**I. Definition**

**Project Overview**

The goal of this project is to train a model that can identify handwritten numbers in a given image. The model 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**

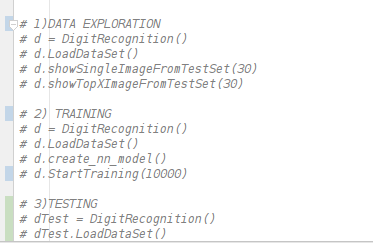
Since I am dealing with a classification problem with a predefined number of classes (0 to 9) and the training dataset is evenly distributed among all classes, I will be using "accuracy" to measure the performance of my model.

**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:*



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



*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:

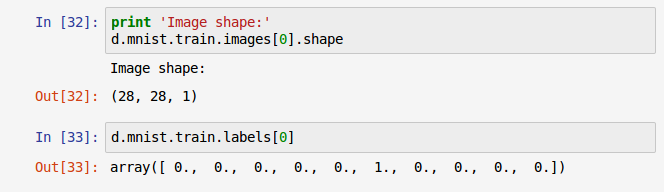
* train-images-idx3-ubyte.gz: training set images (9912422 bytes)
* train-labels-idx1-ubyte.gz: training set labels (28881 bytes)
* t10k-images-idx3-ubyte.gz: test set images (1648877 bytes)
* t10k-labels-idx1-ubyte.gz: test set labels (4542 bytes)

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.

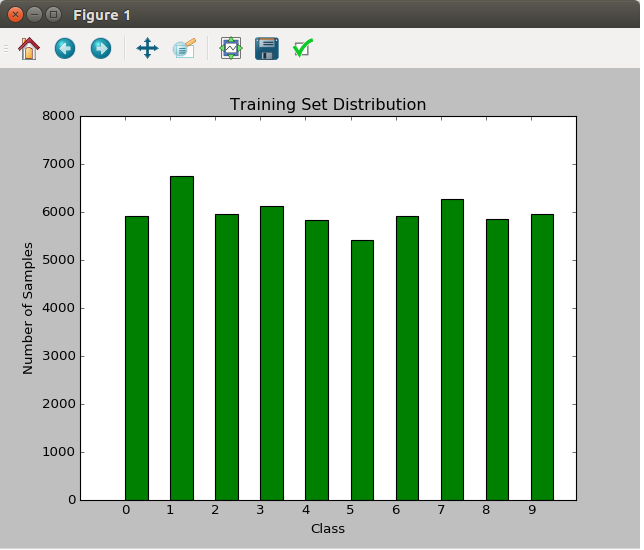
The data available on the website is stored in a very simple file format designed for storing vectors and multidimensional matrices. There are several classes available online to extract the data on a more readable format. I chose to use a class called “mnist\_data.py” from google, that offers functions 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.

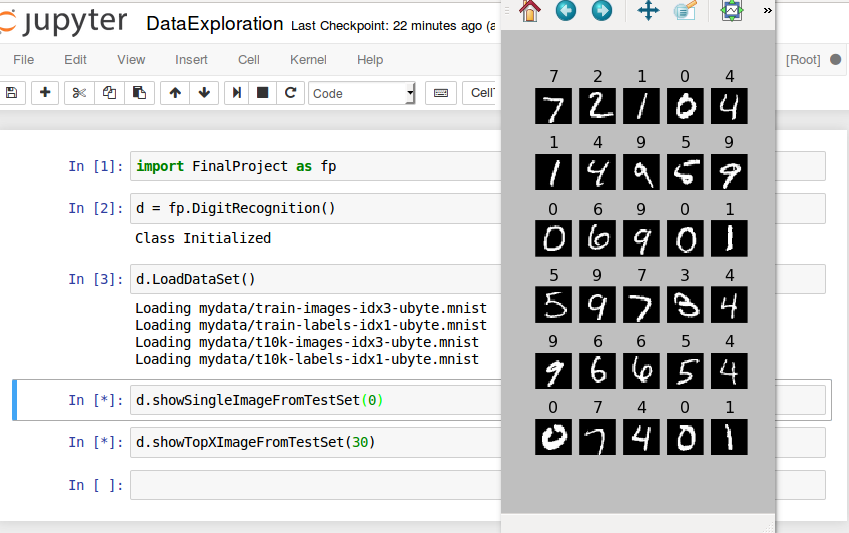


**Exploratory Visualization**

As mentioned before, the data is evenly distributed across the classes. That can be checked on the DataExploration notebook by calling the “getTrainingDataDistribution()” function:



On the same notebook, we have the option of visualizyng any of the digits on the test set (by passign tis index as a parameter) or the firt X images and their label:



**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 complexes 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 bellow.

**Benchmark**

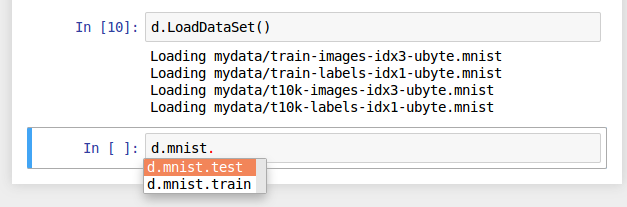
The simplest implementation to this problem can be found on the Tensorflow tutorial (see link section for more details) outputs an accuracy of 92%. So that will be my starting point. The same tutorial also mentions that with simple changes we can get to 97% accuracy, which is the target I’ll be trying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

**III. Methodology**

**Data Pre-processing**

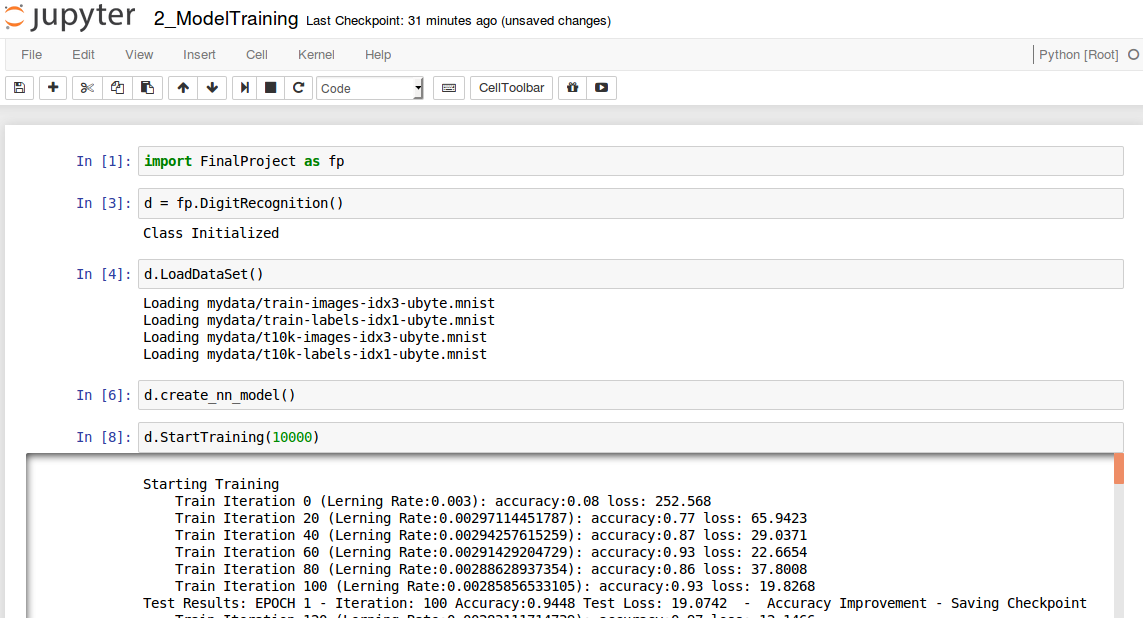
Fortunately, as mentioned before, there are several classes available online that can do the pre-processing for you. I chose to use a class called “mnist\_data.py” from goggle that downloads and extract the data into the "mydata" folder.

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

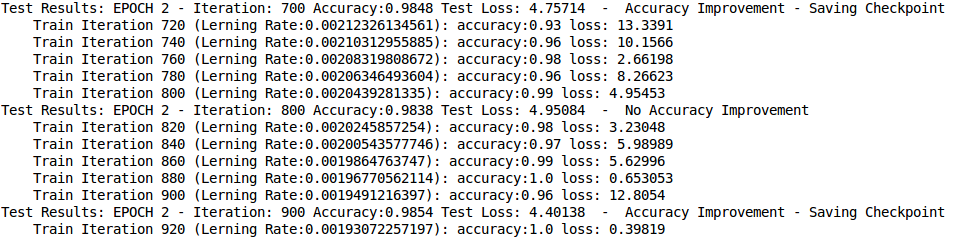
**

**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:



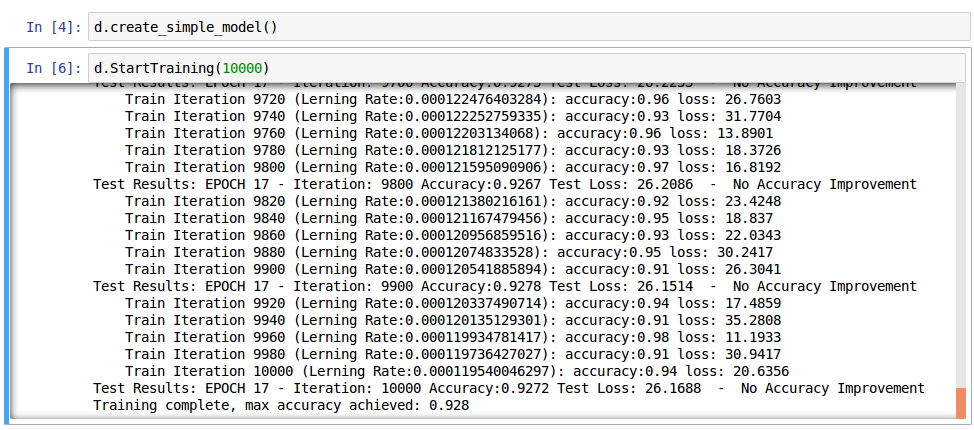
Each iteration will work on 100 image batches. Since we are dealing with a training set of 6000 images, after the 600th iteration, a new EPOCH will start. Every 20 iterations, the accuracy and loss of the model will be evaluated on the train dataset and every 100 iterations, those metrics will be calculated 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:



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 mode 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, 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:

****

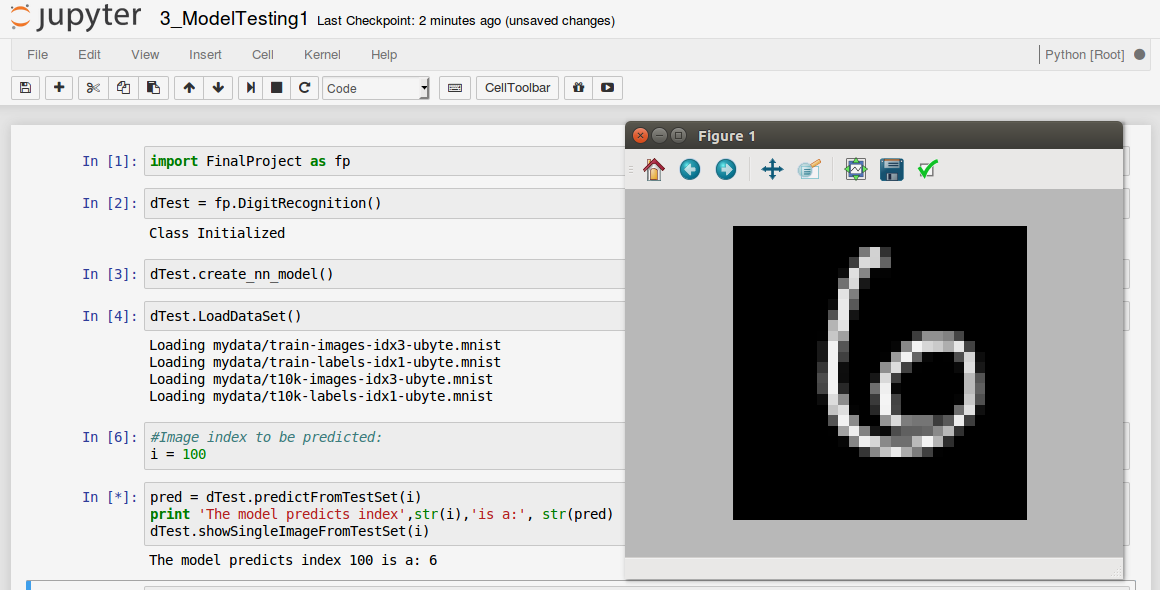
From that basic model, a few things were implemented until the final model. Which are:

* Learning date decay: as already mentioned;
* Adding convolutional layers:
* softmax\_cross\_entropy\_with\_logits
* AdamOptimizer

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

**IV. Results**

**Model Evaluation and Validation**



In this section, the final model and any supporting qualities should be evaluated in detail. It should be clear how the final model was derived and why this model was chosen. In addition, some type of analysis should be used to validate the robustness of this model and its solution, such as manipulating the input data or environment to see how the model’s solution is affected (this is called sensitivity analysis). Questions to ask yourself when writing this section:

* *Is the final model reasonable and aligning with solution expectations? Are the final parameters of the model appropriate?*
* *Has the final model been tested with various inputs to evaluate whether the model generalizes well to unseen data?*
* *Is the model robust enough for the problem? Do small perturbations (changes) in training data or the input space greatly affect the results?*
* *Can results found from the model be trusted?*

**Justification**

In this section, your model’s final solution and its results should be compared to the benchmark you established earlier in the project using some type of statistical analysis. You should also justify whether these results and the solution are significant enough to have solved the problem posed in the project. Questions to ask yourself when writing this section:

* *Are the final results found stronger than the benchmark result reported earlier?*
* *Have you thoroughly analyzed and discussed the final solution?*
* *Is the final solution significant enough to have solved the problem?*

**V. Conclusion**

*(approx. 1-2 pages)*

**Free-Form Visualization**

In this section, you will need to provide some form of visualization that emphasizes an important quality about the project. It is much more free-form, but should reasonably support a significant result or characteristic about the problem that you want to discuss. Questions to ask yourself when writing this section:

* *Have you visualized a relevant or important quality about the problem, dataset, input data, or results?*
* *Is the visualization thoroughly analyzed and discussed?*
* *If a plot is provided, are the axes, title, and datum clearly defined?*

**Reflection**

In this section, you will summarize the entire end-to-end problem solution and discuss one or two particular aspects of the project you found interesting or difficult. You are expected to reflect on the project as a whole to show that you have a firm understanding of the entire process employed in your work. Questions to ask yourself when writing this section:

* *Have you thoroughly summarized the entire process you used for this project?*
* *Were there any interesting aspects of the project?*
* *Were there any difficult aspects of the project?*
* *Does the final model and solution fit your expectations for the problem, and should it be used in a general setting to solve these types of problems?*

**Improvement**

In this section, you will need to provide discussion as to how one aspect of the implementation you designed could be improved. As an example, consider ways your implementation can be made more general, and what would need to be modified. You do not need to make this improvement, but the potential solutions resulting from these changes are considered and compared/contrasted to your current solution. Questions to ask yourself when writing this section:

* *Are there further improvements that could be made on the algorithms or techniques you used in this project?*
* *Were there algorithms or techniques you researched that you did not know how to implement, but would consider using if you knew how?*
* *If you used your final solution as the new benchmark, do you think an even better solution exists?*

**Before submitting, ask yourself. . .**

* Does the project report you’ve written follow a well-organized structure similar to that of the project template?
* Is each section (particularly **Analysis** and **Methodology**) written in a clear, concise and specific fashion? Are there any ambiguous terms or phrases that need clarification?
* Would the intended audience of your project be able to understand your analysis, methods, and results?
* Have you properly proof-read your project report to assure there are minimal grammatical and spelling mistakes?
* Are all the resources used for this project correctly cited and referenced?
* Is the code that implements your solution easily readable and properly commented?
* Does the code execute without error and produce results similar to those reported?
* 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-unbuntu-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>