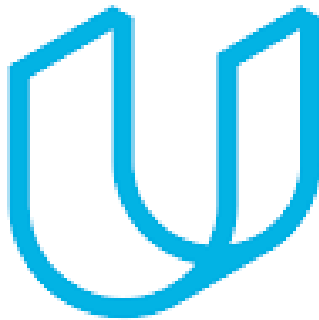


Deep Learning Capstone Project

Identifying Handwritten Digits

Udacity Machine Learning Engineer Nanodegree
Capstone Project

Diego Menin – September 2016



UDACITY

I. Definition

Project Overview

The goal of this project is to build a tool that can identify handwritten numbers in a given image. It will be trained using the MNIST database (see reference section in the end) and will provide the ability to be tested using any test image on the training dataset or any image* the user would like as an input.

* A few restrictions will be applied to this rule, as it will be discussed latter.

Problem Statement

The human visual system is a masterpiece of evolution. Most people can recognize handwritten digits (no matter how bad they are written) with very little effort. But that ease is deceptive. According to Wikipedia, in each hemisphere of our brain, humans have a primary visual cortex, also known as V1, containing 140 million neurons, with tens of billions of connections between them. And yet human vision involves not just V1, but an entire series of visual cortices - V2, V3, V4, and V5 - doing progressively more complex image processing. We carry in our heads a supercomputer, tuned by evolution over hundreds of millions of years, and superbly adapted to understand the visual world. But nearly all that work is done unconsciously. And so we don't usually appreciate how tough a problem our visual systems solve.

The difficulty of visual pattern recognition becomes apparent if you attempt to write a computer program to recognize digits like those above. What seems easy when we do it ourselves suddenly becomes extremely difficult. Simple intuitions about how we recognize shapes - "a 9 has a loop at the top, and a vertical stroke in the bottom right" - turn out to be not so simple to express algorithmically.

Neural networks approach the problem in a different way. The idea is to take a large number of handwritten digits, known as training examples, and then develop a system which can learn from those training examples. In other words, the neural network uses the examples to automatically infer rules for recognizing handwritten digits. Furthermore, by increasing the number of training examples, the network can learn more about handwriting, and so improve its accuracy.






The task of handwritten digit recognition was chosen because it has great importance and use such as on-line handwriting recognition on computer tablets, recognize zip codes on mail for postal mail sorting, processing bank check amounts, numeric entries in forms filled up by hand (for example - tax forms) and so on.

Metrics

Since I am dealing with a classification problem with a predefined number of classes (0 to 9) and the training dataset is evenly distributed among all classes, I will be using "accuracy" to measure the performance of my model.

II. Analysis

Important note: All the code shown on this report can be reproduced in two different ways. The advised way is to open the correspondent Jupyter Notebook and just run the code. There is one notebook for each major step:

<input type="checkbox"/>	 test_images	
<input type="checkbox"/>	 1_DataExploration.ipynb	Running
<input type="checkbox"/>	 2_ModelTraining.ipynb	Running
<input type="checkbox"/>	 3_ModelTesting_AnyImage.ipynb	Running
<input type="checkbox"/>	 3_ModelTesting_FromTestSet.ipynb	Running

Alternately, the FinalProject.py file can be executed uncommenting the desired session on the FinalProject.py file

```
# 1) DATA EXPLORATION
# d = DigitRecognition()
# d.LoadDataSet()
# d.showSingleImageFromTestSet(30)
# d.showTopXImageFromTestSet(30)

# 2) TRAINING
# d = DigitRecognition()
# d.LoadDataSet()
# d.create_nn_model()
# d.StartTraining(10000)

# 3) TESTING
# dTest = DigitRecognition()
# dTest.LoadDataSet()
```

See the "algorithms and techniques" session for dependencies.

Data Exploration

The MNIST database of handwritten digits has a training set of 60,000 examples, and a test set of 10,000 examples. The digits have been size-normalized and centred in a fixed-size image. It is a good database for people who want to try learning techniques and pattern recognition methods on real-world data while spending minimal efforts on pre-processing and formatting. It can be downloaded from the website mentioned on the links section. Four files are available on the site:

- train-images-idx3-ubyte.gz: training set images (9912422 bytes)
- train-labels-idx1-ubyte.gz: training set labels (28881 bytes)
- t10k-images-idx3-ubyte.gz: test set images (1648877 bytes)
- t10k-labels-idx1-ubyte.gz: test set labels (4542 bytes)

The database was constructed from NIST's Special Database 3 and Special Database 1 which contain binary images of handwritten digits. NIST originally designated SD-3 as their training set and SD-1 as their test set. However, SD-3 is much cleaner and easier to recognize than SD-1. The reason for this can be found on the fact that SD-3 was collected among Census Bureau employees, while SD-1 was collected among high-school students.

Drawing sensible conclusions from learning experiments requires that the result be independent of the choice of training set and test among the complete set of samples. Therefore it was necessary to build a new database by mixing NIST's datasets.

The data available on the website is stored in a very simple file format designed for storing vectors and multidimensional matrices. There are several classes available online to extract the data on a more readable format. I chose to use a class called "mnist_data.py" from google, that offers functions to download and extract the data – if not present on a pre-defined folder (I'm using a folder called "mydata").

After loaded, each image will be represented as a (28, 28, 1) matrix of pixels. The code mentioned above also takes care of extracting the labels into a 1D numpy array [index] and converts the pixels from [0, 255] to the [0.0, 1.0] range. The labels are "one-hot" encoded, meaning that they are stored in a length 10 vector where the correct class has value 1 and all other classes have value 0.

```
In [32]: print 'Image shape:'  
         d.mnist.train.images[0].shape
```

```
Image shape:
```

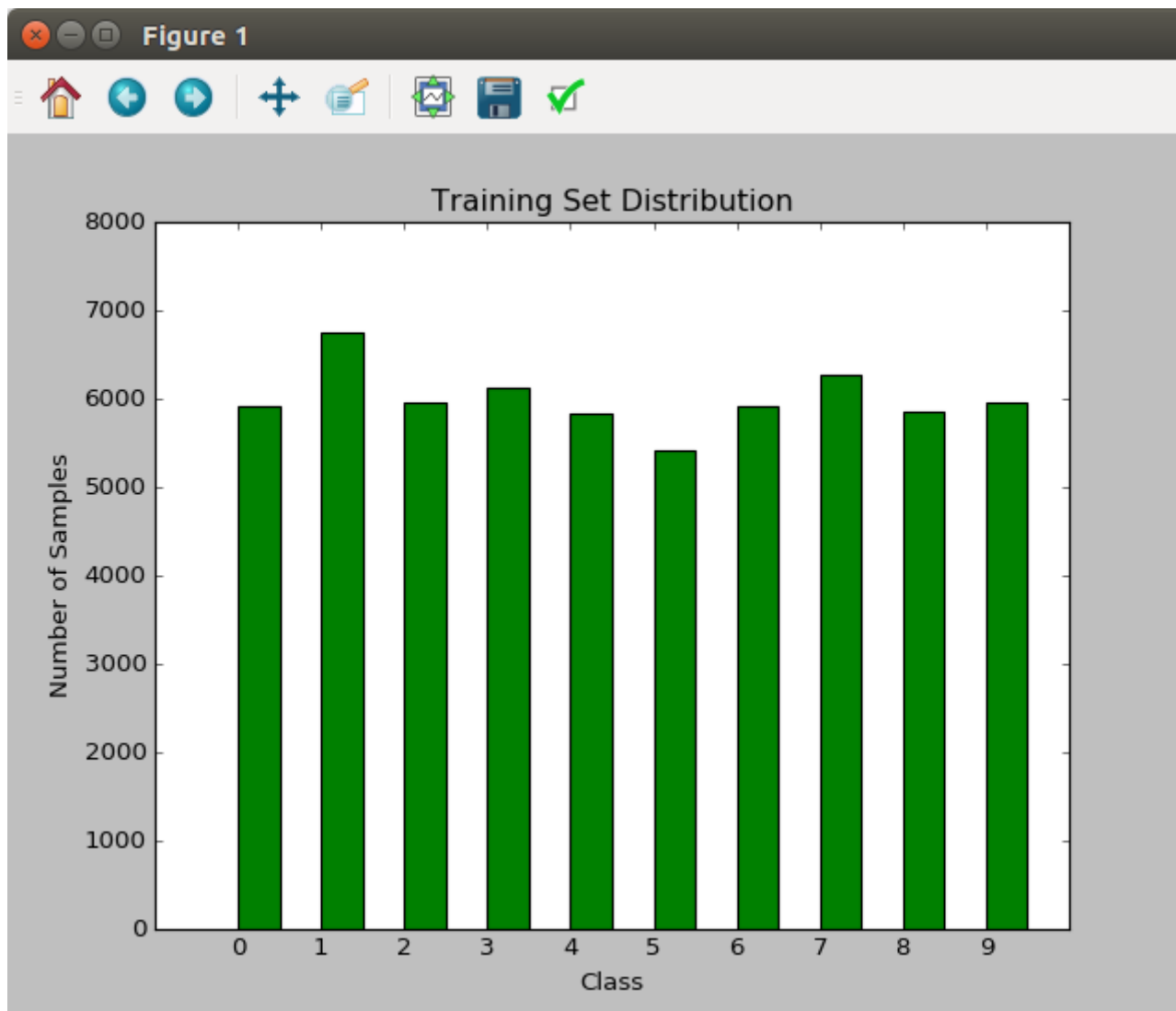
```
Out[32]: (28, 28, 1)
```

```
In [33]: d.mnist.train.labels[0]
```

```
Out[33]: array([ 0.,  0.,  0.,  0.,  0.,  1.,  0.,  0.,  0.,  0.])
```

Exploratory Visualization

As mentioned before, the data is evenly distributed across the classes. That can be checked on the "1_DataExploration" jupyter notebook by calling the "getTrainingDataDistribution()" function:



On the same notebook, we have the option of visualizing any of the digits on the test set (by passing its index as a parameter) or the first X images and their label:

The screenshot shows a Jupyter Notebook titled "DataExploration" with a last checkpoint of 22 minutes ago. The interface includes a menu bar (File, Edit, View, Insert, Cell, Kernel, Help) and a toolbar with icons for saving, adding cells, and running code. The notebook contains several code cells:

```
In [1]: import FinalProject as fp
```

```
In [2]: d = fp.DigitRecognition()
Class Initialized
```

```
In [3]: d.LoadDataSet()
Loading mydata/train-images-idx3-ubyte.mnist
Loading mydata/train-labels-idx1-ubyte.mnist
Loading mydata/t10k-images-idx3-ubyte.mnist
Loading mydata/t10k-labels-idx1-ubyte.mnist
```

```
In [*]: d.showSingleImageFromTestSet(0)
```

```
In [*]: d.showTopXImageFromTestSet(30)
```

```
In [ ]:
```

Overlaid on the right side of the notebook is a 10x10 grid of handwritten digits from the MNIST dataset. The digits are as follows:

7	2	1	0	4
1	4	9	5	9
0	6	9	0	1
5	9	7	3	4
9	6	6	5	4
0	7	4	0	1
0	7	4	0	1
0	7	4	0	1
0	7	4	0	1
0	7	4	0	1

Algorithms and Techniques

The overall idea is to use Tensorflow and Neural Networks to solve this problem. Other auxiliary libraries were used like "cv2" for image processing and "matplotlib" for plotting.

Several networks were tested, since very simple $y = WX + B$ models, until more complex solutions like convolutional networks with several layers.

Another technique used on the final model was "learning rate decay". In training deep networks, it is usually helpful to anneal the learning rate over time. Good intuition to have in mind is that with a high learning rate, the system contains too much kinetic energy and the parameter vector bounces around chaotically, unable to settle down into deeper, but narrower parts of the loss function. Knowing when to decay the learning rate can be tricky: Decay it slowly and you'll be wasting computation bouncing around chaotically with little improvement for a long time.

But decay it too aggressively and the system will cool too quickly, unable to reach the best position it can. There are three common types of implementing the learning rate decay: Step decay, Exponential decay and $1/t$ decay (see the reference links for more information)

All 3 approaches were tested on the final model and the exponential decay proved to produce a smaller loss and bigger final accuracy than the others.

More details will be provided on the "Implementation" section below.

Benchmark

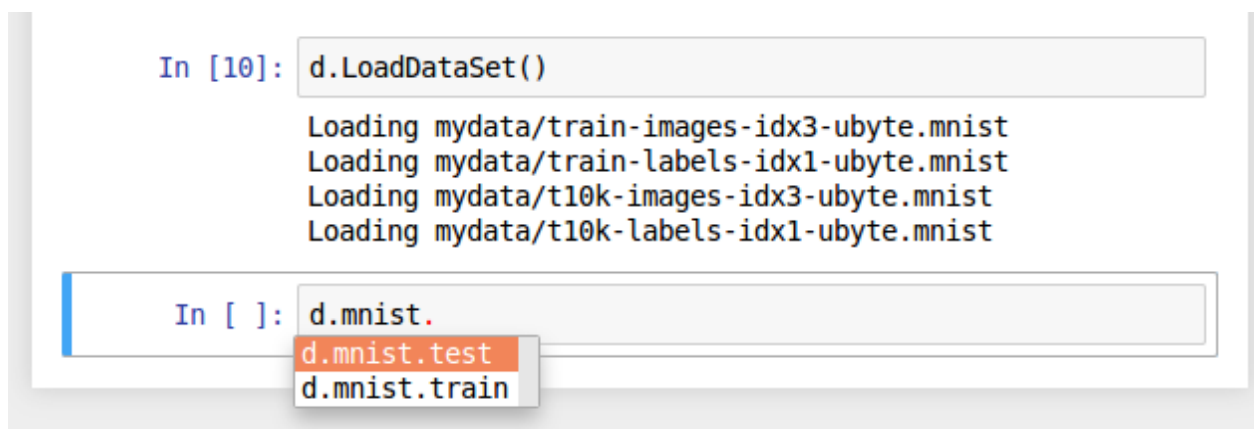
The simplest implementation to this problem can be found on the Tensorflow tutorial (see link section for more details) and outputs an accuracy of 92%. So that will be my starting point. The same tutorial also mentions that with simple changes we can get to 97% accuracy, which is the target I'll be trying to reach (and pass). It also mentions that the best models can get 99.7% accuracy.

III. Methodology

Data Pre-processing

Fortunately, as mentioned before, there are several classes available online that can do the pre-processing for you. I chose to use a class called "mnist_data.py" from goggle that downloads and extract the data into the "mydata" folder.

I've encapsulated everything on the "LoadDataSet" method. Once it is called, the train and test data is available in a mnist object:



```
In [10]: d.LoadDataSet()

Loading mydata/train-images-idx3-ubyte.mnist
Loading mydata/train-labels-idx1-ubyte.mnist
Loading mydata/t10k-images-idx3-ubyte.mnist
Loading mydata/t10k-labels-idx1-ubyte.mnist

In [ ]: d.mnist.
        d.mnist.test
        d.mnist.train
```

Implementation

The dataset will be read on the training step (using the function already mentioned on the previous items) and the training will happen in X iterations. Notebook "2_ModelTraining" shows the training process:

```
File Edit View Insert Cell Kernel Help Python [Root]
In [1]: import FinalProject as fp
In [3]: d = fp.DigitRecognition()
Class Initialized
In [4]: d.LoadDataSet()
Loading mydata/train-images-idx3-ubyte.mnist
Loading mydata/train-labels-idx1-ubyte.mnist
Loading mydata/t10k-images-idx3-ubyte.mnist
Loading mydata/t10k-labels-idx1-ubyte.mnist
In [6]: d.create_nn_model()
In [8]: d.StartTraining(10000)

Starting Training
Train Iteration 0 (Lerning Rate:0.003): accuracy:0.08 loss: 252.568
Train Iteration 20 (Lerning Rate:0.00297114451787): accuracy:0.77 loss: 65.9423
Train Iteration 40 (Lerning Rate:0.00294257615259): accuracy:0.87 loss: 29.0371
Train Iteration 60 (Lerning Rate:0.00291429204729): accuracy:0.93 loss: 22.6654
Train Iteration 80 (Lerning Rate:0.00288628937354): accuracy:0.86 loss: 37.8008
Train Iteration 100 (Lerning Rate:0.00285856533105): accuracy:0.93 loss: 19.8268
Test Results: EPOCH 1 - Iteration: 100 Accuracy:0.9448 Test Loss: 19.0742 - Accuracy Improvement - Saving Checkpoint
```

Each iteration will work on 100 image batches. Since we are dealing with a training set of 6000 images, after the 600th iteration, a new EPOCH will start. Every 20 iterations, the accuracy and loss of the model will be evaluated on the train dataset and every 100 iterations, those metrics will be calculated against the test set. If there is an accuracy improvement, the "model" (or checkpoint) will be saved to the "/checkpoints" folder. After the first epoch is complete, is not common to see situations where the accuracy doesn't increase for a while:

```
Test Results: EPOCH 2 - Iteration: 700 Accuracy:0.9848 Test Loss: 4.75714 - Accuracy Improvement - Saving Checkpoint
Train Iteration 720 (Lerning Rate:0.00212326134561): accuracy:0.93 loss: 13.3391
Train Iteration 740 (Lerning Rate:0.00210312955885): accuracy:0.96 loss: 10.1566
Train Iteration 760 (Lerning Rate:0.00208319808672): accuracy:0.98 loss: 2.66198
Train Iteration 780 (Lerning Rate:0.00206346493604): accuracy:0.96 loss: 8.26623
Train Iteration 800 (Lerning Rate:0.0020439281335): accuracy:0.99 loss: 4.95453
Test Results: EPOCH 2 - Iteration: 800 Accuracy:0.9838 Test Loss: 4.95084 - No Accuracy Improvement
Train Iteration 820 (Lerning Rate:0.0020245857254): accuracy:0.98 loss: 3.23048
Train Iteration 840 (Lerning Rate:0.00200543577746): accuracy:0.97 loss: 5.98989
Train Iteration 860 (Lerning Rate:0.0019864763747): accuracy:0.99 loss: 5.62996
Train Iteration 880 (Lerning Rate:0.00196770562114): accuracy:1.0 loss: 0.653053
Train Iteration 900 (Lerning Rate:0.0019491216397): accuracy:0.96 loss: 12.8054
Test Results: EPOCH 2 - Iteration: 900 Accuracy:0.9854 Test Loss: 4.40138 - Accuracy Improvement - Saving Checkpoint
Train Iteration 920 (Lerning Rate:0.00193072257197): accuracy:1.0 loss: 0.39819
```

After the training is complete, we'll be left with the best "model" on the checkpoints folder, ready to be read for testing. One word of advice though, if the training starts again, it will ignore what is already on the present folder and overwrite it, so caution must be taken if is desired to keep more than one model saved.

Refinement

The final model is defined in the `create_nn_model` function. This function is so independent from the rest of the code that it can easily be changed to test new models. In fact, I left a very simple function (and the first one that I tried), like the one mentioned on the basic Tensorflow tutorial, available as an example. As the tutorial mentions, it can achieve a max of 92% accuracy, even after 10000 training iterations:

```
In [4]: d.create_simple_model()
```

```
In [6]: d.StartTraining(10000)
```

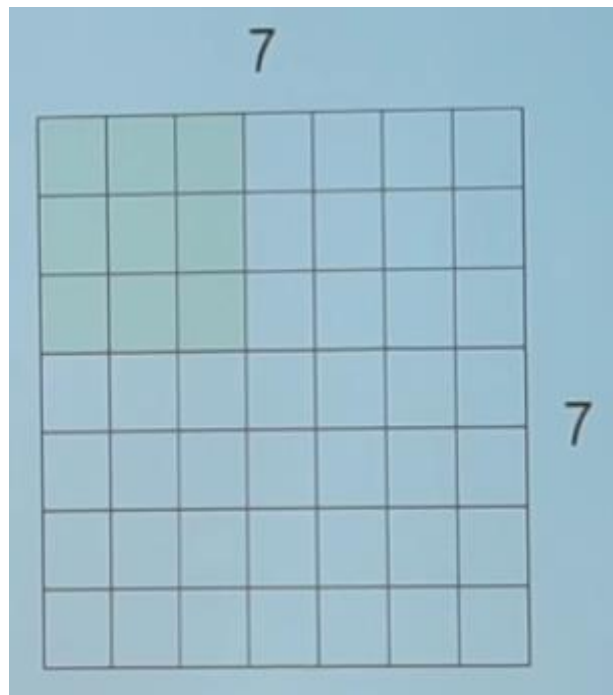
```
Test Results: EPOCH 17 - Iteration: 9700 Accuracy:0.9273 Test Loss: 26.2253 - No Accuracy Improvement
Train Iteration 9720 (Lerning Rate:0.000122476403284): accuracy:0.96 loss: 26.7603
Train Iteration 9740 (Lerning Rate:0.000122252759335): accuracy:0.93 loss: 31.7704
Train Iteration 9760 (Lerning Rate:0.00012203134068): accuracy:0.96 loss: 13.8901
Train Iteration 9780 (Lerning Rate:0.000121812125177): accuracy:0.93 loss: 18.3726
Train Iteration 9800 (Lerning Rate:0.000121595090906): accuracy:0.97 loss: 16.8192
Test Results: EPOCH 17 - Iteration: 9800 Accuracy:0.9267 Test Loss: 26.2086 - No Accuracy Improvement
Train Iteration 9820 (Lerning Rate:0.000121380216161): accuracy:0.92 loss: 23.4248
Train Iteration 9840 (Lerning Rate:0.000121167479456): accuracy:0.95 loss: 18.837
Train Iteration 9860 (Lerning Rate:0.000120956859516): accuracy:0.93 loss: 22.0343
Train Iteration 9880 (Lerning Rate:0.00012074833528): accuracy:0.95 loss: 30.2417
Train Iteration 9900 (Lerning Rate:0.000120541885894): accuracy:0.91 loss: 26.3041
Test Results: EPOCH 17 - Iteration: 9900 Accuracy:0.9278 Test Loss: 26.1514 - No Accuracy Improvement
Train Iteration 9920 (Lerning Rate:0.000120337490714): accuracy:0.94 loss: 17.4859
Train Iteration 9940 (Lerning Rate:0.000120135129301): accuracy:0.91 loss: 35.2808
Train Iteration 9960 (Lerning Rate:0.000119934781417): accuracy:0.98 loss: 11.1933
Train Iteration 9980 (Lerning Rate:0.000119736427027): accuracy:0.91 loss: 30.9417
Train Iteration 10000 (Lerning Rate:0.000119540046297): accuracy:0.94 loss: 20.6356
Test Results: EPOCH 17 - Iteration: 10000 Accuracy:0.9272 Test Loss: 26.1688 - No Accuracy Improvement
Training complete, max accuracy achieved: 0.928
```

This basic model is as simple $WX + B$ one layer linear model. From that model, until the final version, several attempts were made. For the sake of simplicity I won't go through each one in detail (especially because I wasn't keeping track of the impact of every single change), I'll go straight to the final solution.

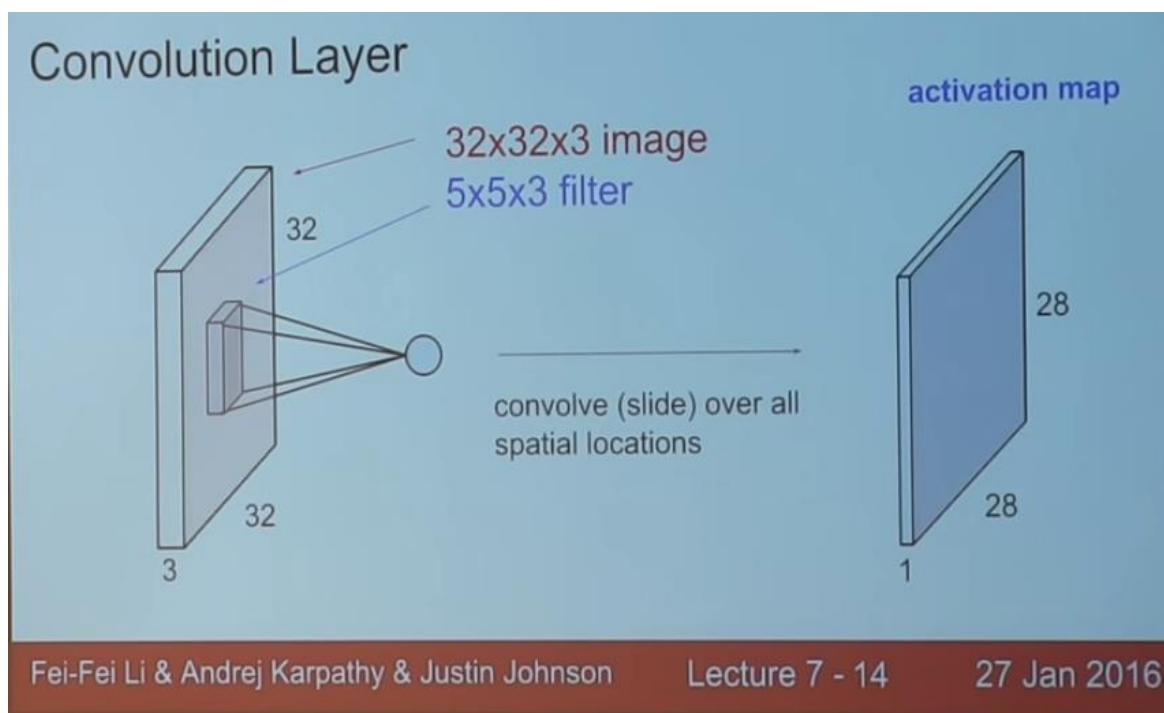
The main driver for the performance improvement was the addition of Convolutional Layers. I won't explain them in deep detail because that would take too long and is not part of the scope of this project, but the idea is to create small "filters" and scan them through the image (so instead of doing a weighted sum across all pixels in the image, it does only on a subset of the image) producing what is called "activation maps".

FYI: all the images bellow are from the "CS231n Winter 2016: Lecture 7: Convolutional Neural Networks" (the link to the videos can be found on the "links" section in the end of this report).

For example, given a 7 X 7 image as the one bellow, we can create a 3 X 3 filter and "move it" to the right using a particular "stride".



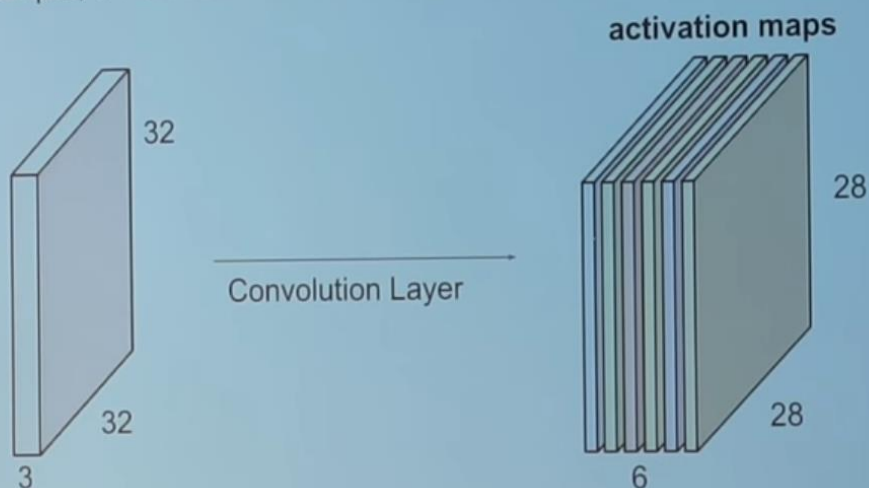
The stride indicates how much does the filter move in one direction, so for example, a stride of 1, moves the filter one pixel at a time, a stride of two moves it two pixels at a time and so on.



The result is (28X28) because there are 28 unique positions for the 5X5X3 filter.

The "activation maps" can then be stacked on top of each other producing a new image, on the example bellow of (28 X 28 X 6):

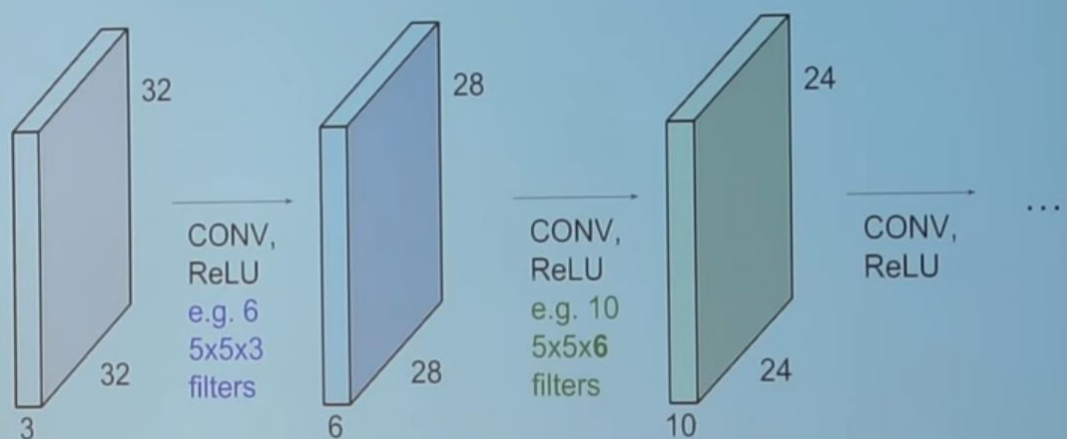
For example, if we had 6 5x5 filters, we'll get 6 separate activation maps:



We stack these up to get a "new image" of size 28x28x6!

Then, several convolutional layers can be used together to improve performance:

Preview: ConvNet is a sequence of Convolutional Layers, interspersed with activation functions



The final model is composed of:

	Filter Size	Input channels	Output channels	Stride	Output size
Convolution 1	6 X 6	1	6	1	28 X 28
Convolution 2	5 X 5	6	12	2	14 X 14
Convolution 3	4 X 4	12	24	2	7 X 7

After the third Convolution layer, I reshape the output from the third convolution for the fully connected layer to be able to run the weighted sum across the one line.

Adding convolutional layers boosted the accuracy up to 99%. The last piece of improvement was achieved by adding a technique called "dropout" which works as follows: one or more neural network node is switched off once in a while so that it will not interact with the network (it weights cannot be updated, nor affect the learning of the other network nodes). With dropout, the learned weights of the nodes become somewhat more insensitive to the weights of the other nodes and learn to decide somewhat more by their own (and less dependent on the other nodes they're connected to). In general, dropout helps the network to generalize better and increase accuracy since the (possibly somewhat dominating) influence of a single node is decreased by dropout.

The next session will give more details on the final model.

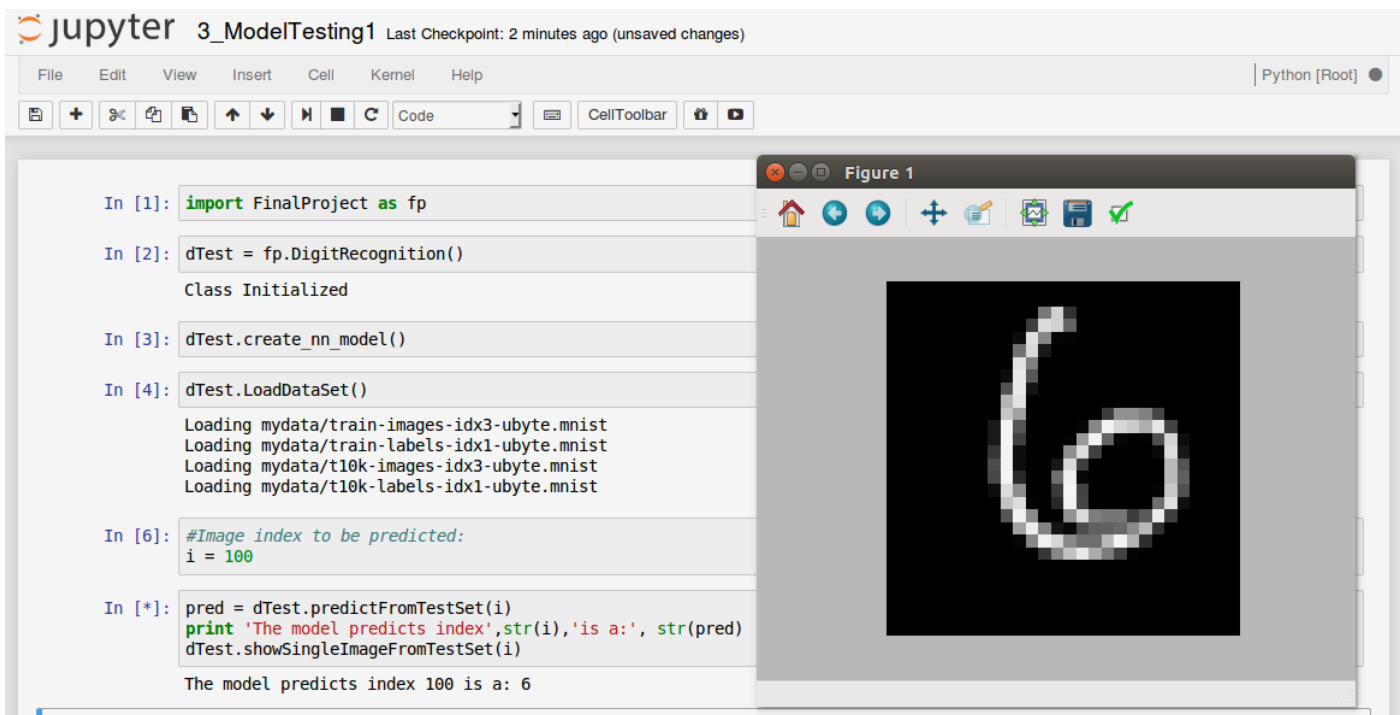
IV. Results

Model Evaluation and Validation

As mentioned before, during training, the model is being constantly evaluated. The final model (after 10000 iterations) resulted with an accuracy of 99.35% , which is a lot higher than I was expecting and mentioned on the "Benchmark" session.

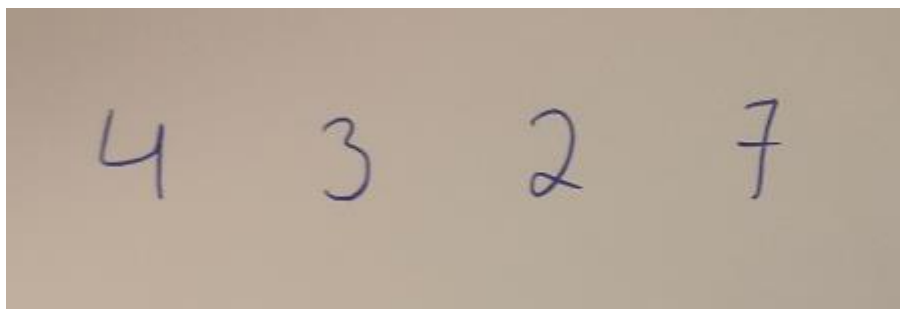
To manually evaluate the model, there are two notebooks that can be used. Both of them read the checkpoint files created on the training step from the "checkpoints" folder.

The first notebook only tests the model using an image from the current test data set. All it needs to be done is to inform an index and the notebook will output the actual image and the predicted value using the model.



The second notebook, called "3_ModelTesting_AnyImage" accepts any image as input. From the input image, it will try to find numbers in it and for each number found, it will use the model to predict which number it is. In order to do so, the following steps need to be followed:

- 1) Place the desired image in the "test_images" folder. Don't worry if there are more images or even folders in there. Here is an example image (this is actually my own handwriting):



- 2) Call the test procedure.
- 3) A folder called "<your_image_name>_result" will be created to store the prediction output. If the folder exist, its content will be deleted so we always get a fresh start.
- 4) A image also called "<your_image_name>_result" will be created where you will be able to see the numbers recognized and the prediction made to each one of them:

```
File Edit View Insert Cell Kernel Help

In [1]: import FinalProject as fp

In [2]: dTest = fp.DigitRecognition()
Class Initialized

In [3]: dTest.create_nn_model()

In [6]: #ImageName
#Place the Image in the "test_images" folder:
imageName = 'hand'

In [7]: pred = dTest.testNewImage(imageName)

13.3978814672 14.9420467478
Prediction for (0, 0, 100)
[1118.575, 4]
13.655796758 14.4263911271
Prediction for (100, 0, 100)
[663.76965, 3]
14.8083033442 13.1039234618
Prediction for (200, 0, 100)
[937.48303, 2]
11.9373760272 14.0446302069
Prediction for (300, 0, 100)
[360.07224, 7]
```



Important note: the algorithm that crops the image is very sensitive to the background so it most likely won't work on a non-plain, whitish background. Due to the scope of the project, which is mainly to build the neural network, I didn't consider it a big deal and didn't spend too much time dealing with this problem. I'd definitely include this in a "future improvements" list. In fact, the best way to have some fun with the model is to use an image manipulation program to "write" the digits on a white background.

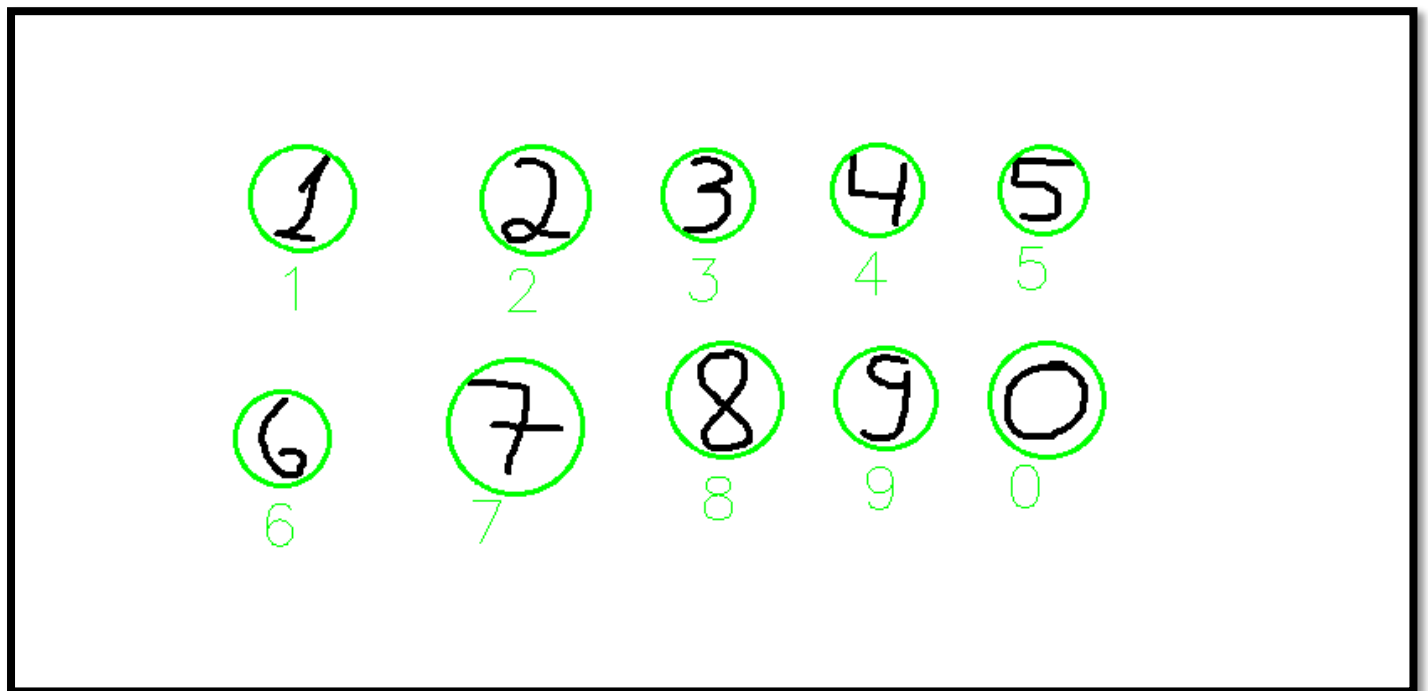
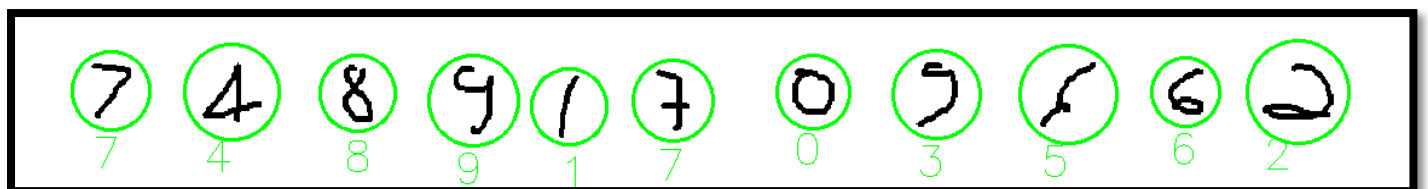
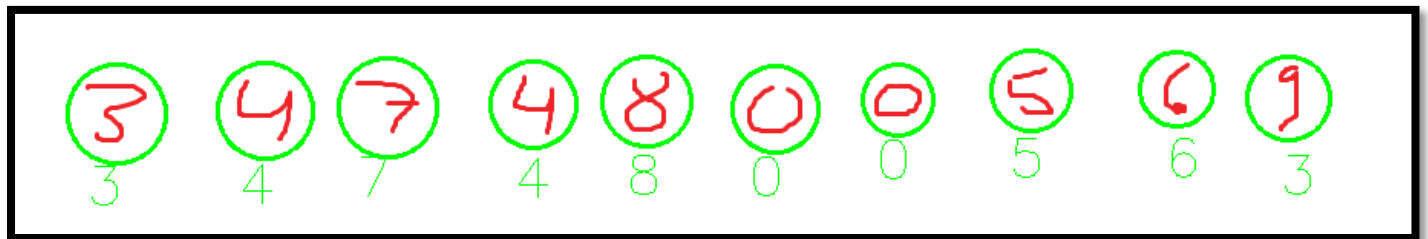
Justification

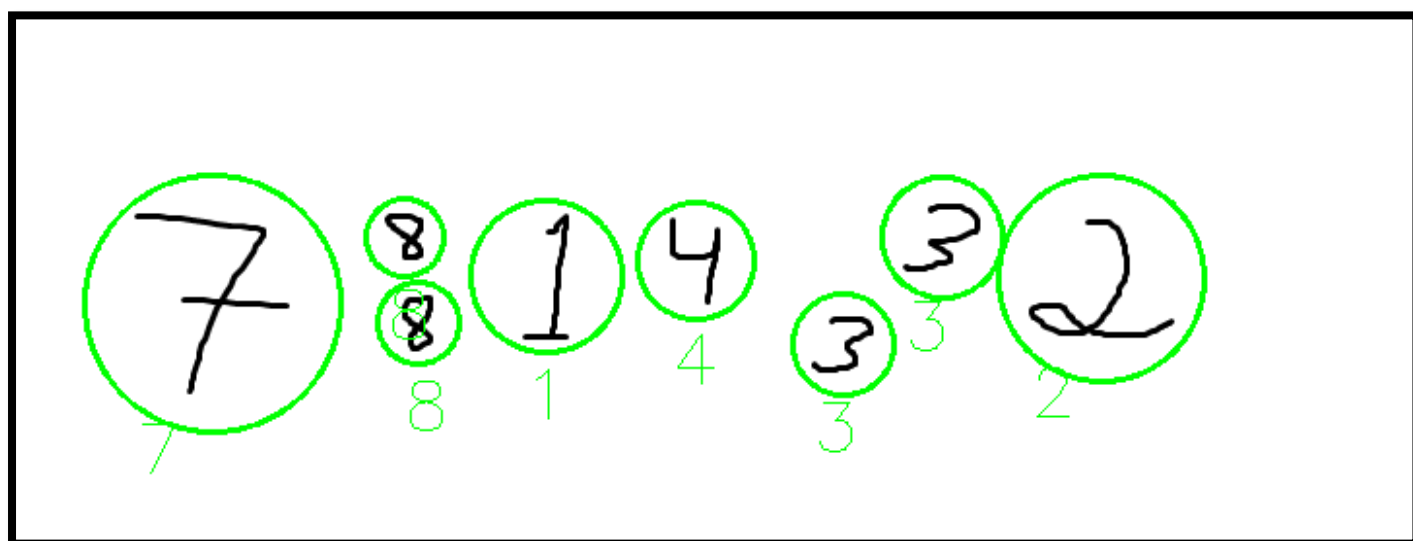
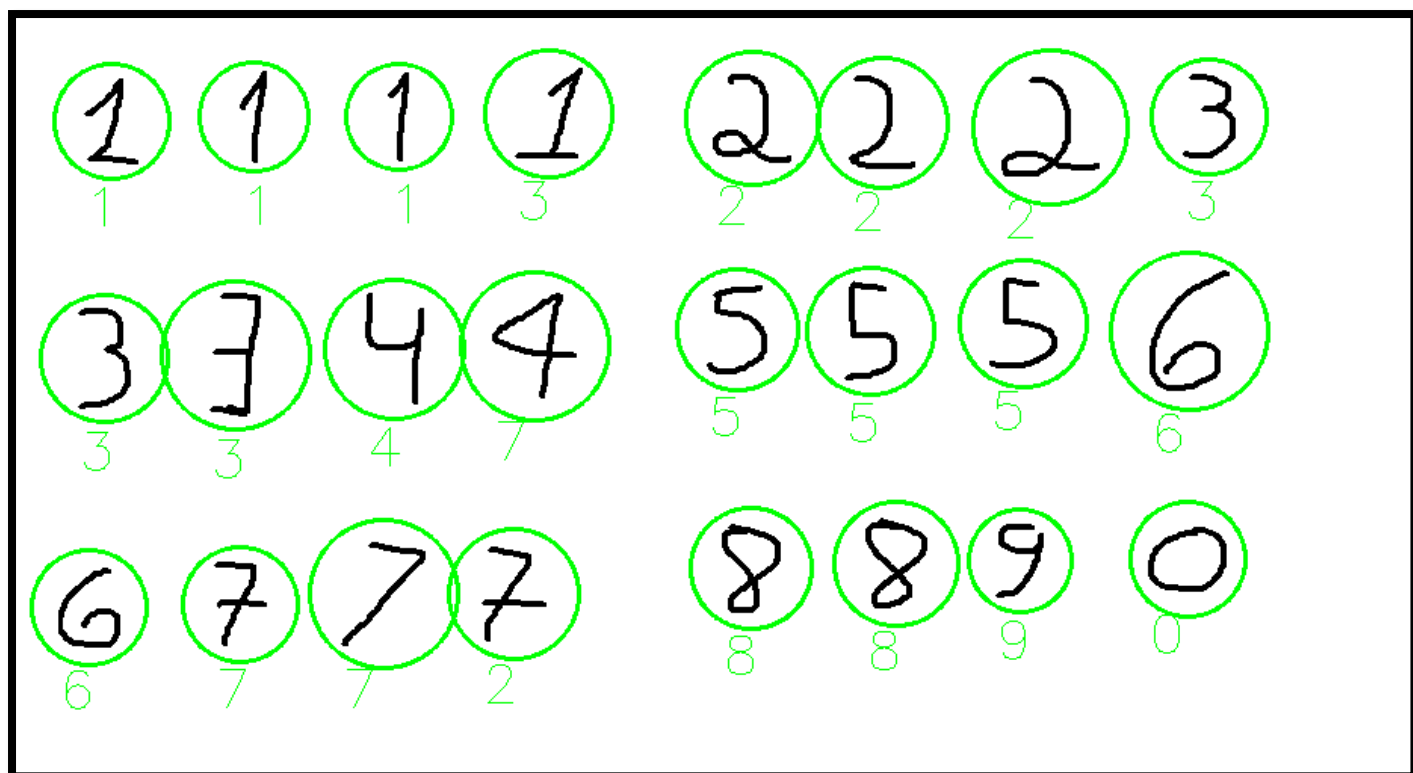
As mentioned on the Benchmark section, the simplest implementation to this problem achieves an accuracy of 92%, an elaborate solution, 97% and "state of the art" models, 99.7%. My goal was to implement a solution that could achieve an accuracy between those two last values. Since I achieve 99.35% on the test set and the model deals with new images, I believe I've achieved that goal.

V. Conclusion

Free-Form Visualization

Here are a few other examples of the result. In all of the examples I used the mouse to "write" on paint (the frame was added later on, it was not part of the picture):





Reflection

In summary, on this project, I trained a model used the MNIST dataset to identify handwritten digits. The model then can be used to identify any new imaged desired by the user. I felt it was a very interesting project to work on due to the complexity and the relevance of the problem.

One of the biggest challenges I've found was precisely to master the concepts and the Tensorflow library as it was completely new to me since it hasn't been cover during the nanodegree.

In the end, the final model and solution exceed my expectations. I believe I've built not only a valuable tool, but also it is very well organized and should be easy to use and understand in case anyone else wants to look or study it.

Improvement

As previously mentioned, the main goal of this project was to build a neural network that can learn from data and predict new values. Not much time was spend worrying about where those number would be and on which type of background. All of my testing was done using handwriting on a plain background or "handwriting using the mouse" on paint. Even though that is ok to test the core of the model, it would be very interesting to spend more time refining the algorithm to work regardless of the different possible image backgrounds.

Useful Links

- The MNIST dataset: <http://yann.lecun.com/exdb/mnist/>
- TensorFlow working on an Ubuntu virtual Machine: <https://dmenin.wordpress.com/2016/08/05/tensorflow-working-on-an-ubuntu-virtual-machine/>
- Installing OpenCV for Python on Ubuntu: <http://stackoverflow.com/questions/25215102/installing-opencv-for-python-on-ubuntu-getting-importerror-no-module-named-cv2>
- Good Introduction to the problem: https://en.wikipedia.org/wiki/Visual_cortex
- Learning rate decay: <http://cs231n.github.io/neural-networks-3/>
- Tensorflow basic tutorial: <https://www.tensorflow.org/versions/r0.9/tutorials/mnist/beginners/index.html>
- CS231n Winter 2016: Lecture 7: Convolutional Neural Networks: https://www.youtube.com/watch?v=LxfUGhug-iQ&list=PLwQyV9I_3POsyBPRNUU_ryNfXzgfiw2p&index=7&spfreload=1
- Dropout: <https://www.quora.com/How-does-the-dropout-method-work-in-deep-learning>