# Senior Project Presentation

Steve Jarvis

April 21, 2013

## What I Learned...

### What's a Neural Network?

A neural network is a general machine learning tool that can be used to learn a large variety of data sets. A neural network is named so because of its inspiration: biological neurons in the brain! A conventional artificial neural network learns by making an estimate, getting feedback, and adjusting the priority given to all its connections (neurons) in such a way that it inches towards the correct answer.

The simplest neural network is one consisting of two layers. A weight connects each node in the first layer to each node in the second layer. Such a network could be trained to learn learn logical OR. Imagine three input nodes and two output, with the bottom left representing false and the bottom right true. When inputting bits representing logical OR, the network should yield output representing true (in this case, a positive value in the right output node and negative in the left output node). See Figure 1 on page 2 for an illustration.

An important consideration for this sample network is its limited learning ability. A two-layer network can only learn linearly separable functions; equations whose positive and negative results, when graphed, can be partitioned by a single line. More complex functions require deeper networks to learn, although the complexity increases quickly. With only a single extra layer (and 200 nodes per layer) the neural net used in this project was able to achieve 93 percent accuracy on the MNIST Database of Handwritten Digits[1].

### Why Is There An Extra Input Node?

The third input node is a bias. The purpose of the bias is to allow any necessary shifting of the activation function. For example, the activation function for the network used in this project is hyperbolic tangent. It was used because its domain is all real numbers, it is smooth, continuous, and symmetrical, and the

[1] http://yann.lecun.com/exdb/mnist/

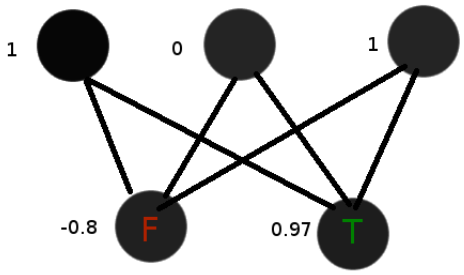


Figure 1: The top layer are inputs, connected by weights to the bottom layer. The weights are changed during training so that they give the desired output for the right input. A common implementation is to use functions with a range such that -1 lt y lt 1, so assuming the network is trained to represent true with 1 and false with -1, this would be a great output.

range is -1 to 1. It is also easily derived, which is important in training. As the weights change, the steepness of the graph is manipulated, but the y-intersect is always 0. The bias node allows shifting of the entire graph, which is the only way to train something other than a 0 output for a 0 input. For example, without a bias node, the two layer network would not even be able to learn logical AND.

## How Are the Correct Weights Calculated?

Finding the right weights is all the work. The correct weights are found via a process called back propagation. Back propagation is a repeated process of error correction, starting with the output nodes and moving back up the network. The error of the output nodes is simply the difference between the desired output and the actual output, but the error calculation for each node higher up the tree must be a summation of all collective errors of the exiting connections, since each node is connected to every node in the level lower. This algorithm proved to be the most difficult part of the project and I consulted numerous and tutorials and open source projects.[2]

The network is always training in the background so the user can see potentially constant improvement, and to learn more and different handwriting

[2]Here are some of the most helpful resources I found:
http://www.cs.montana.edu/ grayd/backprop.htm
http://arctrix.com/nas/python/bpnn.py

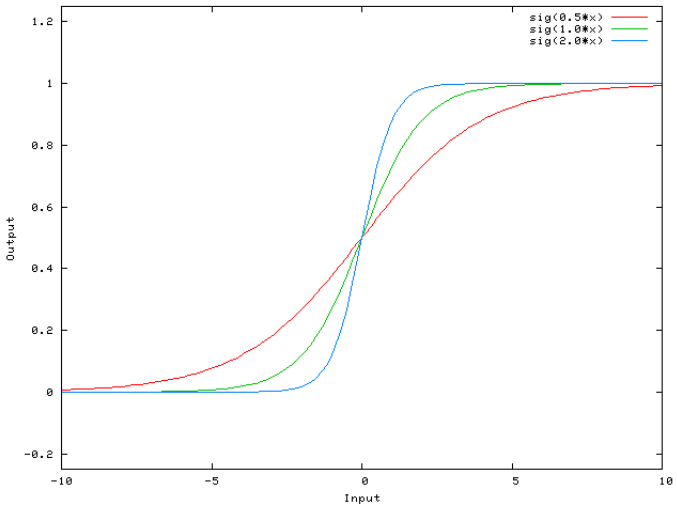


Figure 2: The graph of a general sigmoidal function as the coefficients (weights) change. Pic taken from StackOverflow: http://stackoverflow.com/questions/2480650/role-of-bias-in-neural-networks

there only needs to be more data added to the training instance on the server. Also, the network takes very long to train. At the time of this presentation, this network will have been training on euclid for about a month, and that's not unusual. Because of the dependency on adjacent layers in back propagated training, parallelization is difficult to implement successfully and efficiently[3] and is beyond the scope of this project.

Earlier it was mentioned that hyperbolic tangent is easily derived and that's nice for training. This is because we can use the derivative of the activation function in calculating the weight deltas. Consider how the derivative changes as the hyperbolic tangent is traveled. Since the derivative is greatest at the middle – where the activation is most uncertain – results in the middle will cause a greater change in the associated weights. Conversely, as the network's weights become more established through training, the derivative approaches zero and the network stabilizes.

## iOS, JSON, and CGI.

I wanted a good way to demonstrate the working neural network. Just knowing it can recognize handwritten characters in a database is not very exciting, but having it recognize a user's in real time would be. So an iOS front end was added to take input, query the server on which the network is running, and

[3]https://research.microsoft.com/apps/pubs/default.aspx?id=173312

return an ordering of likeliness, from 0 to 9, of which digit it was sent. The only particular I'd like to mention is how surprisingly easy it was get Apache to serve Python files. A single line file in the pub directory on euclid is all it takes.

```
File: .htaccess

AddHandler cgi-script .py
```

# Software Design...

## How It Works.

Each image of a handwritten digit is divided into 196 subsections. The data from the MNIST database are 28x28 pixel images, so to maintain a physical square with no extraneous pixels the image had to be represented with either 4, 16, 49, 196, or 784 sections. Based on various runs of the experiment (a special mode of network training) it was decided that 196 yielded the best balance of complexity and image granularity. See an example of the experiment's results in Figure 3. In all, the production neural network has 201 inputs, 201 hidden nodes, and 10 outputs. 196 of the inputs are a simple binary representing the presence of ink in that subsection. The additional five inputs represent the relative distribution of ink in the image – percent of ink north of the horizon, south of the horizon, west of middle, east of middle, and percent of total sections containing ink. A section contains ink if even a single pixel in that section is over a predetermined threshold.

The same algorithms are applied to translating the MNIST data to network input as translating the iOS application data to network input. When the user enters a digit on the iPhone application, the image is used to generate the binary representation described above and built into a web request to the web site. The site uses the neural network to interpret the data it received based on the best known configuration at the time, and then outputs a JSON response of the network's output. The iOS application uses that response to display the best estimate of the network and associated certainties. See Figure 4 for an illustration of the interactions. Also note that the optimal learning and momentum rate vary with the complexity of the data and size of the network.

## How It's Organized.

In designing this project I aimed to make it modular. Each logical task is the responsibility of a specific sub-application. The only exception is the experiment, which was used to establish near-optimal coefficients and explain unusual behavior at the start of the project (more on that later). The experiment piggy backs as a part of the network training application because of the significant

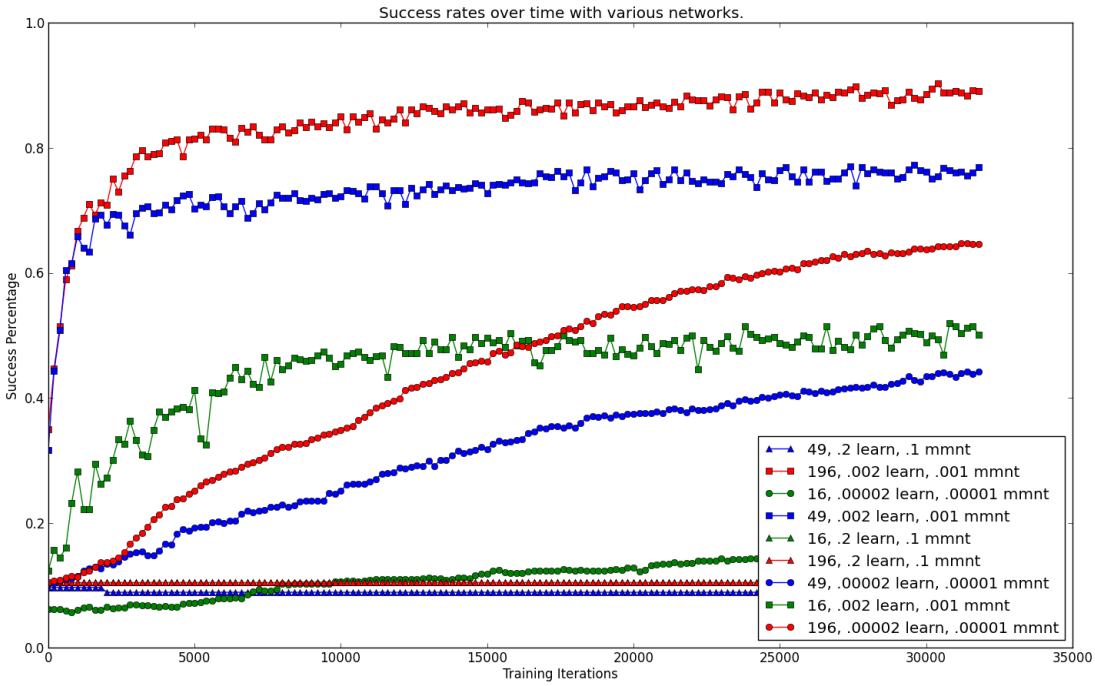


Figure 3: Example output of the experiment feature of the network training application. Runs like this helped to determine the best network size and proper coefficients for learning rates. This particular run of the experiment took 9 hours to complete, running the experiment with the network for 784 section images takes multiple days to complete. The key displays the network sizes (in neurons per layer) and coefficients used for learning and momentum rates.

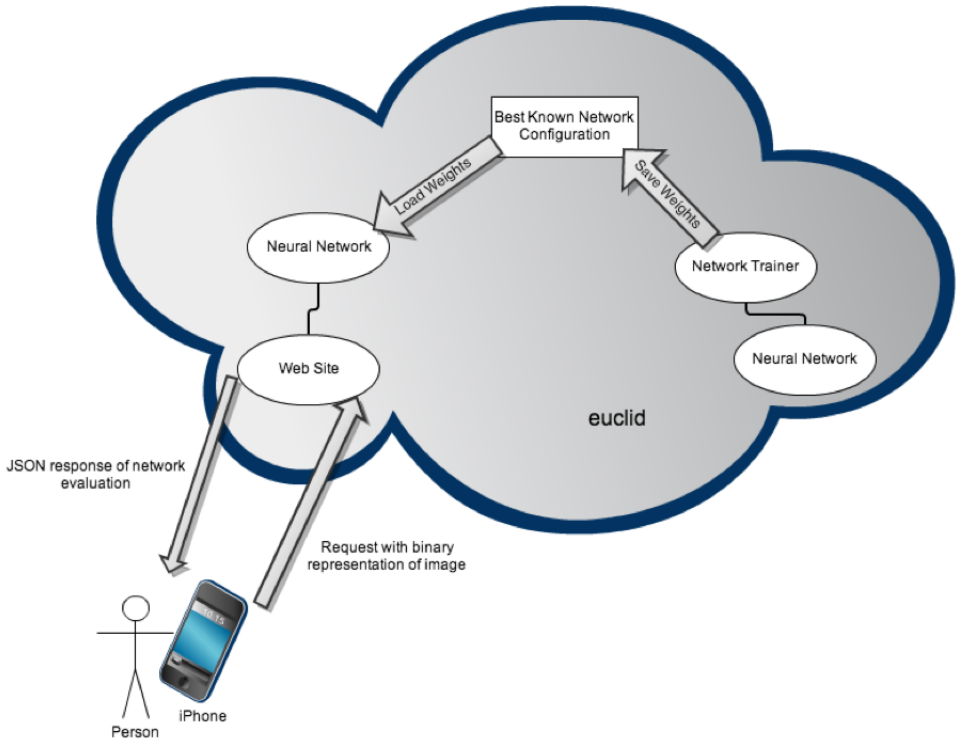


Figure 4: Interaction diagram including each piece of the entire project. Note that the process of training, which generates the best known weight configurations, is entirely distinct from the user cycle displayed on the left side. The only link is the sharing of the same weights store.

overlap in funcionality. In all, there are five sub-applications involved in the process up to this point. The live digit recognition requires the iOS application, the web site, and the neural network package. The training of the network, as well as the experiment, requires the neural network package and the network training application.

The project's parts are all under Git version control. A primary goal of this project was to build a general purpose, reusable neural network, so it exists in its own repository [4]. The other applications exist as subfolders of a repository created specifically for this project. The neural network repository is referenced as a submodule of this general repository [5].

Since there is nothing specific to handwriting recognition in the network package I like the decision to store and reference it as an indepenent package. What could use improvement is the way the network is incorporated with the rest of the project. There are multiple applications that rely on the neural network package yet it is added as a Git submodule under the network training application's directory. It would be more appropriate to have the Git submodule in a more neutral location and reference it accordingly from the other applications.

There is also substantial set up to have a completely working demonstration (web page, network training, adding the neural network package to the Python Path). It would be nice to package the suite in a way that facilitates a more automated out-of-the-box solution.

## As Hard As I Expected...

I was expecting really hard, and it's about what I got. There were a couple especially difficult hurdles as I was working on this project. When I first starting trying to train handwritten digit recognition I didn't get any performance better than about 15% successful recognition. I trained network for days on even small samples of data and it just stalled. Worse still, it was unpredictable and inconsistent. This is why I added the "experiment" mode to the network training application. I could see that "skinny" three-layer networks – networks with fewer than about 60 nodes per layer – mastered data sets quickly and flawlessly. As the networks grew larger they became wildly inconsistent. The experiment trained a constant function for a constant number of iterations on increasingly large networks and graphed the error rate and time to completion versus size. Figure 5 shows the inital results. There are so many moving pieces I couldn't imagine what the issue was. It turns out the answer was local minima, and that turning the learning and momentum rates way down would help to avoid sticking points. Figure 6 shows the nearly flawless performance of the improved network.

[4] github.com/stevejarvis/neural-network
[5] github.com/stevejarvis/scrawl-recognition

The next great challenge was improving the disappointingly poor performance of real life digit recognition. The network was training on euclid as I was finishing the basic functionality of the iOS application, and by the time I finished the logs read it was correctly recognizing more than 93% of the test data. In actual use, however, I found the network to correctly interpret only the nicest of input; perfectly sized and centered submissions. The problem was the strict preprocessing done on the data in the MNIST Database did not represent unfiltered data from the real world. To improve performance, I "messied" up the data by changing the size and rotation. The variations I added turned the training set of 40k samples into a set of 1.5M samples.

The next great obstacle was the inability of a three layer network to learn the more complex data. After swapping the training data, the learning curve became stagnant at about a 70% success rate. So I added a fourth layer. Similar to the bump from two to three layers, the change from three to four meant lower learning rates to avoid the increased chance of becoming stuck in local minima and longer training times to facilitate the exponentially increasing number of connections. Alas, it seems to be working with acceptable performance. To further increase the performance, I believe finding an appropriate bounding box per submitted sample on the iOS application would yield noticable improvements.

Here are the approximate line counts[6].

| Part | Line Count | Language |
|---|---|---|
| Neural Network | 542 | Python |
| Network Training | 1142 | Python |
| Web Site | 67 | Python |
| iOS App | 1564 | Objective C |
| Total | 3315 | All |

[6]Found by wc. iOS App is forked from GLPaint by Apple: https://developer.apple.com/library/ios/#samplecode/GLPaint/

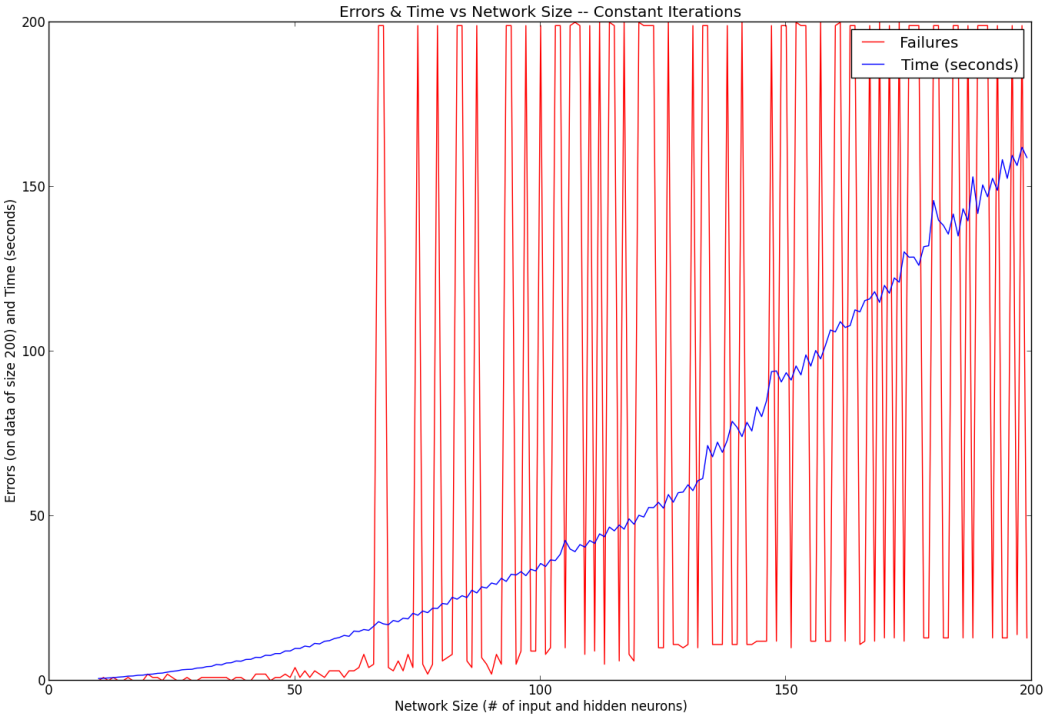


Figure 5: The experiment run with the initial learning rate. Notice that as the network grows larger the success of the network becomes as unsure as a coin flip.

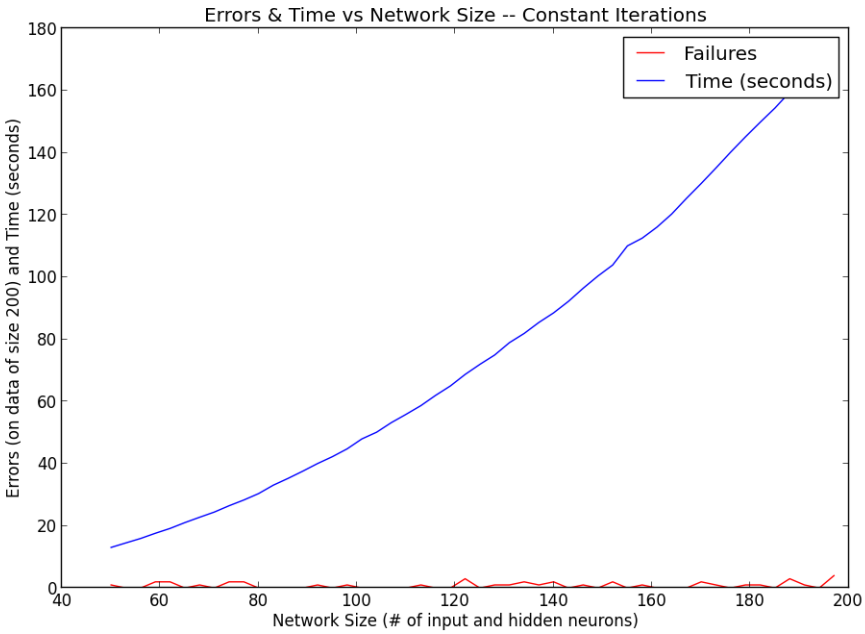


Figure 6: The experiment run with a learning rate and momentum rate 1/1000th the magnitude of that used in Figure 5