CSE 190 – HW 2 (Team Assignment)

Xiuyuan Chen
Student
xic145@ucsd.edu

Nikhilesh Sankaranarayanan
Student
nsankara@ucsd.edu

(a) Abstract

This paper describes the implementation of a neural network designed to read handwritten digits. The network uses the MNIST dataset as training input, and trains itself on 50,000 different characters.

The resulting network recognizes digits pulled from the remaining 10,000 in the MNIST data set. After 40-50 iterations on the training set, the network correctly recognizes nearly 90% of the digits from the test data set.

(b) Introduction

In this paper, we tackle the problem of a computer recognizing handwritten digits. To do this, we have trained a neural network to recognize handwritten digits with a high degree of certainity. The following sections detail the implementation and variations of this network.

(c) Methods

This neural network was implemented in MATLAB 2016 and uses Batch Gradient Descent learning to train itself to recognize handwritten digits. It uses a multi-level architecture, with a hidden layer between the input and output layer, consisting of 128 nodes. The network trains itself on 50,000 characters from the MNIST data set and tests itself on 10,000 more.

Variations of this network were explored and have been detailed. These include changing the architecture, the activation function, parameters such as learning rate and number of iterations, and adjustments to the learning rule such as regularization and momentum effects.

(d) Results

Problem 3

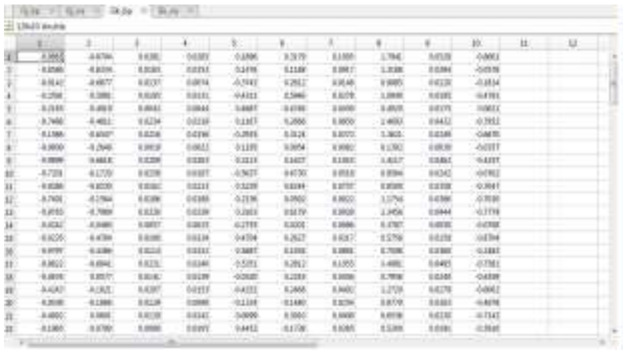
a.) The MNIST dataset was loaded in using the provided helper functions.

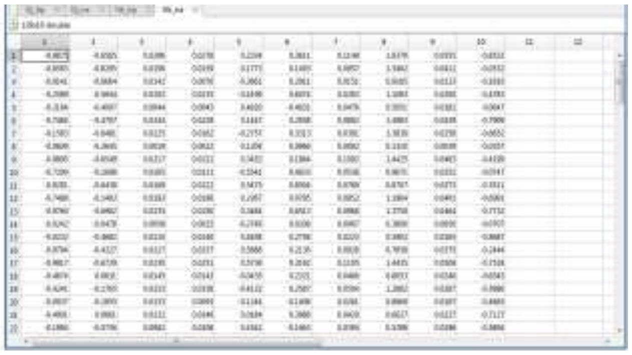
b.) Using the Numerical Approximation method yielded results nearly identical to the gradients from the Gradient Descent method. Screenshots of the gradient matrices are shown below Biases are included in the matrices.

SUBSET 1:

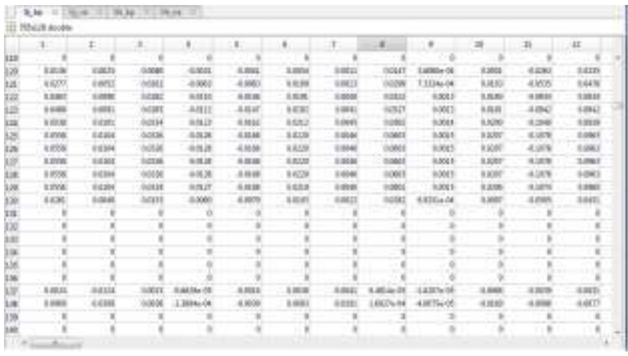
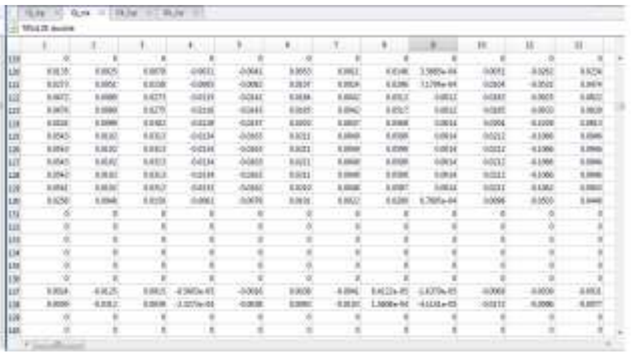
5 units input to the input layer, 128 nodes in the hidden layer

Hidden Layer – Backward Propagation vs Numerical Approximation


[illegible]


[illegible]

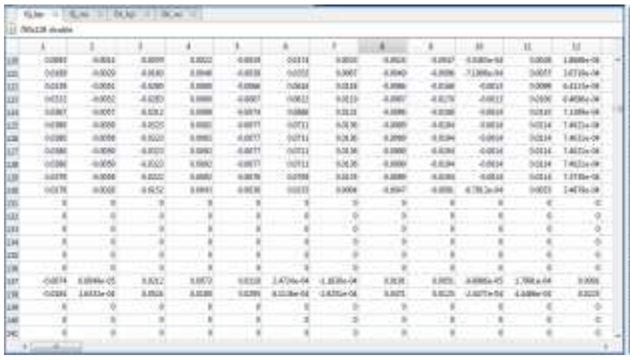
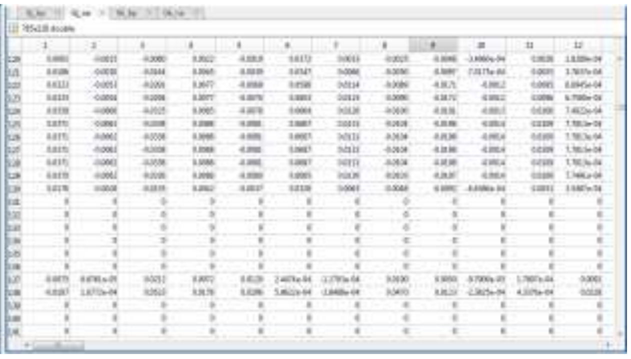
Output Layer - Backward Propagation vs Numerical Approximation

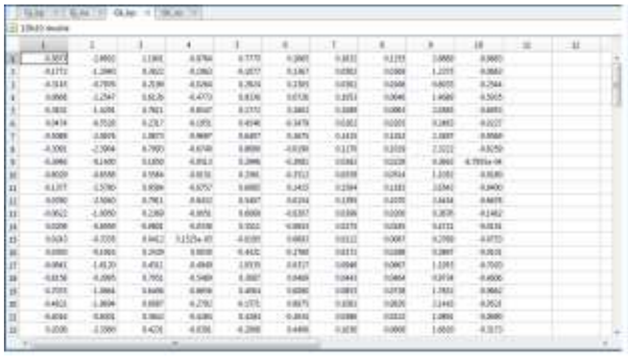
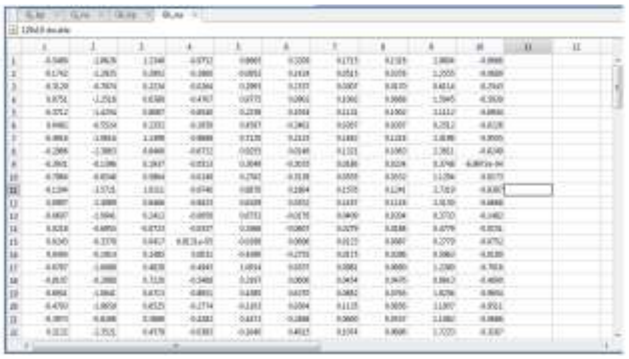
<u>SUBSET 2</u>:

10 units input to the input layer, 128 nodes in the hidden layer

<u>Hidden Layer</u> – Backward Propagation vs Numerical Approximation

<u>Output Layer</u> - Backward Propagation vs Numerical Approximation

As can be seen, **all matching weights are within 10^{-4} of each other.** This trend continues for larger subsets of weights also.

c.) We used cross-validation to determine the stopping criteria of the network. The training set was the first 50,000 input images and the holdout set was the last 10,000.

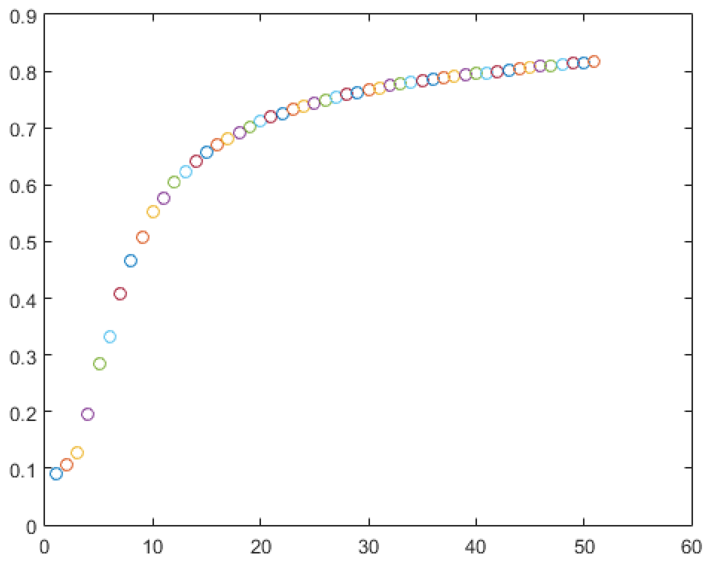
The criteria for stopping the learning was as follows:

- No more than 5 consecutive errors on the validation set
- Once the loss percent reached 10%
- Once there are 0 errors
- When `maxTrial` parameter (set to 50 for all graphs) is exceeded

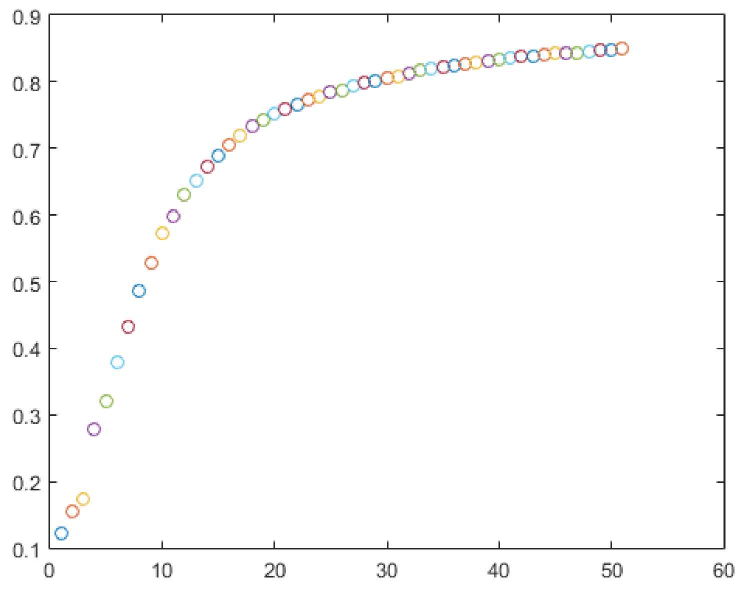
For a learning rate of 0.0001 the accuracy plot only rises with epochs before saturation at around 90% accuracy on both test and training data, and after slightly more than 50 epochs the network stops.

The plots of Accuracy and Loss vs Iterations, for both Training and Test data is shown:

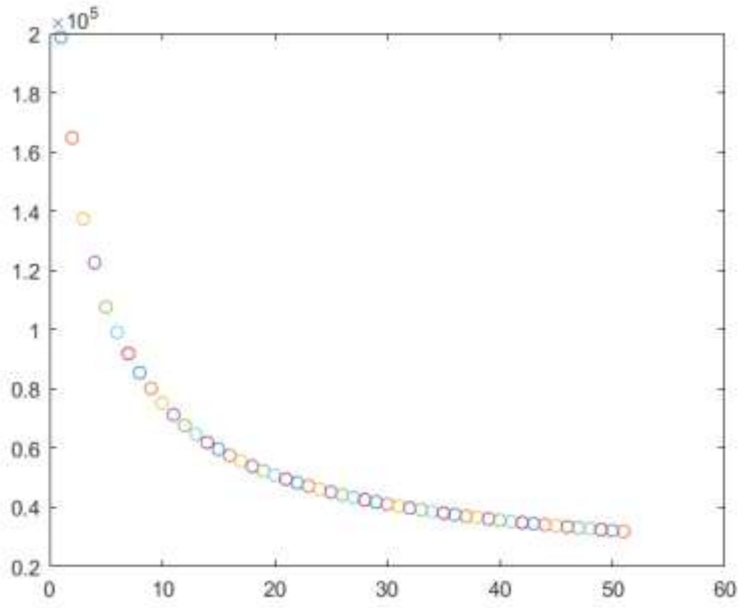
Accuracy (Y) vs Iterations (X) on Training Data:



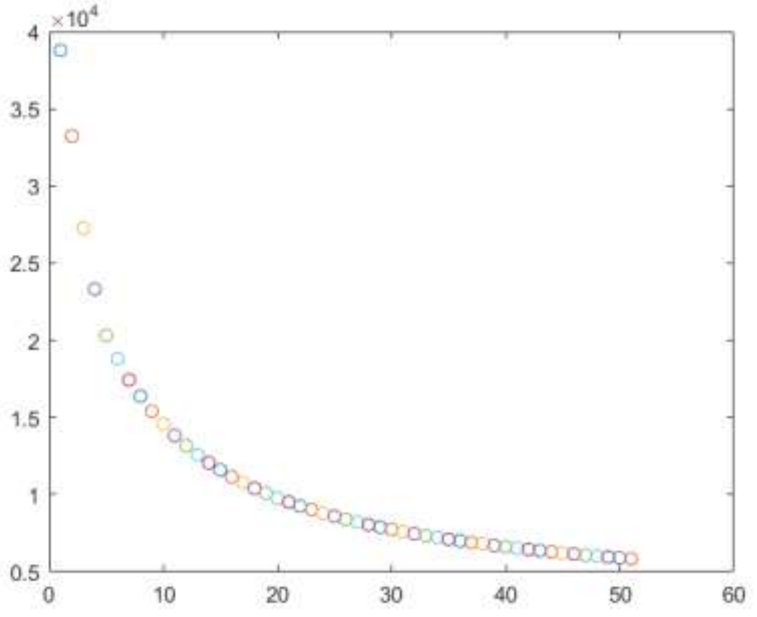
Accuracy (Y) vs Iterations (X) on Test Data:



Loss (Y) vs Iterations (X) on Training Data:



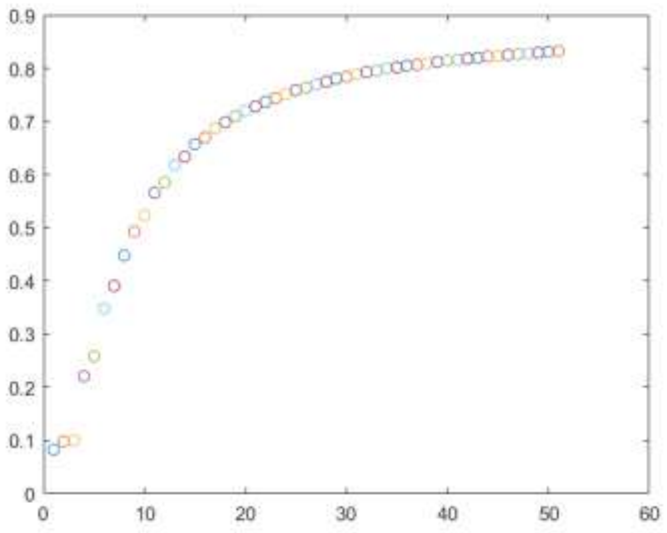
Loss (Y) vs Iterations (X) on Test Data:



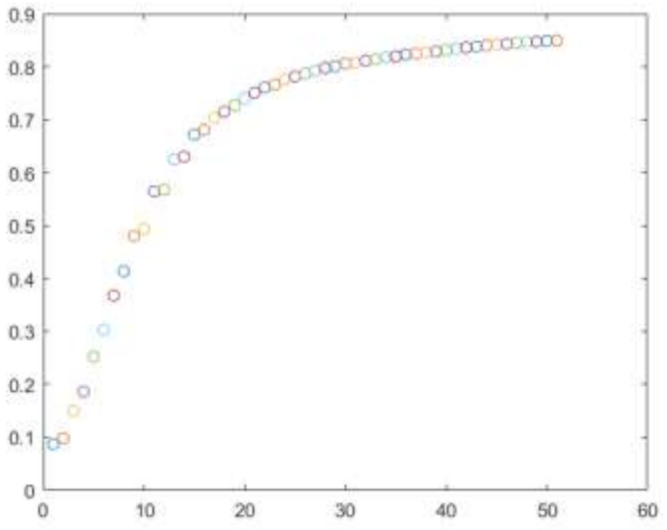
d.) Both L2 and L1 Regularizations were implemented, with a Lamba of 0.001. The plots for L2 regularization are shown below.

Comments on performance: With regularization added, the Loss initially tended to be far higher than in part (c), by nearly $2x10^4$ misclassifications. However, it also converged more quickly. It reached around 90% accuracy around 5 epochs earlier than without regularization.

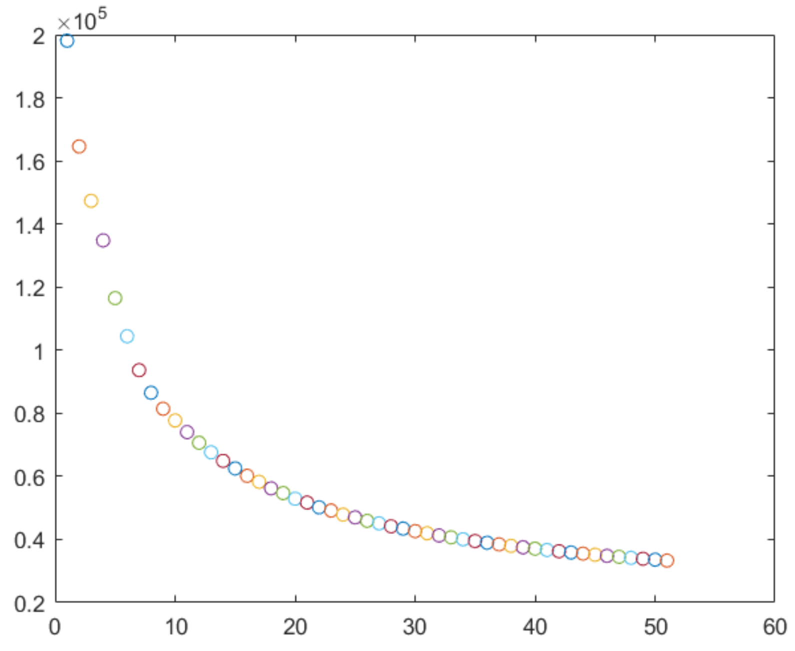
Accuracy (Y) vs Iterations (X) on Training Data:



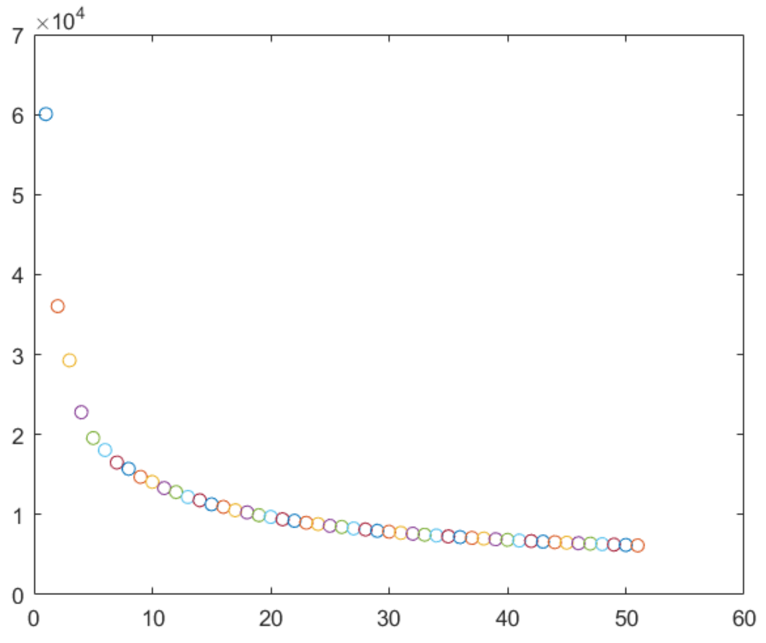
Accuracy (Y) vs Iterations (X) on Test Data:



Loss (Y) vs Iterations (X) on Training Data:

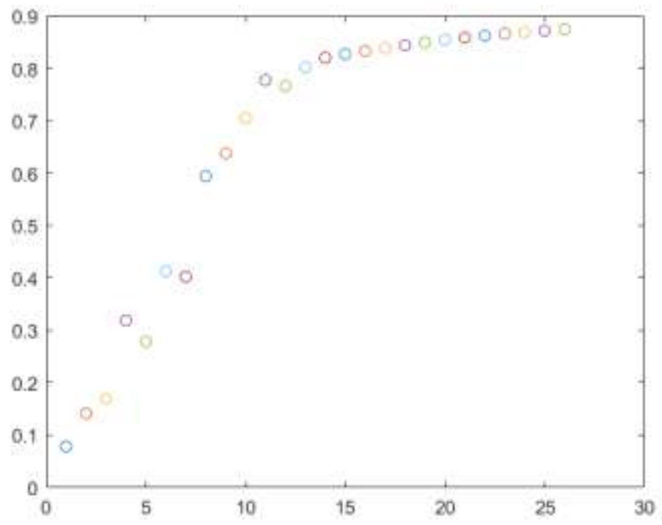


Loss (Y) vs Iterations (X) on Test Data – Notice the far higher starting error

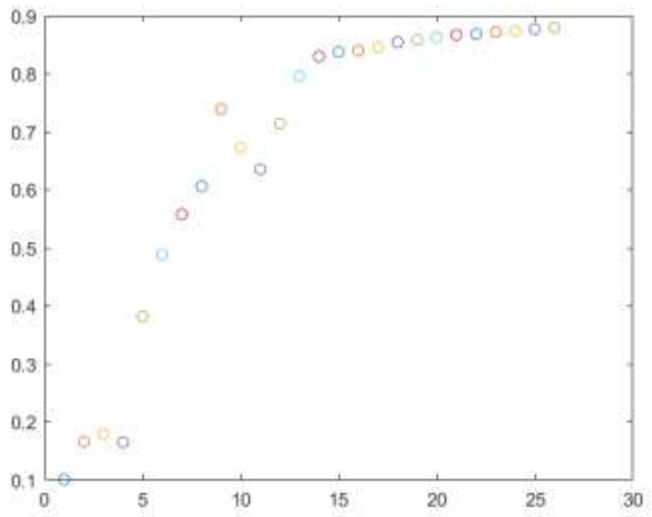


e.) We applied Netsorev momentum to the neural network, so we first move in the same direction as the last time we changed the weight and then solve the gradient at that point, and apply corrections.

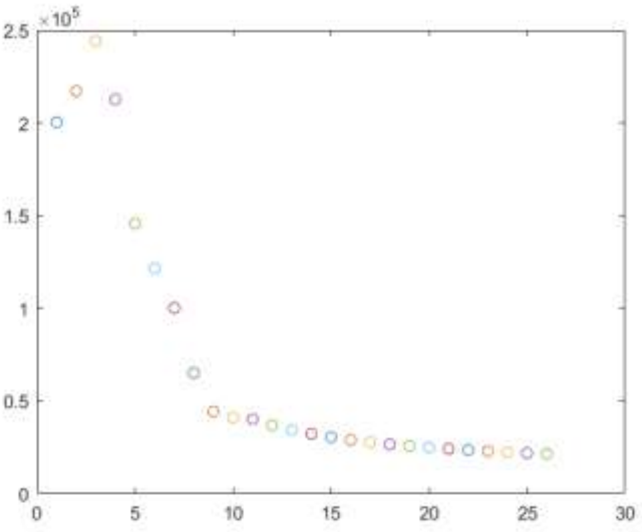
Accuracy (Y) vs Iterations (X) on Training Data



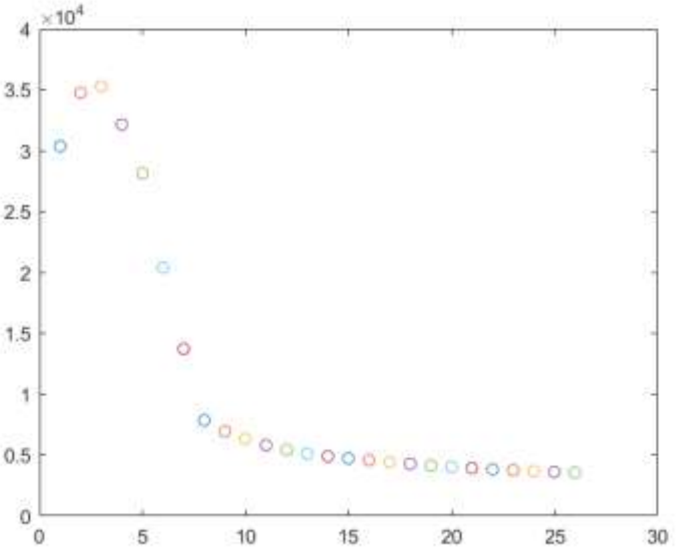
Accuracy (Y) vs Iterations (X) on Test Data



Loss (Y) vs Iterations (X) on Training Data



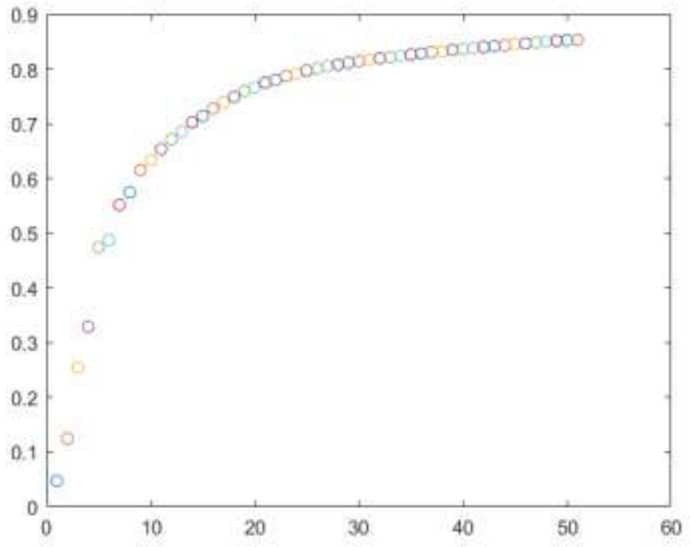
Loss (Y) vs Iterations (X) on Test Data



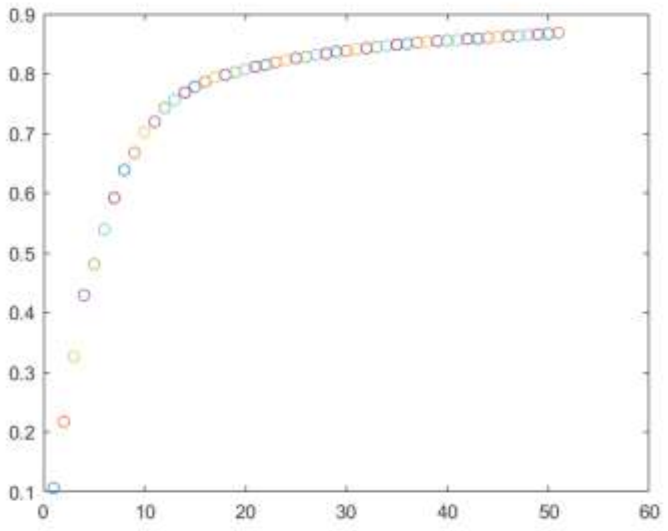
f.) The tanh and reLU functions slightly improve the speed of convergence. The reLU function needed a slower learning rate (0.000005) to avoid oscillations and maintain a consistent direction. The computation time was also significantly lower compared to sigmoid and tanh. This is probably due to the simplicity of the reLU function.

<u>**tanh function:**</u>

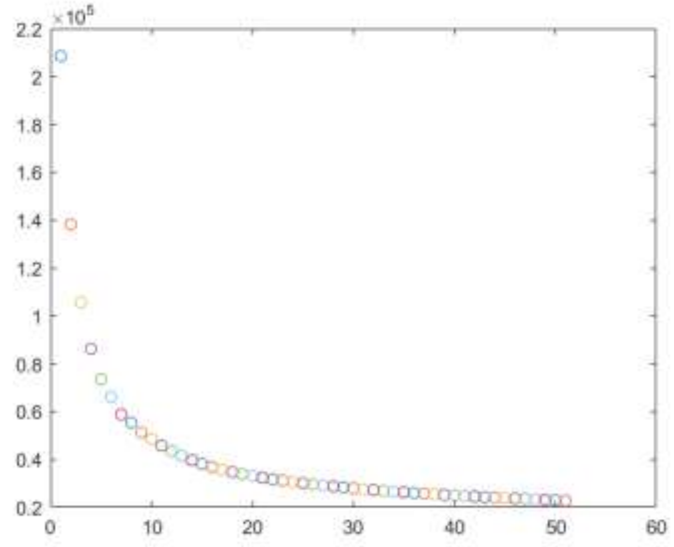
Accuracy (Y) vs Iterations (X) on Training Data



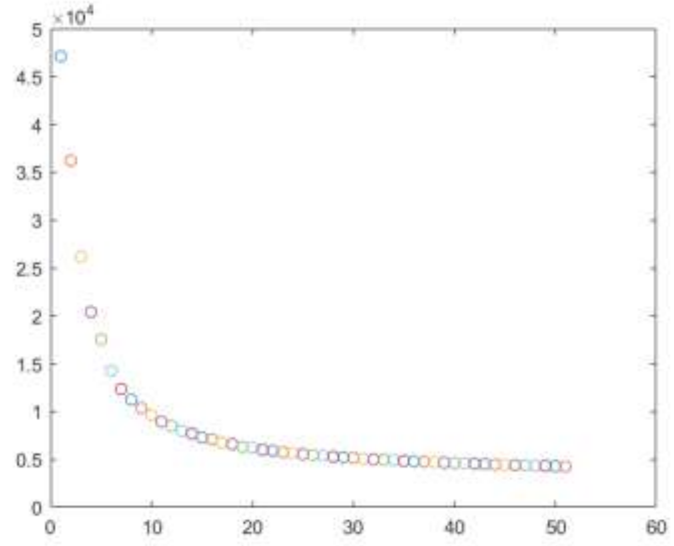
Accuracy (Y) vs Iterations (X) on Test Data



Loss (Y) vs Iterations (X) on Training Data

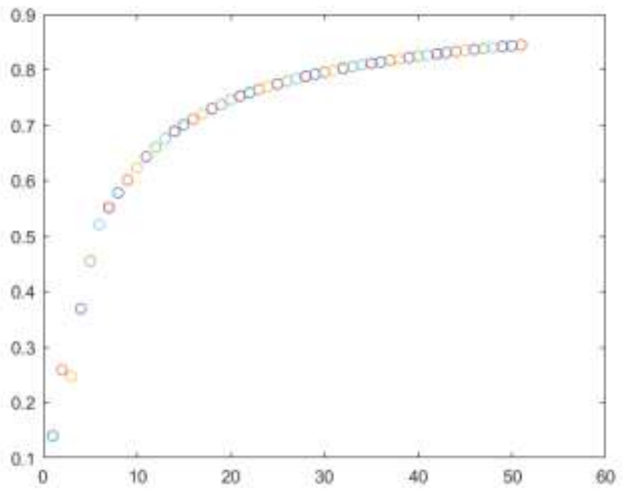


Loss (Y) vs Iterations (X) on Test Data

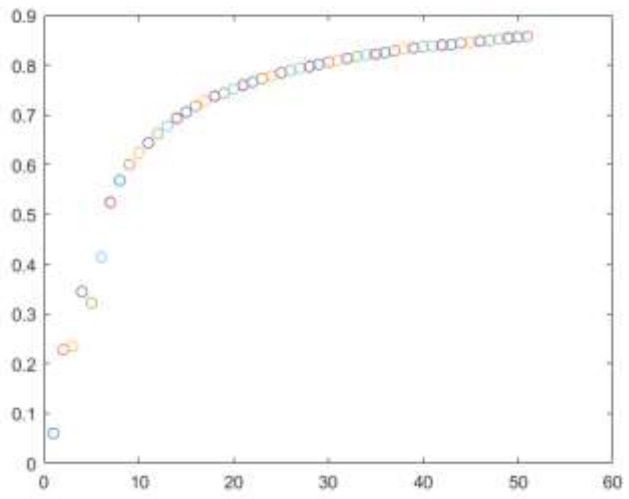


reLU function:

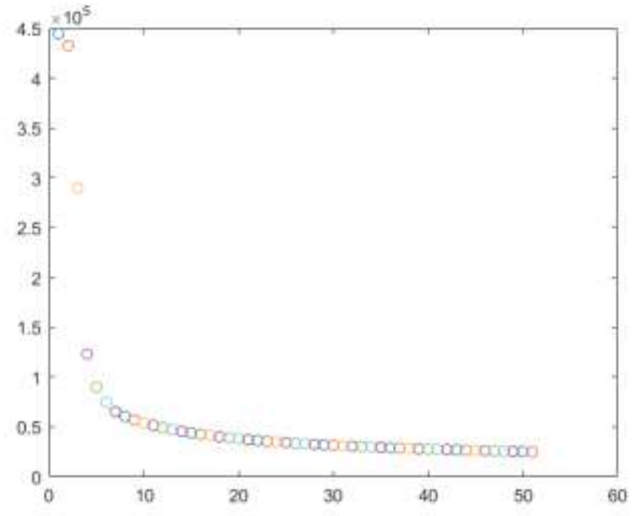
Accuracy (Y) vs Iterations (X) on Training Data



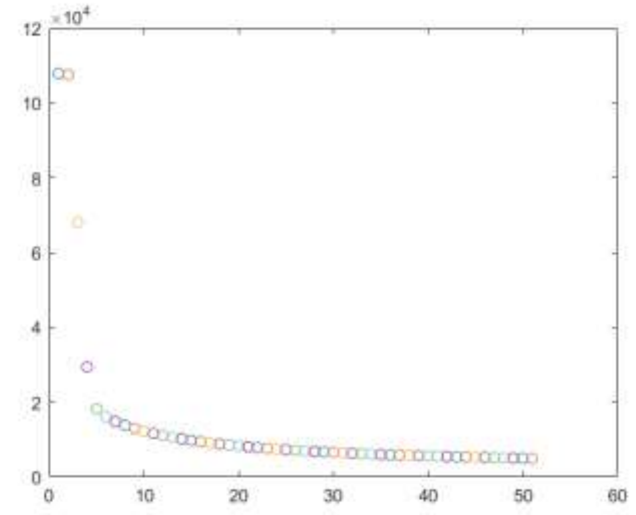
Accuracy (Y) vs Iterations (X) on Test Data



Loss (Y) vs Iterations (X) on Training Data



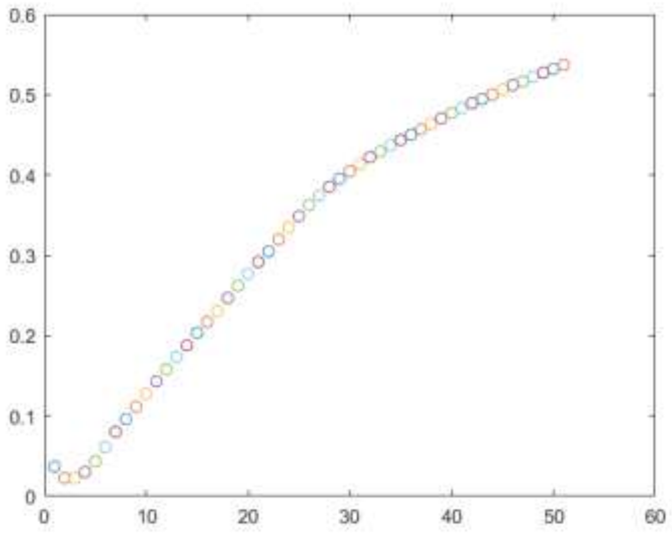
Loss (Y) vs Iterations (X) on Test Data



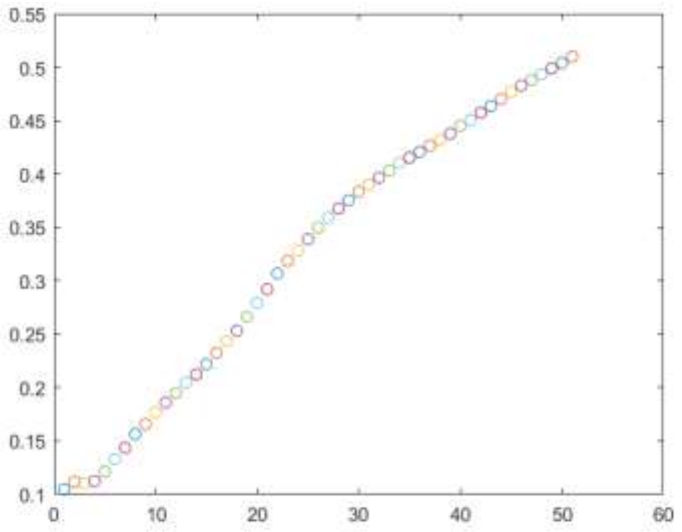
---

g.) Changing the number of hidden nodes to 64 (halved from 128) did slightly lower the final test data accuracy by about 10%. Also, the program ran a little bit faster. However, cutting hidden nodes down to **10** produced a more noticeable change in the plots, as pictured (the program did run really fast though). The maximum accuracy reached is only around 50%, and there's a lot more fluctuation in the error in the early iterations.

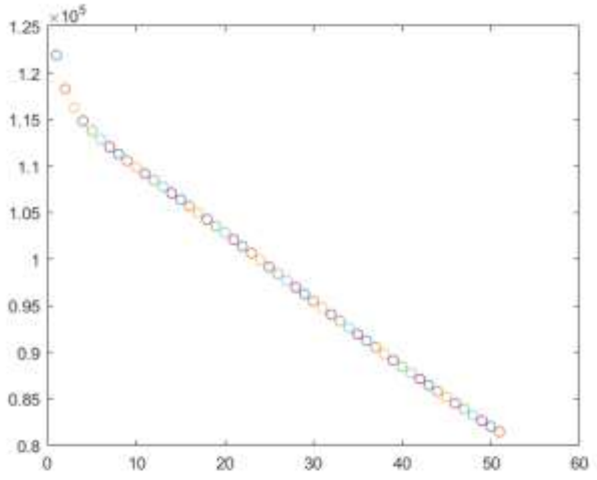
Accuracy (Y) vs Iterations (X) on Training Data with **10 hidden nodes**:



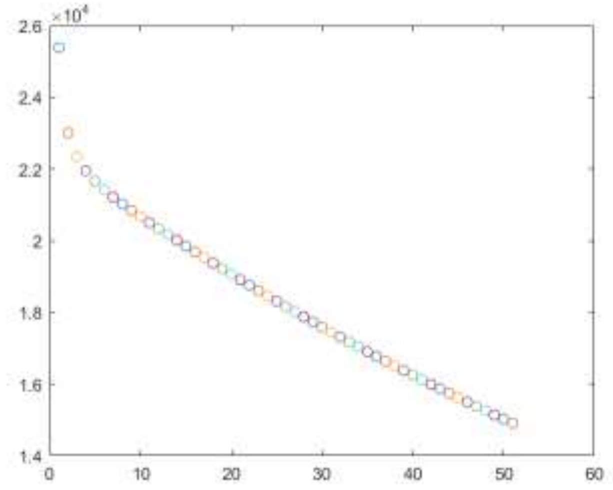
Accuracy (Y) vs Iterations (X) on Test Data with **10 hidden nodes**:



Loss (Y) vs Iterations (X) on Training Data **with 10 hidden nodes:**

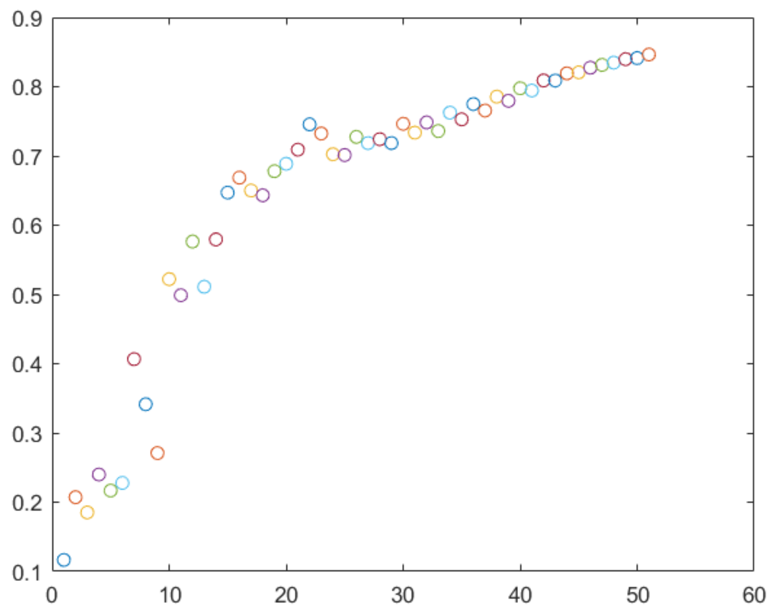


Loss (Y) vs Iterations (X) on Test Data **with 10 hidden nodes:**

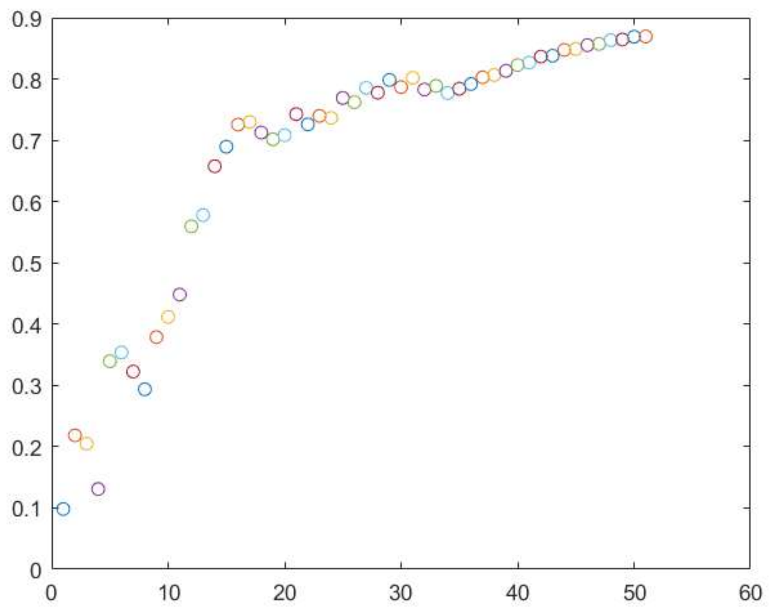


Now the number is increased to **400 hidden units**. This network now achieves the same 90% accuracy as 128 hidden units given enough epochs, but the initial few iterations experience much larger fluctuations and back-and-forth swings. These many hidden nodes is probably "too many". Unlike other extensions of the network this one actually increases in errors before eventually converging.

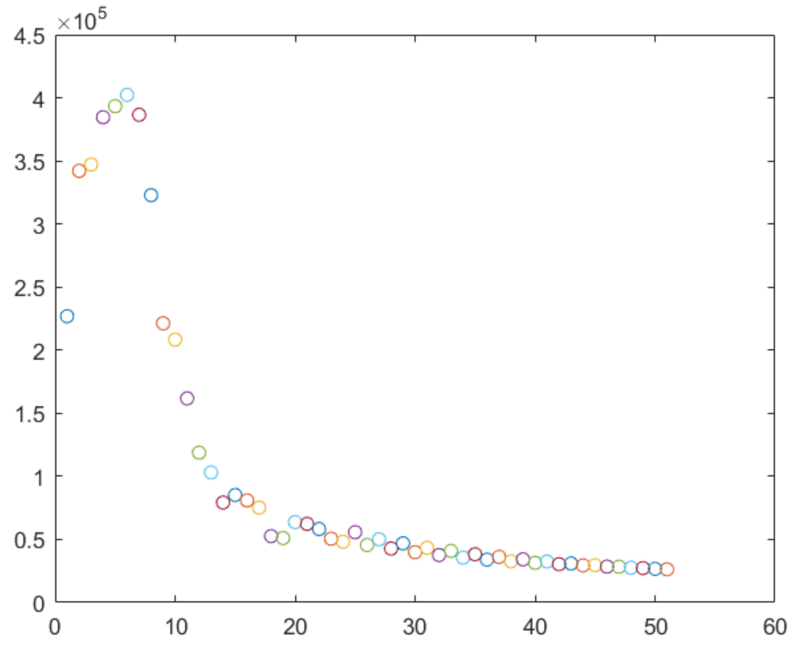
Accuracy (Y) vs Iterations (X) on Training Data with **400 hidden nodes**:



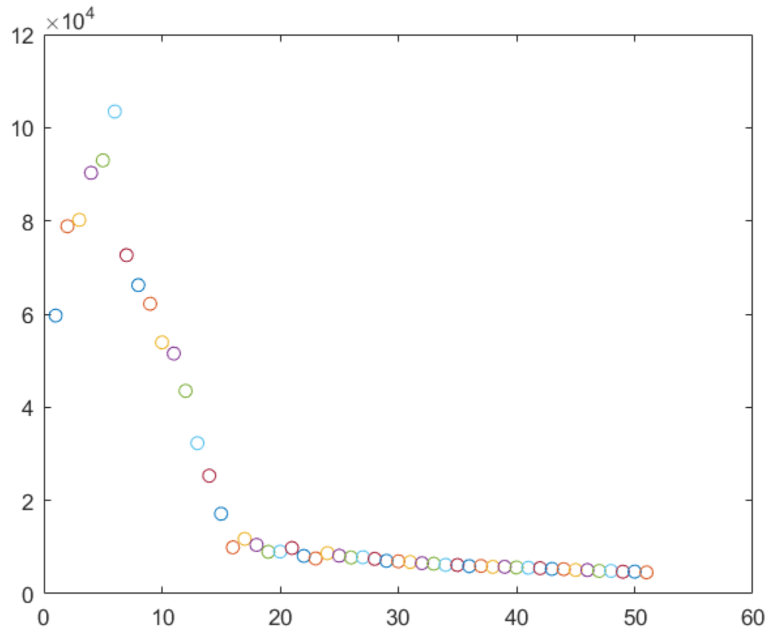
Accuracy (Y) vs Iterations (X) on Test Data with **400 hidden nodes**:



Loss (Y) vs Iterations (X) on Training Data **with 400 hidden nodes:**



Loss (Y) vs Iterations (X) on Test Data **with 400 hidden nodes:**



h.) To visualize the weights, we took the 784x129 long W_{ij} weight matrix, took a random column from it and reshaped it into a 28x28 matrix. Then, the `imshow()` function produced a grayscale image with each value of weights scaled between 0 and 255.

This was the image corresponding to the 6th row (therefore representing the number 6). As with other variations, learning rate = 0.00001.

The faintest white outline of a "6" shape can be made out if you squint really hard. It's an interesting image, but clearly the neural network can make more sense of it than our eyes can.

Here's another image of the 4th row.

With a superhuman level of mental gymnastics, one can almost clearly make out the 4 in the image:

Problem 4

Using the reLU activation function with no other variations added, the network was run on the notMNIST dataset, conveniently packaged at

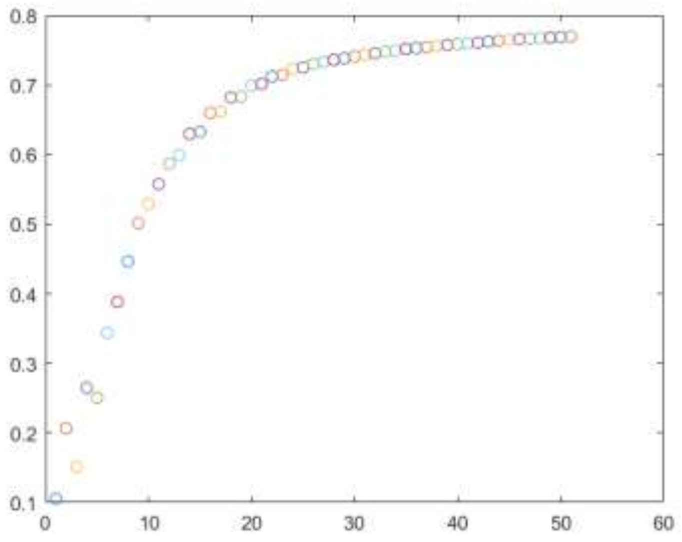
https://github.com/davidflanagan/notMNIST-to-MNIST

For this test learning rate was set to 0.000005. Nothing else but the input path to the data files was changed. As with the MNIST data set, we use cross-validation to test our network – the first 50,000 rows are used as training data, tested against the last 10,000.

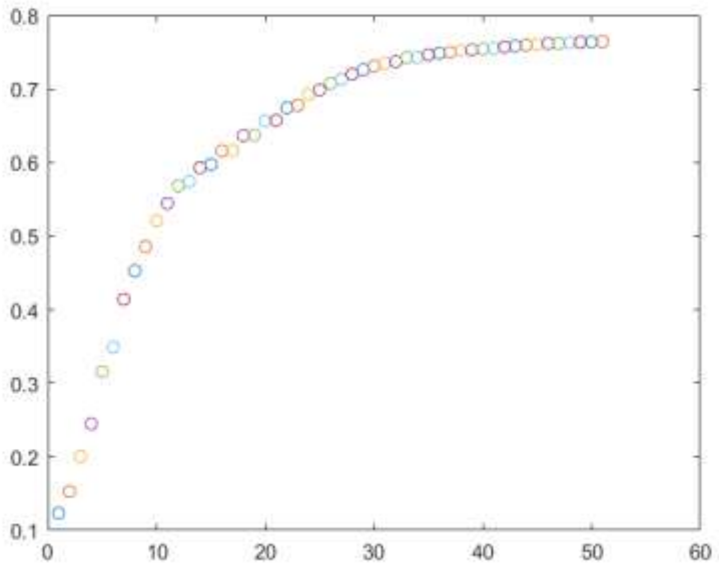
The network performed quite well, but was **significantly slower** than the reLU version on the MNIST dataset., probably because its slightly larger.

This dataset seems a bit harder as after 50 iterations the network still had a 23.65% error rate. The graphs of accuracy and loss vs iterations are plotted:

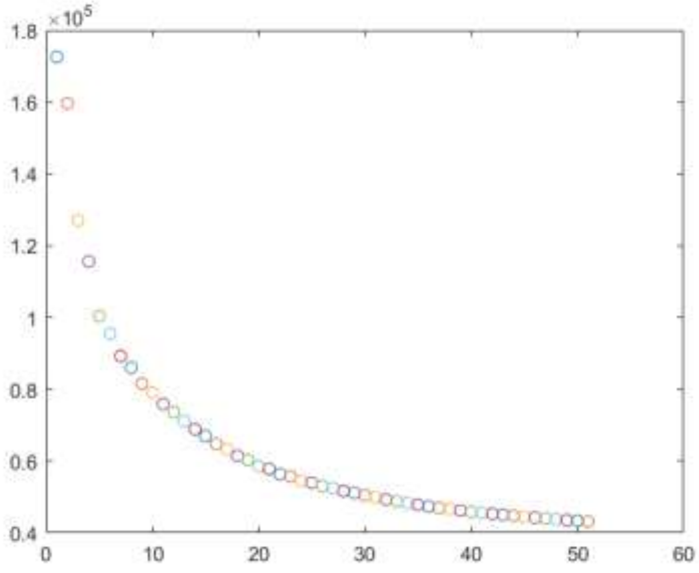
Accuracy (Y) vs Iterations (X) on Training Data:



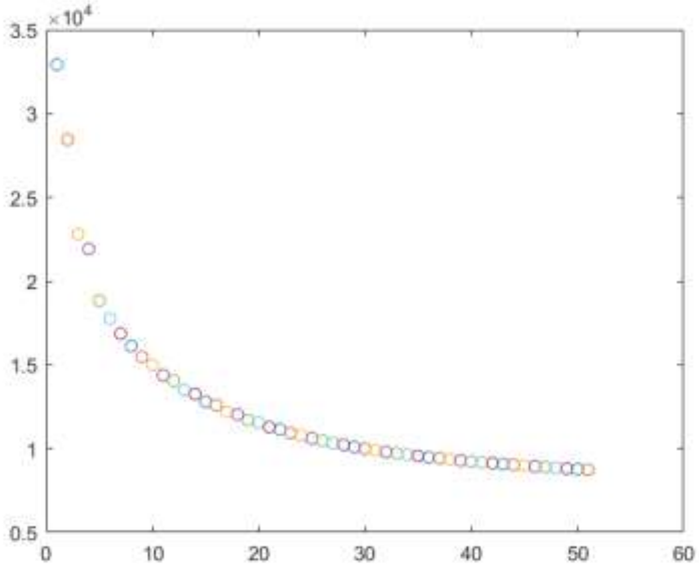
Accuracy (Y) vs Iterations (X) on Test Data:



Loss (Y) vs Iterations (X) on Training Data



Loss (Y) vs Iterations (X) on Test Data



Team Member Contributions

The team programming part was completely pair-programmed, with both members present at the station coding. For problem 3, most of the “driver” coding was by Xiuyuan (specifically parts c,d,e) Xiuyuan wrote the code to generate plots in MATLAB, and the error checking code and stopping criteria.

Nikhilesh coded parts (f,g,h) and loaded the notMNIST files for problem 4. Nikhilesh also wrote up the report and produced the accuracy/loss plots. This includes writing the code for weights visualization and producing it.

Since both members were together for the whole duration, we feel that this was a well balanced team effort.