EEL5840: Elements of Machine Intelligence

Project Three: Multilayer Perceptron (MLP) – MNIST Dataset

Jason Gutel
University of Florida
Gainesville, FL
gutel@hcs.ufl.edu

***Abstract*—This work focuses on using multilayer perceptrons in analyzing the MNIST digit dataset. To achieve the best results possible multiple configurations of MLPs are trained and evaluated using a training and validation sub-set of the data respectively. The best MLP configuration is the one that achieves the best results with the validation set. This MLP configuration is then used to classify the test set of data and the results are then analyzed and used to comment on the effectiveness of the testing method and the configuration used for classification.**

***Index Terms*—Multilayer Perceptron, MNIST Digit Data Set, digit recognition, classification, artificial neural network**

I. Introduction

In 1943 Warren McCulloch and Walter Pitts, a neurophysiologist and an engineer, created a model called an artificial neuron. An artificial neuron is a model based off *simple neurons* which are binary (on or off) based off some threshold of aggregated input data. This model is known as the McCulloch-Pitts (MCP) Neuron.

This simple MCP Neuron was capable of simple logic functions such as *A and B* as well as *A or B* [1]. However, it was the work of Rosenblatt who perceived of using multiple augmented MCP neurons for learning. Rosenblatt based his system off Hebb's Rule, which states [2]:

> *"When an axon of cell A is near enough to excite a cell B and repeatedly or persistently takes part in firing it, some growth process or metabolic change takes place in one or both cells such that A's efficiency, as one of the cells firing B, is increased." –Donald Hebb*

Rosenblatt's new artificial neuron was called the *perceptron* and uses weighted inputs to the perceptrons which are trained over iterations with a training set of data. The perceptron still exhibits a non-linear binary output of 0 or 1 however when an output of 0 is exhibited but it should have been a 1 based off the training input data the weights are adjusted to receive the desired outcome.

A visual representation of the perceptron is shown in figure 1. In figure 1 the inputs to the perceptron are X1, X2, X3 each with a weight (W1, W2, W3). These weighted inputs are summed together and a non-linear activation function $f(\sum_{i=1}^{n} W_i X_i)$ outputs an on or off signal (1,0) or (1,-1) if the input is greater than or less than some threshold. This on or off signal is the output Y.

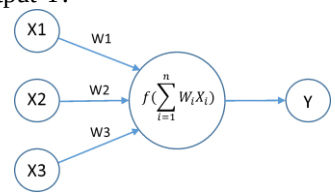

Figure 1: The Perceptron Model [3]

To train the perceptron, the following steps are used [4]:

1. Start the perceptron with random weights
2. Compute the perceptron's output with some input training sample
3. If the perceptron's output doesn't match the desired output
    a. Output should have been 0 but was 1 → decrease weights that had an input of 1
    b. Output should have been 1 but was 0 → increase weights that had an input of 1
4. Go to next sample and repeat 1-3; continue until no more mistakes are made in training data

Perceptrons can solve a linearly separable data set however when a more complicated relationship exists than a purely one dimensional linear separation the perceptron fails to achieve classification. Figure 2 shows such a difference by evaluating the logical *XOR* operator. In the image a clear separation can be made to solve the *AND, OR* functions but one cannot be made for the *XOR* function.

The solution to the *XOR* problem was found by two researchers, Minsky and Papert whom suggested that to compute the *XOR* function one perceptron was not up to the task. In fact, it would take multiple interconnected perceptrons [5].

Figure 2: Linear classification of *AND, OR* and the failure of classifying *XOR* for perceptrons [4]

A. Multilayer Perceptrons

Figure 3 is an example of a multilayer perceptron (MLP) also known as an artificial neural network (ANN). By using an ANN, the input data is sent to an arbitrarily large number of intermediate neurons within the *hidden layers*. The benefit of doing this is that the hidden layer perceptrons can act on very specific features of the input data set. By training and aggregating all the features that are important to classification these are all fed to the final output layer which performs the actual classification.

Previously with a single perceptron the entire input data set is treated as raw data and the relationships had to be found implicitly by what was visible in the raw data. With multilayer networks the data can be more nuanced and specific features such as shapes, colors, etc. (in the context of an image) can be acted upon individually.

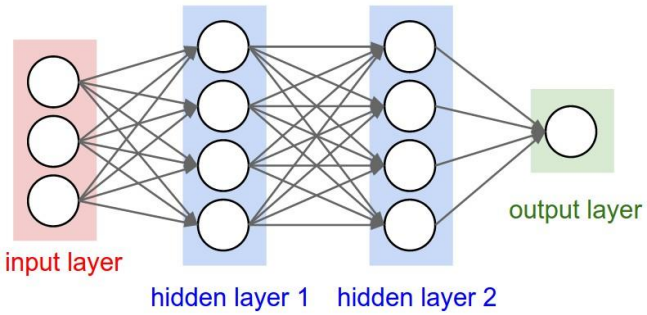

Figure 3: Artificial Neural Network [6]

When using ANNs an issue arises with how to properly train them. Previously the algorithm developed by Rosenblatt is developed to iteratively train a perceptron based on the output layer alone. With an ANN, each perceptron in the network needs to be trained with respect to its contribution to the output classification.

ANNs have two main events that happen during training; feed-forward and backpropagation as shown in figure 4. In this figure the feed-forward (or just forward) arrow represents passing training data (a face in this example) through the network. The system then classifies the image and compares it to the true label (dog v. human face); the error is then fed back through the network to update the weights of the system called *backpropagation*.

This is done for the entire training set multiple times per training set (known as epochs) until some criteria has been met – often a maximum number of epochs or the error becomes static or moves less than some threshold.

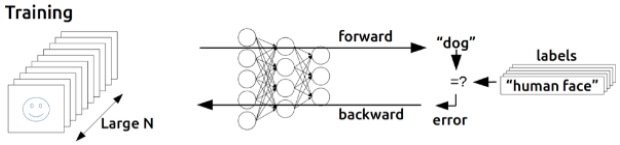

Figure 4: Feed-Forward and Backpropagation in an ANN [7]

B. Backpropagation Algorithm

In order to perform the backpropagation weight adjustments, a criterion for the error must first be established. The original method, and the one evaluated in this paper, is the stochastic gradient descent (SGD) criterion using the Mean Squared Error. The Mean Squared Error (in general) is defined as:

$$MSE = J(w) = \frac{1}{2N}\sum_{k}(d_k - y_k)^2$$

Where d_k is the desired output and y_k is the observed output. However, since y_k is the output of the activation function the MSE for a perceptron is defined as:

$$\frac{1}{2N}\sum_{k}\left(d_k - f\left(\sum_{j} w_{kj} f(net_j) + b_k\right)\right)^2$$

Where w_{kj} is the weight from the hidden layer to the output, net_j is the net input at layer j, b_k is a bias input to the neuron, and f is the activation function. By using the chain rule from calculus it can be shown that the change in weight, Δw_{kj} can be expressed by:

$$\Delta w_{kj} = -\eta \frac{\partial E}{\partial y_k}\frac{\partial y_k}{\partial net_k}\frac{\partial net_k}{\partial w_{kj}}$$

Where η is the learning rate and the equation for the changing weight is taken from the Least Means Square rule [8] variation.

To find the derivative of the activation function (found within y_k) the activation function must be a continuous real function. The previous activation function used was the Heaviside function, which is not differentiable around 0. So in MLPs a different activation function is used; two of the most used are the sigmoid and arctan functions.

These functions approach ±1 in the arctan case or 0,1 with sigmoid when saturated but are nearly linear around zero. When the net to these functions are within the bounds of (-1,1), or (0,1) respectively, they affect the outcome the most and when they are in the deeply saturated regions they affect the outcome the least.

This paper will be using the rectifier activation function. This function leads to faster training on deep ANN architectures and is defined as:

$$f(x) = \max(0, x)$$

By evaluating each of these derivatives Δw_{kj} can now be written as:

$$\Delta w_{kj} = \eta(d_k - y_k)f'(net_j)net_j$$

By letting, $\delta_k = (d_k - y_k)f'(net_k)$, then evaluating the weight change for the input to hidden layer (Δw_{ij}) the error used in the hidden layer is necessary to determine the current error and the equation is adjusted to be:

$$\Delta w_{ji} \propto \left[\frac{\partial E}{\partial y_k}\frac{\partial y_k}{\partial net_k}\frac{\partial net_k}{\partial y_j}\right]\frac{\partial y_j}{\partial net_j}\frac{\partial net_j}{\partial w_{ji}}$$

Which evaluates to:

$$\Delta w_{ji} = \eta\left[\sum_{k}\delta_k w_{kj}\right]f'(net_j)net_i$$

This is known as the delta rule. The delta values represent the backpropagation of errors leading to the current layer's weights. By using the delta rule the new weights can be found by (for the hidden and output layers, respectively):

$$w_{kj} = w_{kj} + \Delta w_{kj}$$

and

$$w_{ji} = w_{ji} + \Delta w_{ji}$$

With this after every feed-forward phase the network can be adjusted based on the results to approach a desired outcome. One addition to the backpropagation algorithm that can lead to better convergence is to use a momentum term.

The idea behind momentum is to add a fraction of the previous term to the current term. This in effect increases the step size when the gradient is approaching a minimum. If the minimum is not a global, but a local, minima then the momentum will potentially help step *out* of the local minima and prevent the solution from becoming stuck in it. The new weight update with momentum (μ) becomes [9]:

$$\Delta w_{kj}(n) = \eta \delta_k net_j + \mu \Delta w_{kj}(n-1)$$

II. Problem and Methodology

The MNIST dataset is a database of 70,000 handwritten digits with a near equal distribution between 0 and 9. In this project a MLP was trained by analyzing many possible configurations to achieve the best results in classifying the MNIST data.

The 70,000 digits in the dataset have two main sections; the first 60,000 which are distributed between 0-9 (ordered) and are typically used for training and the latter 10,000 digits distributed between 0-9 which are used for validation and/or testing.

To create unique validation and testing data the last 10,000 digits of the MNIST set were randomly shuffled and then split into two groups of 5,000 digits each. The data distribution is shown in table one and a histogram of the validation and testing data is shown in figure 5. It can be seen from figure 5 that the data is nearly uniformly distributed between 0-9.

Python was used to perform all the analysis and computations in this project, with the sklearn MLPClassifier method in particular to create the neural networks [10]. This was used to optimize the work and decrease computational time allowing for more configurations to be analyzed.

Data Set	Number of Data Points in Set
Training	60k
Validation	5k
Testing	5k

Table 1: Distribution of MNIST data

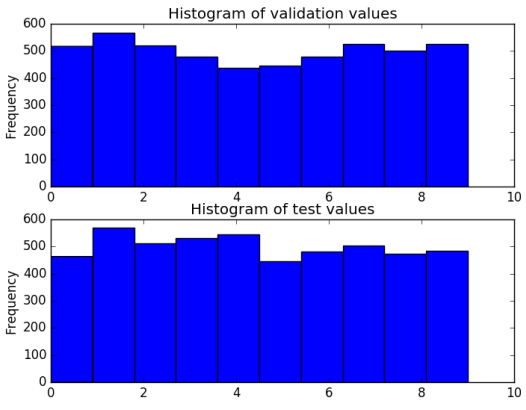

Figure 5: Histogram of validation and testing data

A. Training

There were four different variations of initial MLP configurations that were used. For each of these configurations ten different training runs were conducted while varying the number of processing elements (PEs/perceptrons) per hidden layer from 10 to 100 in multiples of 10. For each training session, there were 350 epochs used. There was also a provided tolerance level of 0.0001 to break the training if an improvement of that tolerance isn't made after two consecutive iterations.

The four configurations used were:

1. Single hidden layer with varying number of PEs, rectifier activation function, momentum factor = 0, tolerance level of 0.0001, learning rate = 0.1
2. Single hidden layer with varying number of PEs, rectifier activation function, Momentum factor = 0.9, tolerance level of 0.0001, learning rate = 0.1
3. Double hidden layer with varying number of PEs, rectifier activation function, momentum factor = 0, tolerance level of 0.0001, learning rate = 0.1
4. Double hidden layer with varying number of PEs, rectifier activation function, Momentum factor = 0.9, tolerance level of 0.0001, learning rate = 0.1

After running each of these configurations 10 times (for each quantity of PEs) the input data was down sampled by using a decimating FIR filter and the process was repeated. This occurred five times for decimation levels from 1-5x.

There were then (10 PE densities x 5 decimation levels) 50 runs conducted during training for EACH model resulting in 200 total individual models.

B. Validating Models

Each of the training models were then used to classify both the training data used as well as the validation set. The model (out of 200 candidates) that did the best job of predicting the validation set was chosen as the best model for classification.

This selected model was then fine-tuned by re-training with the same training set over a new range of learning rates. The model was re-trained for learning rates from 0.1 – 1 and from 0.01 – 0.1 if 0.1 was the best learning rate from the previous results.

C. Classification

The winning model from the previous two sections was then used to classify the final, unique test data. After classification was completed the true labels and the predicted labels for the test data were used to generate a confusion matrix and to observe the accuracy of the final model.

A confusion matrix is a matrix where the column values represent the true label and the rows represent the predicted label.

III. Results

A. Training

During training plots were made for each of the configuration types over each decimation level. Figures 6 and 7 show the configuration for the double hidden layer perceptron

with a decimation of 3x on the training data without and with momentum respectively. These images omit the 10 PE run since it had a significantly larger starting error and was an outlier.

From figures 6 and 7 the neural network with momentum produces a lower error on the training data as well as converges to a solution faster for every level of PE density. This implies that the networks without momentum tend to get stuck in some local minima and take longer to reach the tolerance level (if they ever do).

The networks with momentum reach any minimum faster as they increase their weight with respect to the negative gradient. With all the *momentum* that the network gathers it is then able to escape local minima of the error surface and keep searching; leading to a lower overall error.

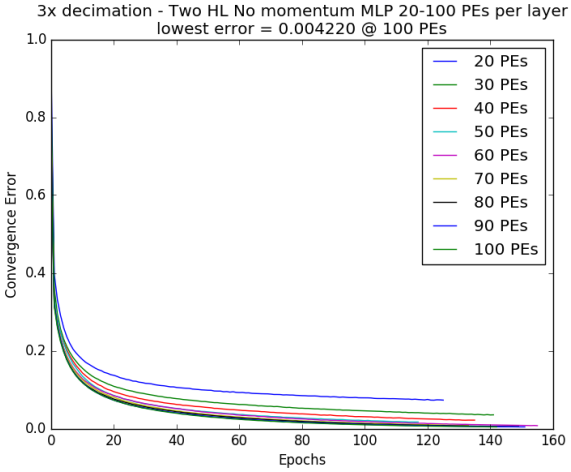

Figure 6: Training rate of double hidden layer ANN with 3x decimation and no momentum

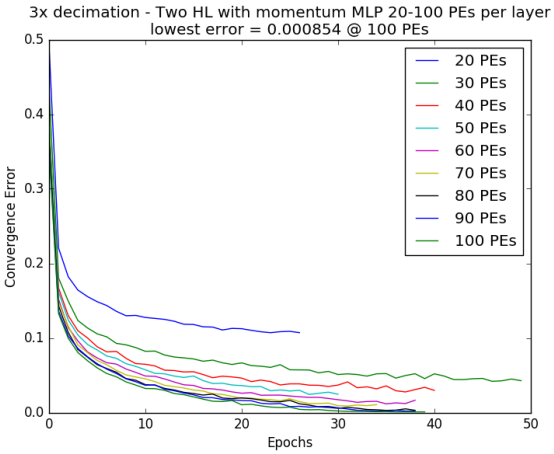

Figure 7: Training rate of double hidden layer ANN with 3x decimation and with momentum

Similarly, figures 8 and 9 show the same configuration (and results) for the single hidden layer neural network this time for the 1x (or no) decimation training data. In these figures, the non-decimated, single hidden layer ANN performs in much the same manner; momentum leads to better and quicker convergence levels.

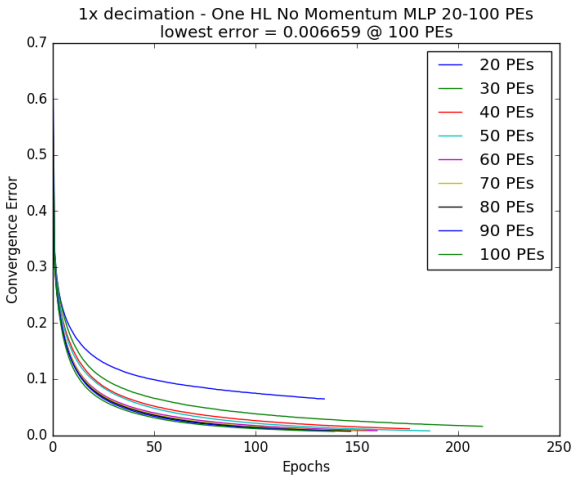

Figure 8: Training rate of single hidden layer ANN with 1x decimation and no momentum

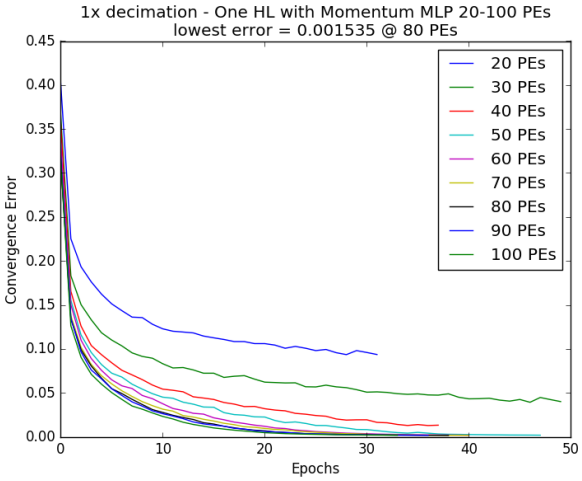

Figure 9: Training rate of single hidden layer ANN with 1x decimation and with momentum

For the 1x and 2x decimation level models about half of the configurations could score with 100% accuracy the training data as shown in figure 10. While the 3x, 4x, and 5x models failed to achieve perfect convergence for any of their configurations. Figure 11 is an example of this with the 5x decimation model.

With higher decimation levels the accuracy on the training data was decreased implying that the non-decimating would either give better performance overall or it was overfitting the training data. To find out which was the case the models were then used to fit the validation data set.

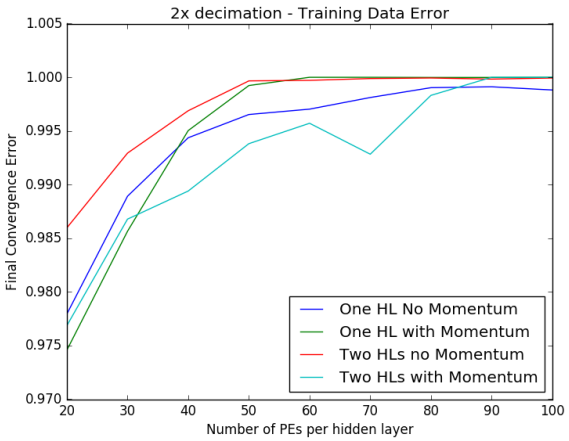

Figure 10: Accuracy on training data of 2x decimation model

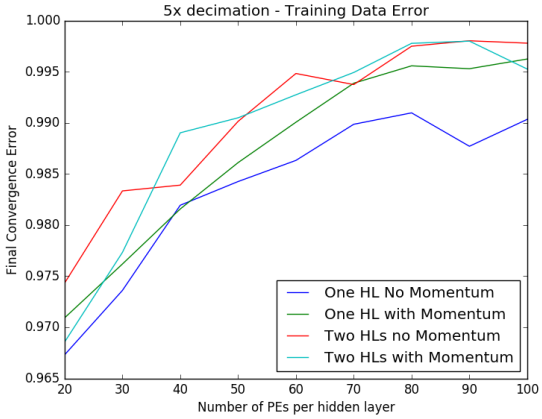

Figure 11: Accuracy on training data of 5x decimation model

B. Validation of the Models

The ANNs were then used to score the validation data to find the best configuration out of those tested. Figure 12 shows the accuracy of the same model as figure 10 on the validation data. By comparing figures 10 and 12 the scoring of the validation data is more chaotic than the typically increasing accuracy observed on the training data as the number of PEs are increased. This is to be expected since the models have never seen this data.

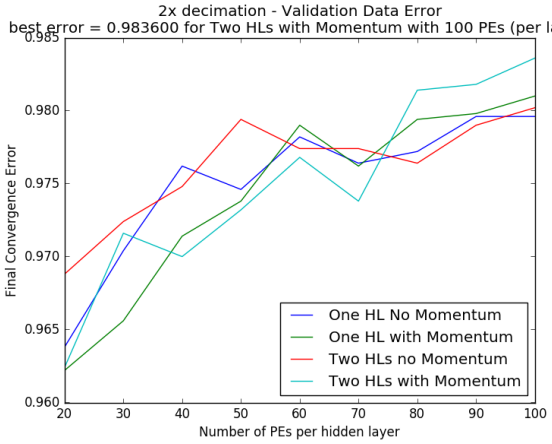

Figure 12: The 2x decimated model's error on the validation data

Figure 13 shows the best results per level of decimation against the validation data set. For every level of decimation, the best result occurs with momentum (which is not a surprise based on previous data). The oddball occurs with the non-decimated data showing that the best configuration occurs with only a single hidden layer while the others all show double hidden layers performing best.

A tie was also seen for the best result on the validation data set between the 2x and 3x decimated data sets. To break the tie the code considered 3 metrics:

1. Consider higher decimation better
2. Consider less PEs better
3. Consider the smallest training time

The winner in this case was the one that won on all three metrics being the 3x decimation level with momentum. The normalized training times for each model type is shown in figure 14 which shows an expected monotonically decreasing shape.

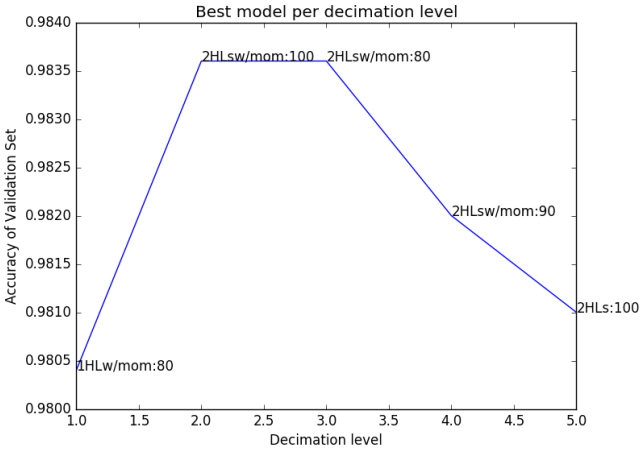

Figure 13: Best results of all models per decimation level on the validation set

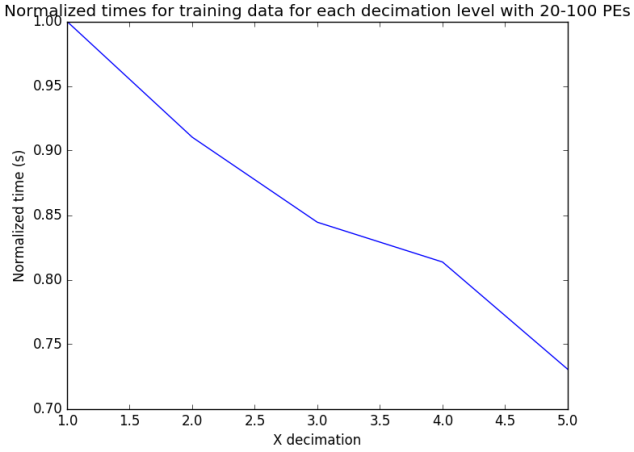

Figure 14: Normalized training time for each

Once the optimal model was selected amongst the 200 candidates the learning rate was fine-tuned on this model. First the learning rate was ranged from 0.1 → 1. When 0.1 was shown to be the best out of that range it was again varied over the range 0.01 → 0.1 as shown in figure 15.

Figure 15: Reverifying the models accuracy on the validation set by tweaking the learning rate

From figure 15 it was observed that the optimal learning rate for this model was at 0.07. From the validating data set the best model was the one shown in table 2.

Parameter	Value
Number of Hidden Layers	2
Number of PEs per hidden layer	80
Momentum Factor	0.9
Learning Rate	0.07

Table 2: The configuration of the optimal model

C. Classification

The training of this specific model was captured in figure 16. It was seen that the model achieved convergence on the training data after 43 epochs with a recognition rate of 99.89%. When performed on the validation set the model achieved an accuracy of 98%.

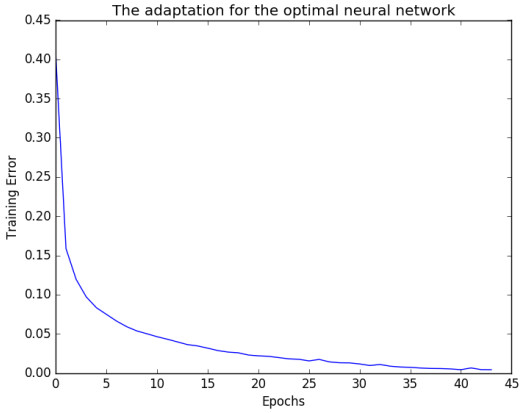

Figure 16: Recognition rate for the optimal network

The optimal model was then used to classify the 5k values in the testing set of data where it achieved an accuracy of 97.74%. In figure 17 the confusion matrix of the classification shows an extremely well-tuned model. From the confusion matrix, it was observed that the model did a very good job of identifying the digits indicated by the dense (dark colored) elements occurring along the main diagonal.

Figure 18 shows a histogram of the misclassified digits. From figure 18 it was seen that the most misclassified digit was *8* and the runner ups were 9 and 4 while the best scoring digits were 0 and 1. Out of 5,000 total digits 113 of them were misclassified.

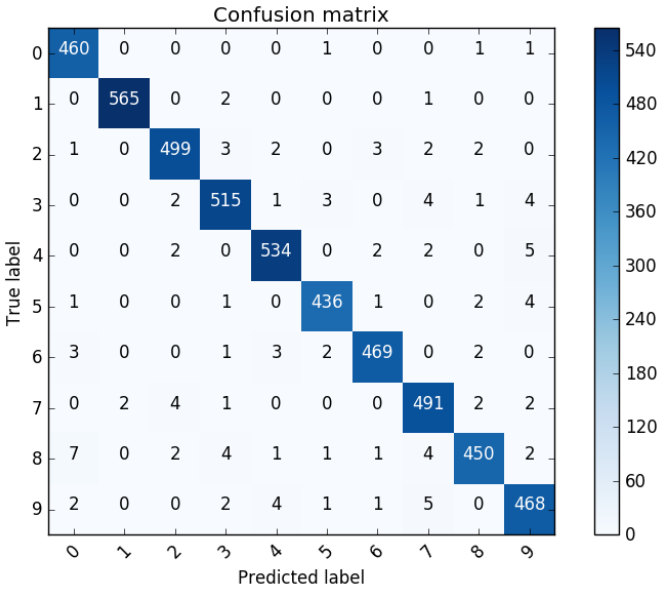

Figure 17: Confusion matrix of the final model on the testing set; color intensity refers to the density of that matrix element

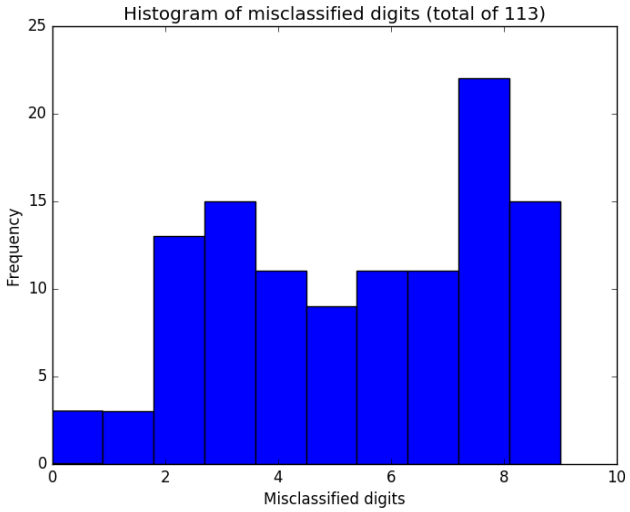

Figure 18: Histogram of the misclassified digits

Figure 19 shows a sample set of some of the misclassified digits. By observing figure 19 it can be easy to see how the system may have misclassified these digits; some of which (notably 2 and 6) even a human observer may have a hard time classifying.

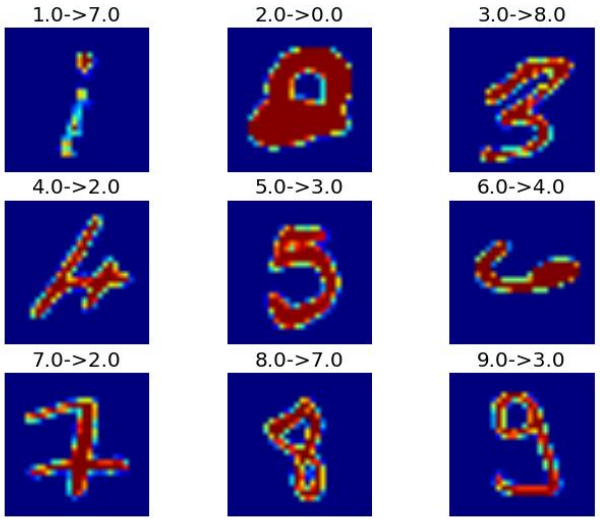

Figure 19: Sample of misclassified digits, read as (true value → classification)

IV. Conclusions

The training and validation methods of the network could be improved by a multiple of factors. These involve more sophisticated, time consuming, and/or redundant methods.

In this work the momentum was kept at a static 0.9 but can feasibly take on any value between (0,1). Tuning the momentum in the same fashion that the learning rate was tuned may increase the accuracy of the model.

Figure 13 which compared the best model per decimation level saw an equal amount of classification on the validation data with both the 2x decimation with 100PEs as the 3x decimation with 80 PEs in the double hidden layer. Since the modeling was only done for a maximum of 100 PEs it is possible that by increasing the number of PEs available to the network's hidden layers the 2x decimation (or others) may outperform the chosen best in class.

Decimation may also be replaced by a more nuanced method of feature reduction such as Principal Component Analysis (PCA) which seeks to maximize the variance between the data. The dimensions with the least variance could be removed overall dropping the feature size rather than naïvely filtering out some of the data.

Finally using a different activation function than the rectifier function such as the sigmoid or arctan function may increase the accuracy of the models at the cost of computational time.

The total computational time can also be reduced by using more powerful computers including outsourcing the work to GPUs. This would allow for more models to be analyzed.

Overall, it has been shown that using ANNs on the MNIST digit dataset for classification performs extremely well on about 97% of the data with the current implementation.

The method of choosing the model used in the classification was shown to be successful in weeding out less accurate models by splitting the test data into a test and validation dataset. If no validation was done then the best model would have been one of the non-decimated earlier models which was seen to be overfitting to the training data. This would have led to a worse classification.

The code and results for this project (including many graphs not included in this report) can be found at https://github.com/gutelfuldead/MLP_NMIST_Project.

V. References

[1] C. Stergiou and D. Siganos, "Neural Networks," Imperial College London, [Online]. Available: https://www.doc.ic.ac.uk/~nd/surprise_96/journal/vol4/cs11/report.html#Appendix A - Historical background in detail. [Accessed 9 December 2016].

[2] D. Hebb, The Organization of Behavior, New York: Wiley & Sons, 1949.

[3] Lincoln and Nicholas, "Identifying Subatomic Particles with Neural Networks," October 2015. [Online]. Available: http://2centsapiece.blogspot.com/2015/10/identifying-subatomic-particles-with.html. [Accessed 10 December 2016].

[4] A. Kurenkov, "A 'Brief' History of Neural Nets and Deep Learning, Part 1," 24 December 2015. [Online]. Available: http://www.andreykurenkov.com/writing/a-brief-history-of-neural-nets-and-deep-learning/. [Accessed 9 December 2016].

[5] M. Minsky and S. Papert, Perceptrons: an introduction to computational geometry, Cambridge: MIT Press, 1969.

[6] M. Nielsen, "CS231n Convolutional Neural Networks for Visual Recognition," [Online]. Available: http://cs231n.github.io/neural-networks-1/#nn. [Accessed 10 December 2016].

[7] NVIDIA, "Inference: The Next Step in GPU-Accelerated Deep Learning," NVIDIA, 11 November 2015. [Online]. Available: https://devblogs.nvidia.com/parallelforall/inference-next-step-gpu-accelerated-deep-learning/. [Accessed 10 December 2016].

[8] J. Principe, "Backpropagation Learning Algorithm," University of Florida, Gainesville, FL, 2016.

[9] G. Orr, "Momentum and Learning Rate Adaption," Willamette University, 1999. [Online]. Available: https://www.willamette.edu/~gorr/classes/cs449/momrate.html. [Accessed 10 December 2016].

[10] scikit learn, "sklearn.neural_network.MLPClassifier," scikit learn, 2016. [Online]. Available: http://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPClassifier.html. [Accessed 1 December 2016].