input and filter tensors.

Inputs:

- tensor of shape [batch, in_height, in_width, in_channels]. x of shape [batch_size,28 ,28, 1]
- a filter / kernel tensor of shape [filter_height, filter_width, in_channels, out_channels]. W is of size [5, 5, 1, 32]
- stride which is [1, 1, 1, 1]. The convolutional layer, slides the "kernel window" across the input tensor. As the input tensor has 4 dimensions: [batch, height, width, channels], then the convolution operates on a 2D window on the height and width dimensions. **strides** determines how much the window shifts by in each of the dimensions. As the first and last dimensions are related to batch and channels, we set the stride to 1. But for second and third dimension, we could set other values, e.g. [1, 2, 2, 1]

Process:

- Change the filter to a 2-D matrix with shape [5*5*1,32]
- Extracts image patches from the input tensor to form a *virtual* tensor of shape `[batch, 28, 28, 5*5*1]`.
- For each batch, right-multiplies the filter matrix and the image vector.

Output:

- A `Tensor` (a 2-D convolution) of size tf.Tensor 'add_7:0' shape=(?, 28, 28, 32)- Notice: the output of the first convolution layer is 32 [28x28] images. Here 32 is considered as volume/depth of the output image.

```
convolve1= tf.nn.conv2d(x_image, W_conv1, strides=[1, 1, 1, 1], padding='SAME') + b
_conv1
```
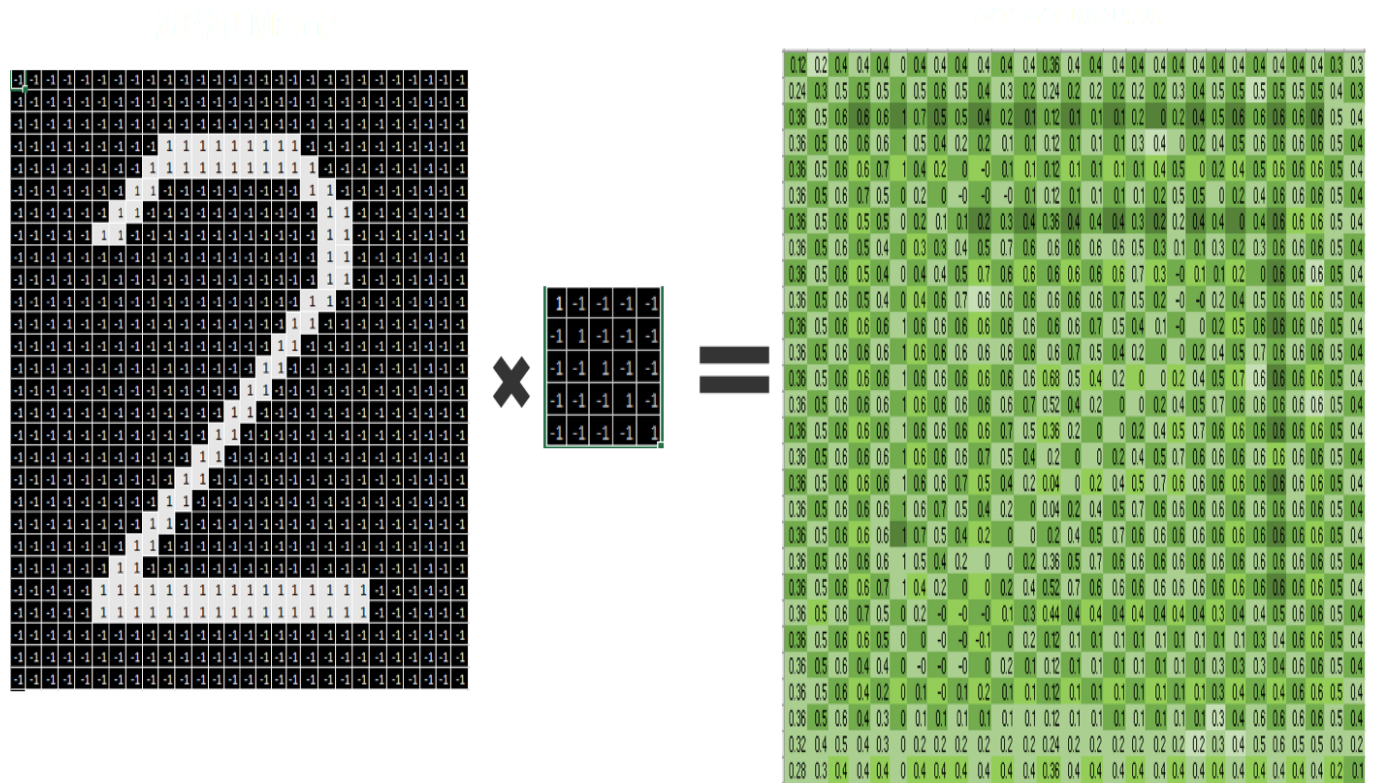

Apply the ReLU activation Function

In this step, we just go through all outputs convolution layer, **convolve1**, and wherever a negative number occurs, we swap it out for a 0. It is called ReLU activation Function.
Let f(x) is a ReLU activation function $f(x) = max(0,x)$.

In [21]:

```
h_conv1 = tf.nn.relu(convolve1)
```
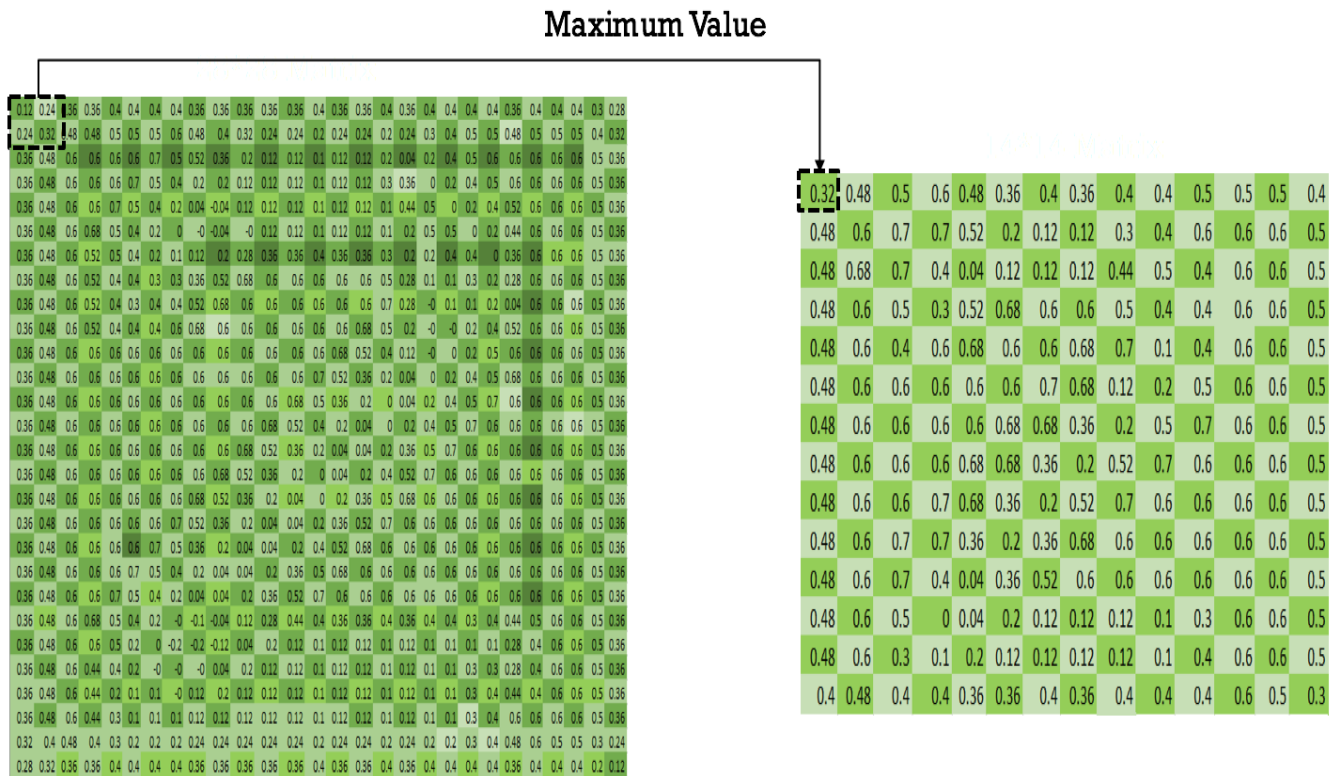
Apply the max pooling

max pooling is a form of non-linear down-sampling. It partitions the input image into a set of rectangles and, and then find the maximum value for that region.

Lets use **tf.nn.max_pool** function to perform max pooling. **Kernel size:** 2x2 (if the window is a 2x2 matrix, it would result in one output pixel)
Strides: dictates the sliding behaviour of the kernel. In this case it will move 2 pixels everytime, thus not overlapping. The input is a matrix of size 28x28x32, and the output would be a matrix of size 14x14x32.

In [22]:

```
conv1 = tf.nn.max_pool(h_conv1, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='SAME') #max_pool_2x2
conv1
```

Out[22]:

```
<tf.Tensor 'MaxPool:0' shape=(?, 14, 14, 32) dtype=float32>
```

First layer completed

Convolutional Layer 2

Weights and Biases of kernels

We apply the convolution again in this layer. Lets look at the second layer kernel:

- Filter/kernel: 5x5 (25 pixels)
- Input channels: 32 (from the 1st Conv layer, we had 32 feature maps)
- 64 output feature maps

Notice: here, the input image is [14x14x32], the filter is [5x5x32], we use 64 filters of size [5x5x32], and the output of the convolutional layer would be 64 convolved image, [14x14x64].

Notice: the convolution result of applying a filter of size [5x5x32] on image of size [14x14x32] is an image of size [14x14x1], that is, the convolution is functioning on volume.

In [23]:

```
W_conv2 = tf.Variable(tf.truncated_normal([5, 5, 32, 64], stddev=0.1))
b_conv2 = tf.Variable(tf.constant(0.1, shape=[64])) #need 64 biases for 64 outputs
```

Convolve image with weight tensor and add biases.

In [24]:

```
convolve2= tf.nn.conv2d(conv1, W_conv2, strides=[1, 1, 1, 1], padding='SAME') + b_c
onv2
```

Apply the ReLU activation Function

In [25]:

```
h_conv2 = tf.nn.relu(convolve2)
```

Apply the max pooling

In [26]:

```
conv2 = tf.nn.max_pool(h_conv2, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding=
'SAME') #max_pool_2x2
conv2
```

Out[26]:

```
<tf.Tensor 'MaxPool_1:0' shape=(?, 7, 7, 64) dtype=float32>
```

Second layer completed. So, what is the output of the second layer, layer2?

- it is 64 matrix of [7x7]

Fully Connected Layer

You need a fully connected layer to use the Softmax and create the probabilities in the end. Fully connected layers take the high-level filtered images from previous layer, that is all 64 matrices, and convert them to a flat array.

So, each matrix [7x7] will be converted to a matrix of [49x1], and then all of the 64 matrix will be connected, which make an array of size [3136x1]. We will connect it into another layer of size [1024x1]. So, the weight between these 2 layers will be [3136x1024]
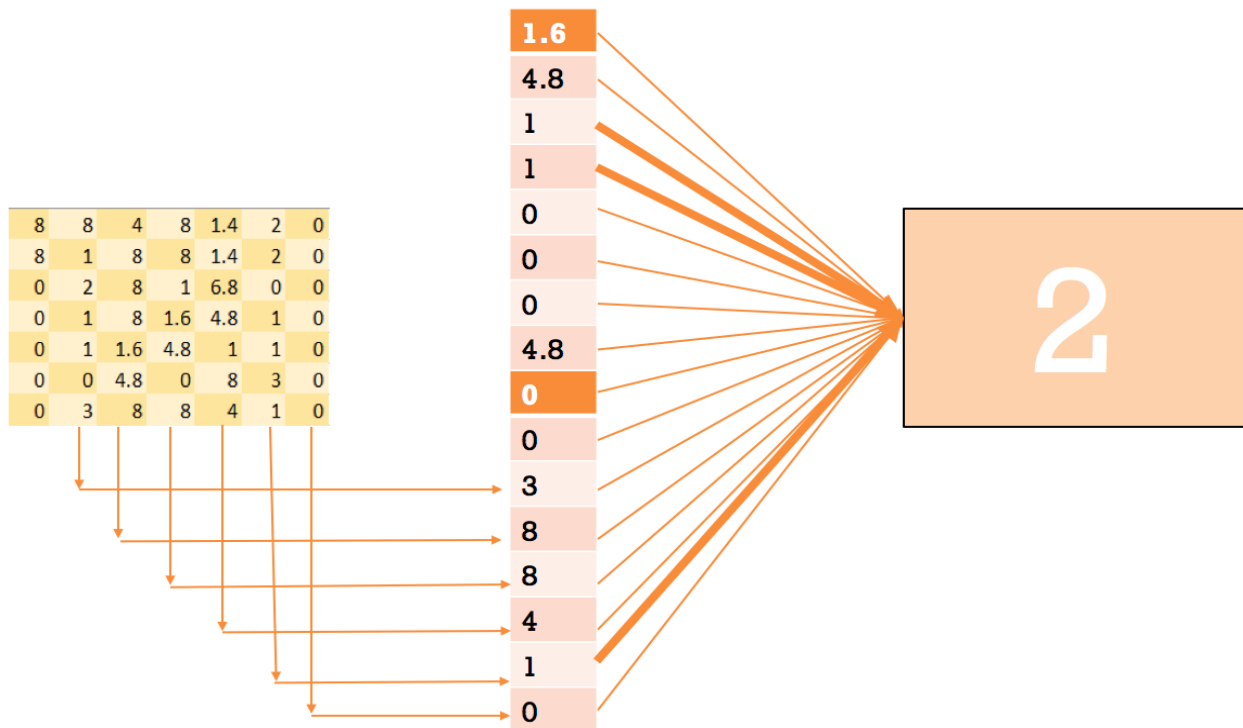
Flattening Second Layer

In [27]:

```
layer2_matrix = tf.reshape(conv2, [-1, 7 * 7 * 64])
```

Weights and Biases between layer 2 and 3

Composition of the feature map from the last layer (7x7) multiplied by the number of feature maps (64); 1027 outputs to Softmax layer

In [28]:

```
W_fc1 = tf.Variable(tf.truncated_normal([7 * 7 * 64, 1024], stddev=0.1))
b_fc1 = tf.Variable(tf.constant(0.1, shape=[1024])) # need 1024 biases for 1024 out
puts
```

Matrix Multiplication (applying weights and biases)

```
fcl = tf.matmul(layer2_matrix, W_fc1) + b_fc1
```

Apply the ReLU activation Function

```
h_fc1 = tf.nn.relu(fcl)
h_fc1
```

```
<tf.Tensor 'Relu_2:0' shape=(?, 1024) dtype=float32>
```

Third layer completed

Dropout Layer, Optional phase for reducing overfitting

It is a phase where the network "forget" some features. At each training step in a mini-batch, some units get switched off randomly so that it will not interact with the network. That is, it weights cannot be updated, nor affect the learning of the other network nodes. This can be very useful for very large neural networks to prevent overfitting.

```
keep_prob = tf.placeholder(tf.float32)
layer_drop = tf.nn.dropout(h_fc1, keep_prob)
layer_drop
```

```
<tf.Tensor 'dropout/mul:0' shape=(?, 1024) dtype=float32>
```

Readout Layer (Softmax Layer)

Type: Softmax, Fully Connected Layer.

Weights and Biases

In last layer, CNN takes the high-level filtered images and translate them into votes using softmax. Input channels: 1024 (neurons from the 3rd Layer); 10 output features

```
W_fc2 = tf.Variable(tf.truncated_normal([1024, 10], stddev=0.1)) #1024 neurons
b_fc2 = tf.Variable(tf.constant(0.1, shape=[10])) # 10 possibilities for digits [0,
1,2,3,4,5,6,7,8,9]
```

Matrix Multiplication (applying weights and biases)

```
fc=tf.matmul(layer_drop, W_fc2) + b_fc2
```

Apply the Softmax activation Function
softmax allows us to interpret the outputs of **fcl4** as probabilities. So, **y_conv** is a tensor of probabilities.

```
y_CNN= tf.nn.softmax(fc)
y_CNN
```

```
<tf.Tensor 'Softmax_1:0' shape=(?, 10) dtype=float32>
```

Summary of the Deep Convolutional Neural Network

Now is time to remember the structure of our network

0) Input - MNIST dataset

1) Convolutional and Max-Pooling

2) Convolutional and Max-Pooling

3) Fully Connected Layer

4) Processing - Dropout

5) Readout layer - Fully Connected

6) Outputs - Classified digits

Define functions and train the model

Define the loss function

We need to compare our output, layer4 tensor, with ground truth for all mini_batch. we can use **cross entropy>/b> to see how bad our CNN is working - to measure the error at a softmax layer.
The following code shows an toy sample of cross-entropy for a mini-batch of size 2 which its items have been classified. You can run it (first change the cell type to code in the toolbar) to see how cross entropy changes.**

import numpy as np layer4_test =[[0.9, 0.1, 0.1],[0.9, 0.1, 0.1]] y_test=[[1.0, 0.0, 0.0],[1.0, 0.0, 0.0]] np.mean(- np.sum(y_test * np.log(layer4_test),1))

reduce_sum computes the sum of elements of (y_ * tf.log(layer4) across second dimension of the tensor, and reduce_mean computes the mean of all elements in the tensor..

In [35]:

```
cross_entropy = tf.reduce_mean(-tf.reduce_sum(y_ * tf.log(y_CNN), reduction_indices =[1]))
```

Define the optimizer

It is obvious that we want minimize the error of our network which is calculated by cross_entropy metric. To solve the problem, we have to compute gradients for the loss (which is minimizing the cross-entropy) and apply gradients to variables. It will be done by an optimizer: GradientDescent or Adagrad.

In [36]:

```
train_step = tf.train.AdamOptimizer(1e-4).minimize(cross_entropy)
```

Define prediction
Do you want to know how many of the cases in a mini-batch has been classified correctly? lets count them.

In [37]:

```
correct_prediction = tf.equal(tf.argmax(y_CNN, 1), tf.argmax(y_, 1))
```

Define accuracy
It makes more sense to report accuracy using average of correct cases.

In [38]:

```
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
```

Run session, train

In [39]:

```
sess.run(tf.global_variables_initializer())
```

If you want a fast result (it might take sometime to train it)

In [40]:

```
for i in range(1100):
    batch = mnist.train.next_batch(50)
    if i%100 == 0:
        train_accuracy = accuracy.eval(feed_dict={x:batch[0], y_: batch[1], keep_pr
ob: 1.0})
        print("step %d, training accuracy %g"%(i, float(train_accuracy)))
    train_step.run(feed_dict={x: batch[0], y_: batch[1], keep_prob: 0.5})
```

```
step 0, training accuracy 0.12
step 100, training accuracy 0.94
step 200, training accuracy 0.92
step 300, training accuracy 0.88
step 400, training accuracy 0.86
step 500, training accuracy 0.94
step 600, training accuracy 0.98
step 700, training accuracy 1
step 800, training accuracy 0.98
step 900, training accuracy 0.94
step 1000, training accuracy 0.94
```

for i in range(20000): batch = mnist.train.next_batch(50) if i%100 == 0: train_accuracy = accuracy.eval(feed_dict={ x:batch[0], y_: batch[1], keep_prob: 1.0}) print("step %d, training accuracy %g"% (i, train_accuracy)) train_step.run(feed_dict={x: batch[0], y_: batch[1], keep_prob: 0.5})

PS. If you have problems running this notebook, please shutdown all your Jupyter runnning notebooks, clear all cells outputs and run each cell only after the completion of the previous cell.

Evaluate the model

Print the evaluation to the user

In [41]:

test accuracy 0.9673000007867814

## Visualization

Do you want to look at all the filters?

In [42]:

In [43]:

--2020-06-07 23:50:29-- http://deeplearning.net/tutorial/code/utils.py
Resolving deeplearning.net (deeplearning.net)... 132.204.26.28
Connecting to deeplearning.net (deeplearning.net)|132.204.26.28|:80...
connected.
HTTP request sent, awaiting response... 200 OK
Length: 5097 (5.0K) [text/plain]
Saving to: 'utils1.py'

utils1.py   100%[===================>]   4.98K --.-KB/s   in
0s

2020-06-07 23:50:29 (334 MB/s) - 'utils1.py' saved [5097/5097]
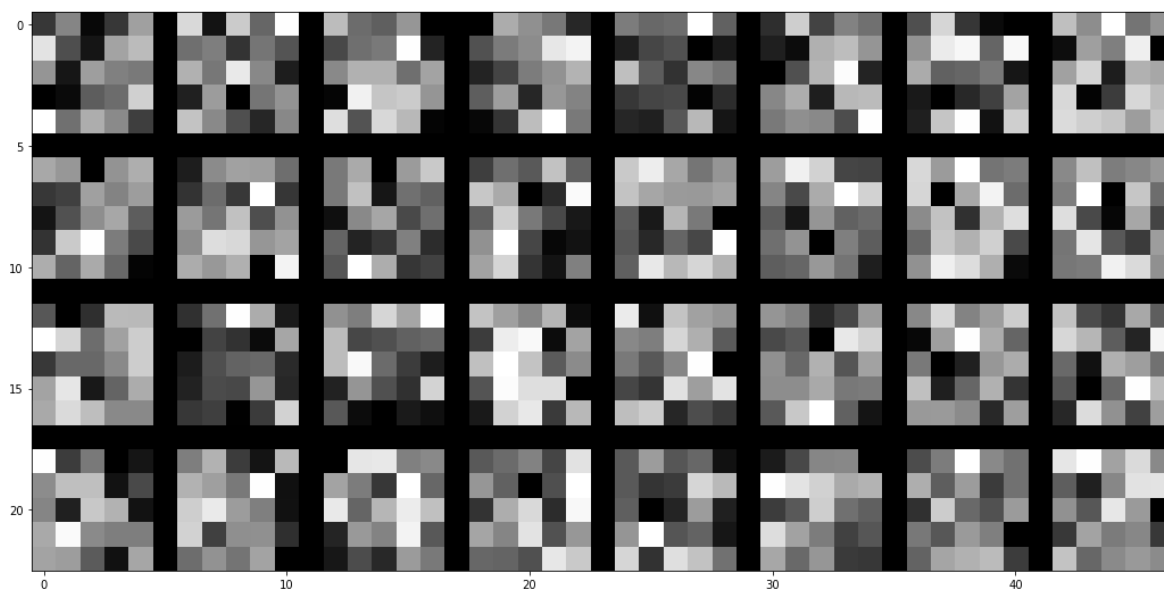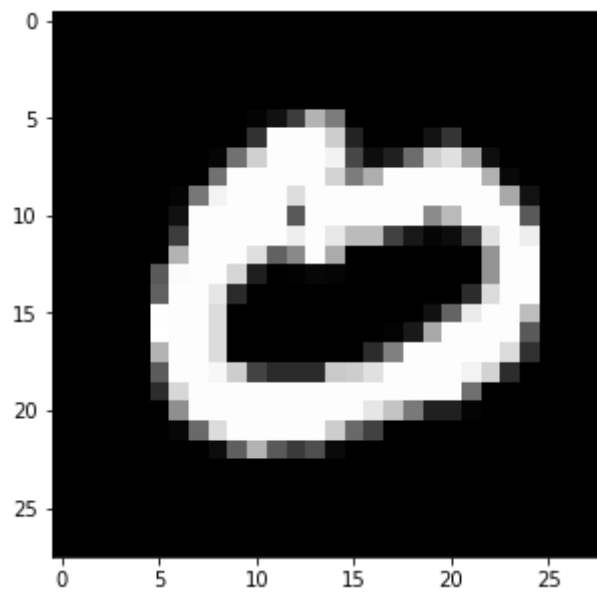
Do you want to see the output of an image passing through first convolution layer?

<matplotlib.image.AxesImage at 0x7fc98e6d0e10>

What about second convolution layer?

Want to learn more?

Running deep learning programs usually needs a high performance platform. PowerAI speeds up deep learning and AI. Built on IBM's Power Systems, PowerAI is a scalable software platform that accelerates deep learning and AI with blazing performance for individual users or enterprises. The PowerAI platform supports popular machine learning libraries and dependencies including TensorFlow, Caffe, Torch, and Theano. You can use [PowerAI on IMB Cloud (https://cocl.us/ML0120EN_PAI)](https://cocl.us/ML0120EN_PAI).

Also, you can use Watson Studio to run these notebooks faster with bigger datasets.Watson Studio is IBM's leading cloud solution for data scientists, built by data scientists. With Jupyter notebooks, RStudio, Apache Spark and popular libraries pre-packaged in the cloud, Watson Studio enables data scientists to collaborate on their projects without having to install anything. Join the fast-growing community of Watson Studio users today with a free account at [Watson Studio (https://cocl.us/ML0120EN_DSX)](https://cocl.us/ML0120EN_DSX).This is the end of this lesson. Thank you for reading this notebook, and good luck on your studies.

Thanks for completing this lesson!

Created by [Saeed Aghabozorgi (https://linkedin.com/in/saeedaghabozorgi)](https://linkedin.com/in/saeedaghabozorgi), [Luis Otavio Silveira Martins (https://linkedin.com/in/luisotsm)](https://linkedin.com/in/luisotsm), [Erich Natsubori Sato (https://linkedin.com/in/erich-natsubori-sato)](https://linkedin.com/in/erich-natsubori-sato) </h4>

References:

[https://en.wikipedia.org/wiki/Deep_learning (https://en.wikipedia.org/wiki/Deep_learning)](https://en.wikipedia.org/wiki/Deep_learning)
[http://ruder.io/optimizing-gradient-descent/ (http://ruder.io/optimizing-gradient-descent/)](http://ruder.io/optimizing-gradient-descent/)
[http://yann.lecun.com/exdb/mnist/ (http://yann.lecun.com/exdb/mnist/)](http://yann.lecun.com/exdb/mnist/)
[https://www.quora.com/Artificial-Neural-Networks-What-is-the-difference-between-activation-functions (https://www.quora.com/Artificial-Neural-Networks-What-is-the-difference-between-activation-functions)](https://www.quora.com/Artificial-Neural-Networks-What-is-the-difference-between-activation-functions)
[https://www.tensorflow.org/versions/r0.9/tutorials/mnist/pros/index.html (https://www.tensorflow.org/versions/r0.9/tutorials/mnist/pros/index.html)](https://www.tensorflow.org/versions/r0.9/tutorials/mnist/pros/index.html)