Blake Mallory

Professor Minai

Intelligent Systems

12 November 2019

Intelligent Systems: Backpropagation

**Problem 1: Using a Multi-Layer Perceptron to Classify Numbers**

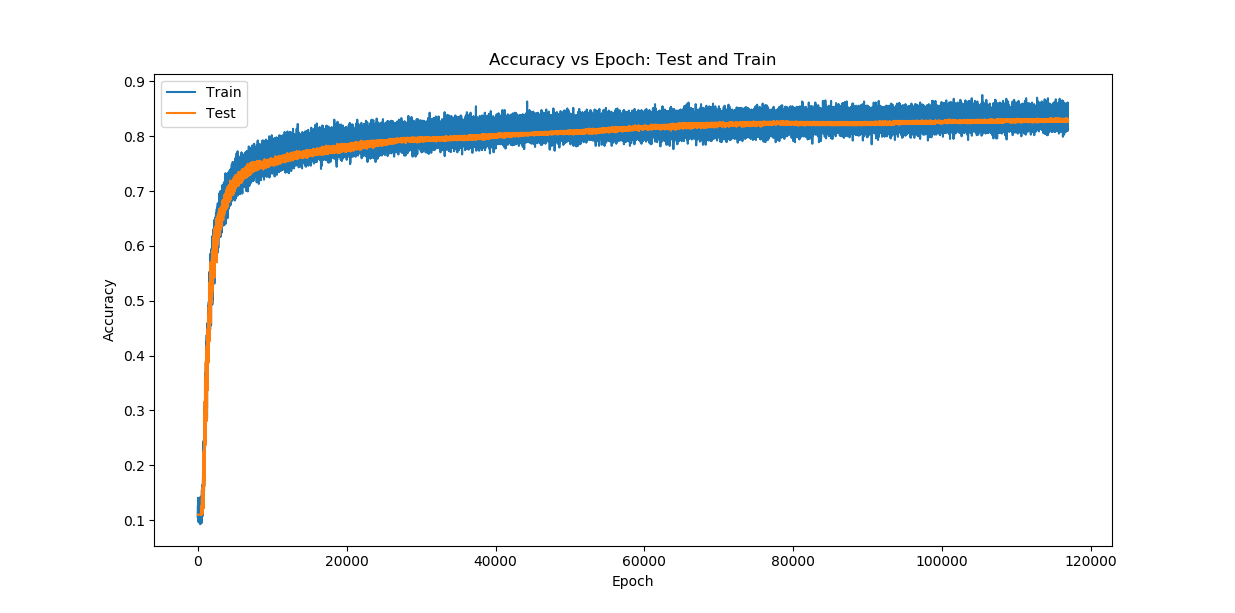
**Problem Summary:**

Classification is one of the largest categories of applications for AI. Last lab we learned how perceptrons can be used to classify data. However, there are often problems that can not be solved with a simple perceptron. To solve this, we add multiple hidden layers to aid to in classifying a dataset. In this lab we used 3 layers to classify a 28x28 greyscale image of the MNIST dataset, that being 0-9 of handwritten numbers. 5000 images were provided, 4000 were used for training, 1000 were used for testing.

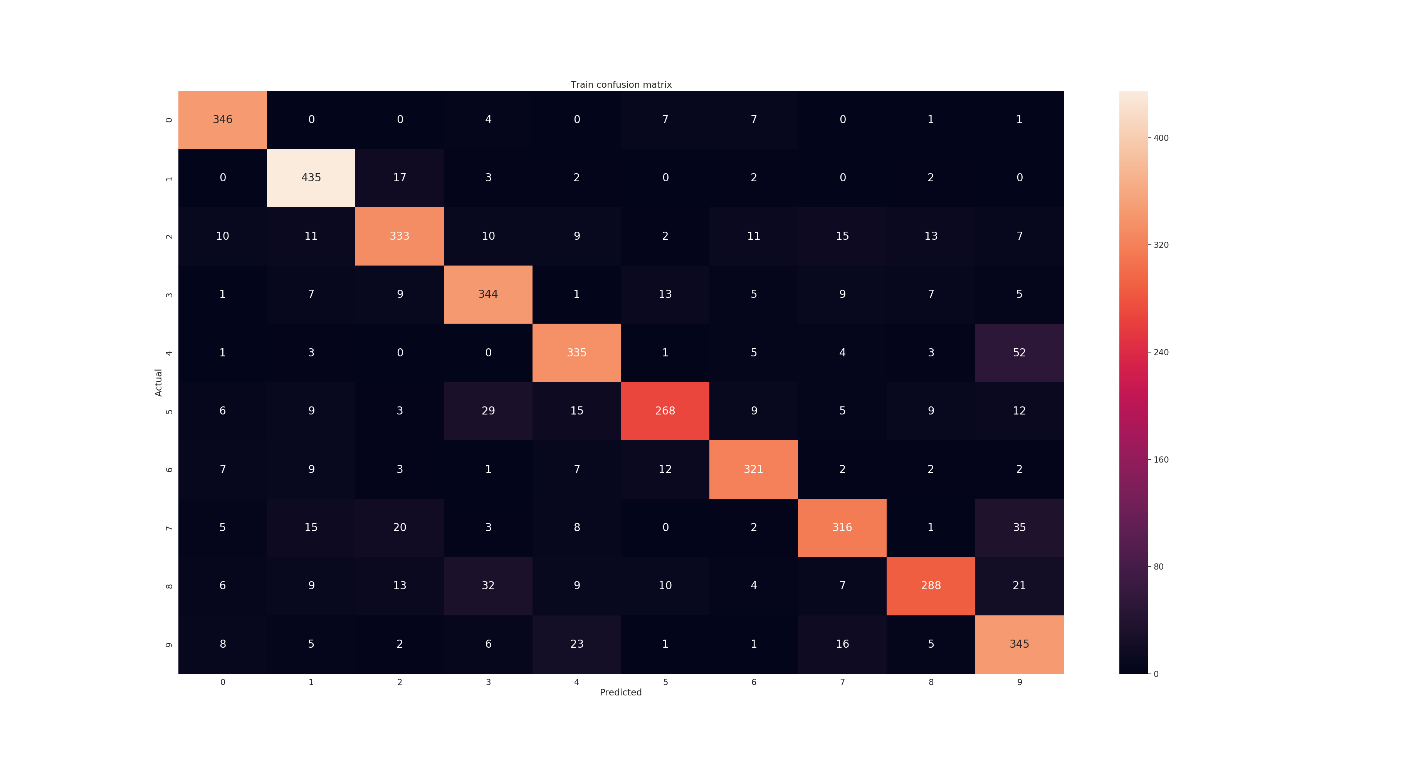
**System Description:**

The multi-layer perceptron I made consisted of 3 layers. The layers included 1 input layer, 1 output layer and 1 hidden layer. The input layer consisted of 784 inputs, 200 neurons for the hidden layer, and 10 outputs neurons, each output corresponding to a respective digit. 200 neurons were selected, as while it trained a bit slower, it allowed me to achieve high hit rate than when I use fewer neurons. The learning rate was set to 0.1. This is due to the fact that a lower learning rate of 0.01 was far to slow, meanwhile a rate of 0.1 was allowing learning to still occur, and at a much faster rate. After some research, I decided on a momentum rate of 0.5. Once momentum was added to the system, the learning occurred much quicker, especially during the early stages of learning. I settled on 0.5 as the momentum rate as it was on the lower end of recommended momentums and I had used a high base learning rate. The output thresholds were set to 0.05 and 0.95 for a zero and one respectfully. Previous values of 0.25 and 0.75 were tried both those values proved to hamper learning. The rule for selecting starting weights was selecting N number of weights from a normal distribution, then dividing those values by 100. The random distribution was done to insure a diversity of initial weight values. The division added to the weights was to ensure that no single weight was so large as to hamper learning, especially since so many weights were in this network. During training, both the training and testing performances were tested and stored in a npy file to be used later for graphing. Training for the network was manually stopped once the rate of changes in performance reached low rate of improvement. If overfitting has occurred, previous saved weights that had been saved could then be reloaded.

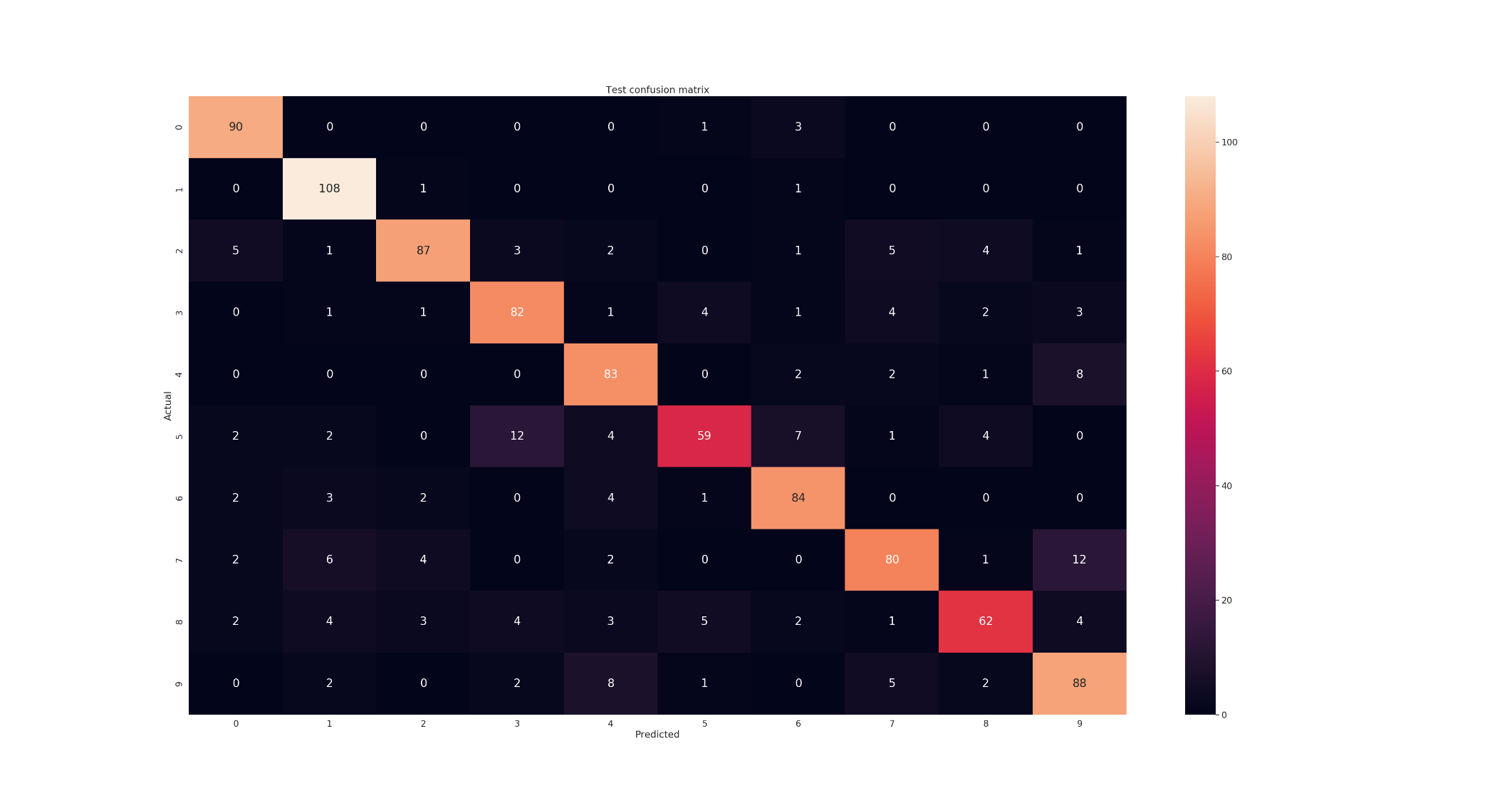
**Results:**

After 116,930 epochs of training the MLP, I stopped training. I stopped training after the performance of the network was failing to improve by any meaningful amount. Below you can see a plot of the hit rate for both testing and training datasets. The hit rate was around 84% before training of the network stopped.

**Figure 1: Training and Testing Hit Rate for MLP**

A confusion matrix tracks correct and incorrect hits of a neural network, with the diagonals being hits and every other square being a miss. Confusion matrices for both the training and test dataset where created and can be seen below.

**Figure 2: Confusion Matrix For The MLP On Training Data**

**Figure 3: Confusion Matrix For The MLP For Test Data**

**Analysis:**

Looking at the hit rate graph we can see that both the test and training dataset are very close. This is good, as a large divergence in the training and testinng graphs would suggest that the network was overfitting on the training data. Judging from these results I can say that the network did not overfit. Looking at the graph, we achieve around 84 percent hit rate. While this is good it could always be better. One possible reason for this could be that even with the relatively high learning rate and momentum, that while the network was descending down the error gradient, that it got stuck in a local minimum.

Looking at the confusion matrices we can see that the network overall did very well. The matrices look very similar with regards to values it got right and the values that were guessed wrong. This is good as it means our network is generalized enough to be useful on data it has never seen before. One point of interest is that the network had a bit more issues with the group of 3s,5s,8s and the group of 4s and 9s. This makes sense, as numbers written poorly in these groups can sometimes look like other numbers. In the dataset there must have been examples of these poorly written numbers, which gave the network some issues.

**Problem 2: Using Autoencoders to Encode and Generate Images:**

**Problem Summary:**

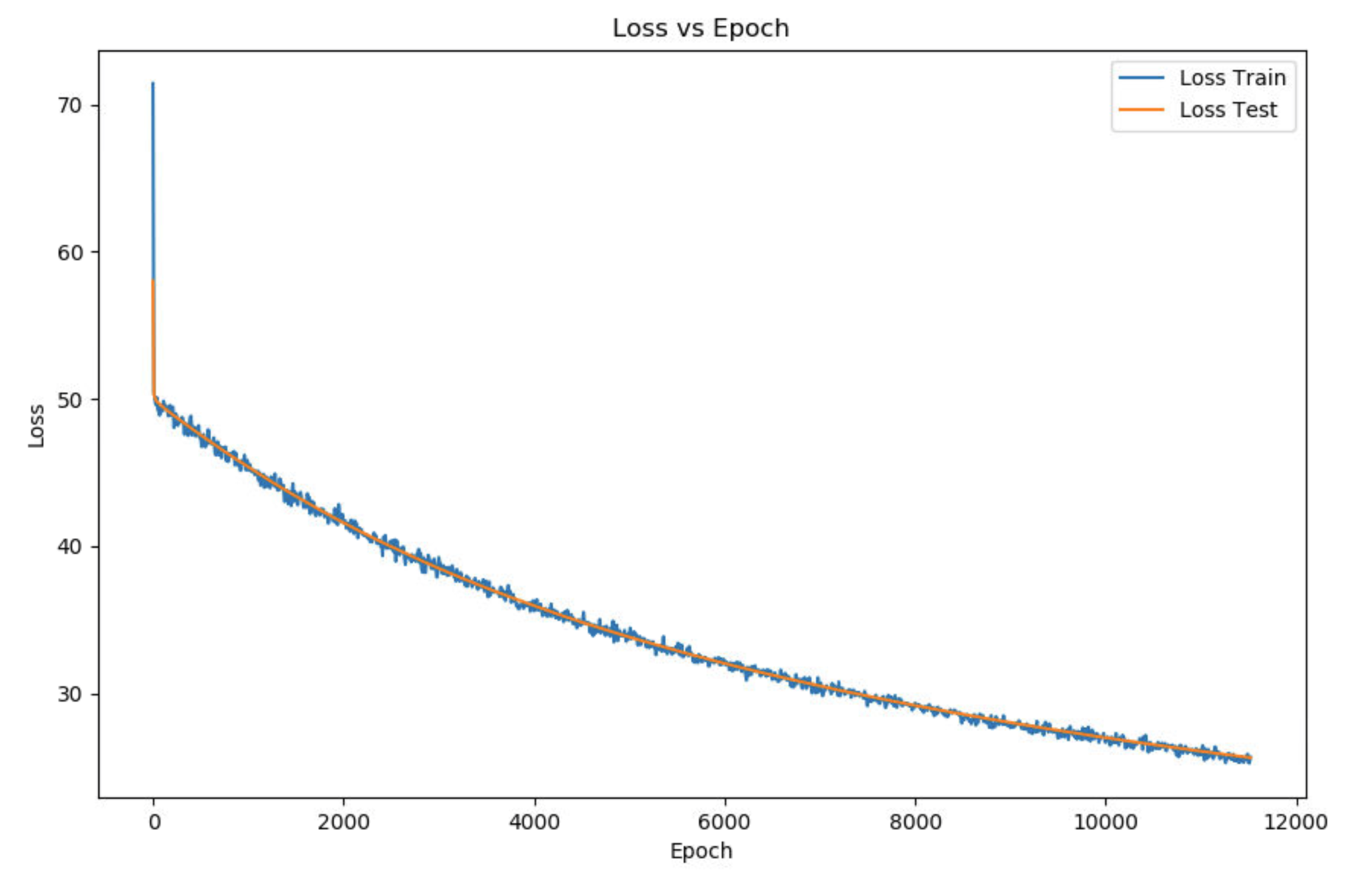
While the majority of machine learning is in the realm of classification, there are other uses for machine learning outside of classification. Two examples of those other applications include encoding information into a smaller dimensional space, and generating new material of a given dataset. Autoencoders are a simple way of doing both. In this lab we will use an autoencoder to generate images of digits after being encoded, as well as look into the features of autoencoders.

**System Description:**

The autoencoder used 3 layers. The first layer contained 784 inputs, 1 for each of the pixels. The second layer contained 200 neurons, the same as the MLP’s middle layer. The last layer contained 784 neurons so that the network can attempt to reconstruct the images. The learning rate was set to 0.1 and the momentum was set to 0.5. The learning rate and momentum were selected for the similar reasons for MLP. Due to not having a GPU, a high learning rate is needed as the networks took a very long time to train. However, due to having a high learning rate, I did not want to have a high momentum as well as to prevent the network from learning. The initial weights were selected in the same fashion as the MLP, that being a normal distribution that was then divided by 100. A normal distribution to ensure a good diversity of initial weights, and divided by 100 to insure that any single weight does not overwhelm the network and prevent learning. As the network trained, the loss for both the training dataset and the test dataset were measured and stored in a npy file to be later referenced in a graph. Once the rate of loss decreasing became sufficiently slow, I stopped the network and used weights I had previously saved for the rest of my analysis.

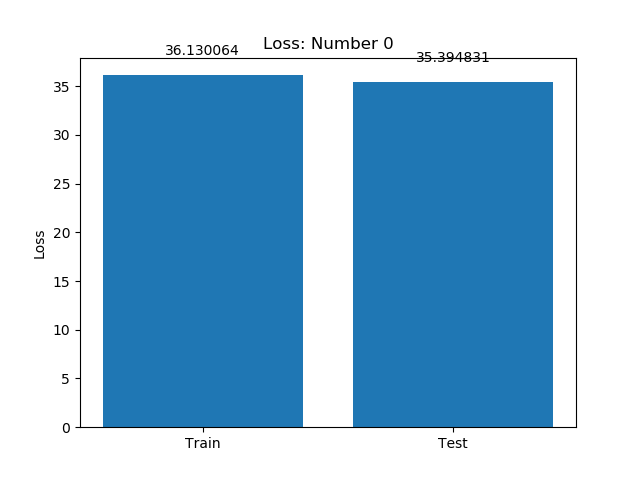
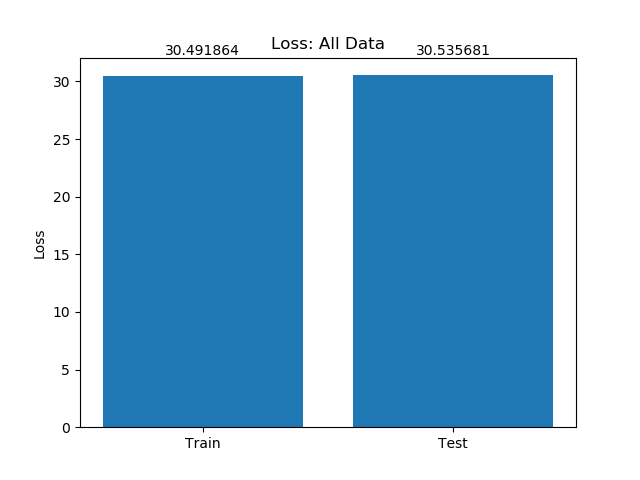
**Results:**

Below one can see a plot of the loss over time with both the training data and the test data. The loss for each dataset stay fairly close together as one can see.



**Figure 4: Loss Rate For The Autoencoder With Training And Test Data**

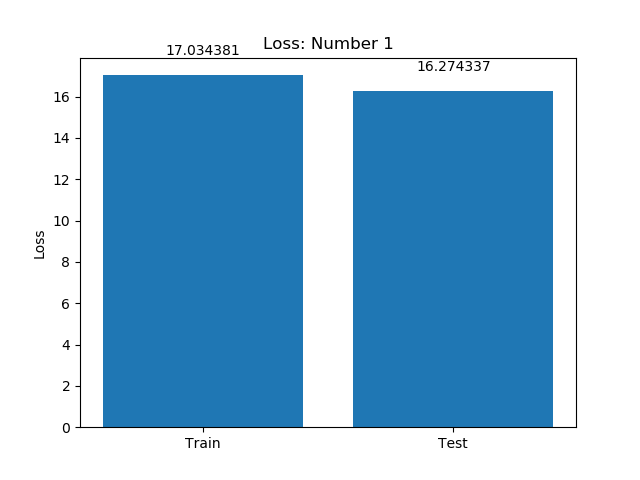
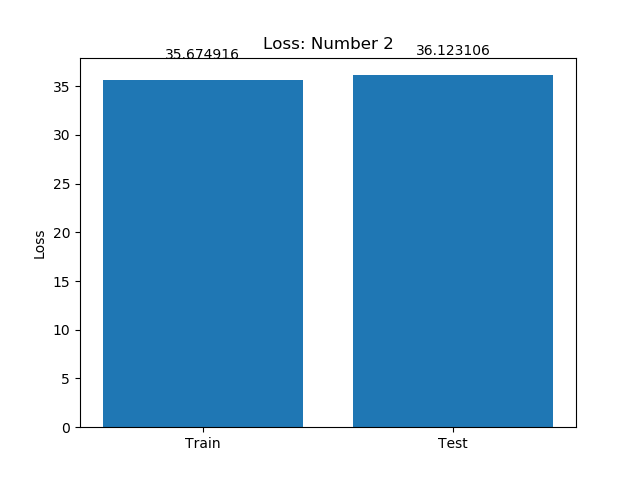
Below one can see a bar graph showing the performance of the autoencoder. The performance was very similar for both the training dataset and the testing dataset when it comes to all the data at once. However, for some numbers we can see a significant difference between the two losses. We also see that for some of the numbers, the loss is significantly different from the others.

****

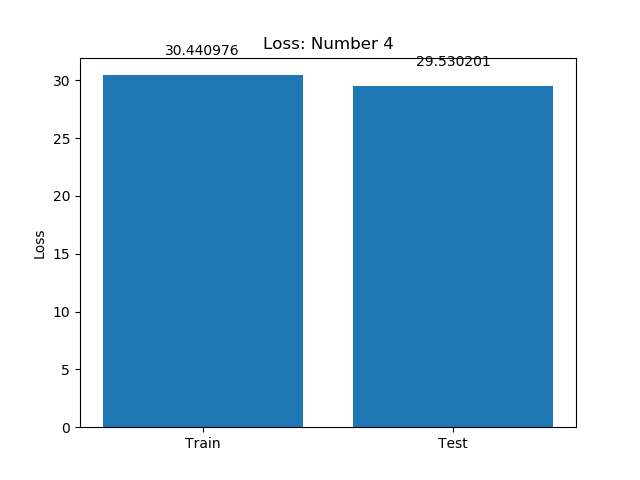
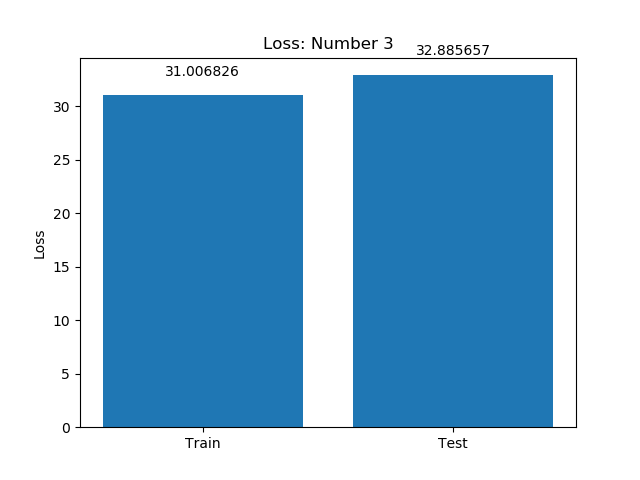
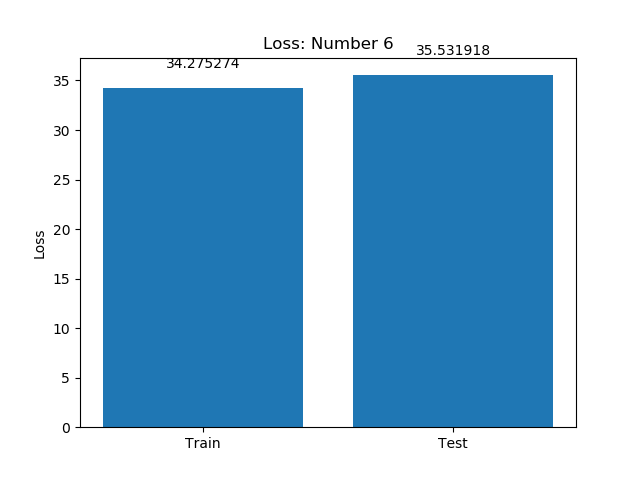
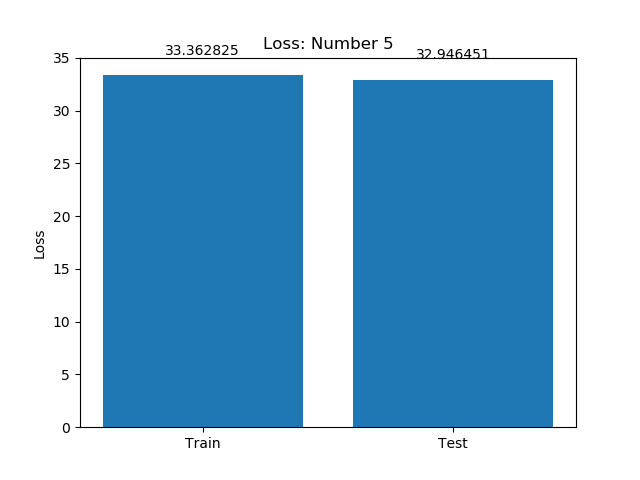
**Figure 8: Loss For 2’s**

**Figure 5: Loss For All Data**

**Figure 6: Loss For 0’s**

****

**Figure 7: Loss For 1’s**

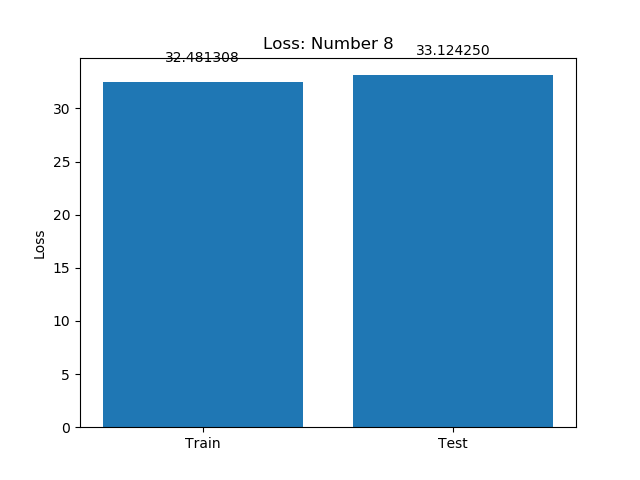
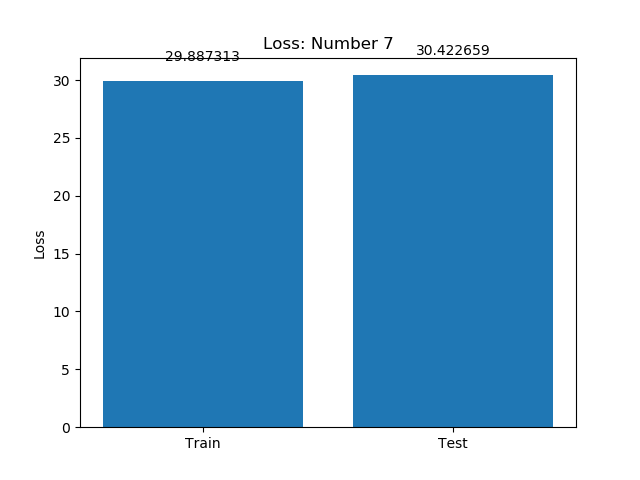
****

**Figure 10: Loss For 4’s**

**Figure 9: Loss For 3’s**

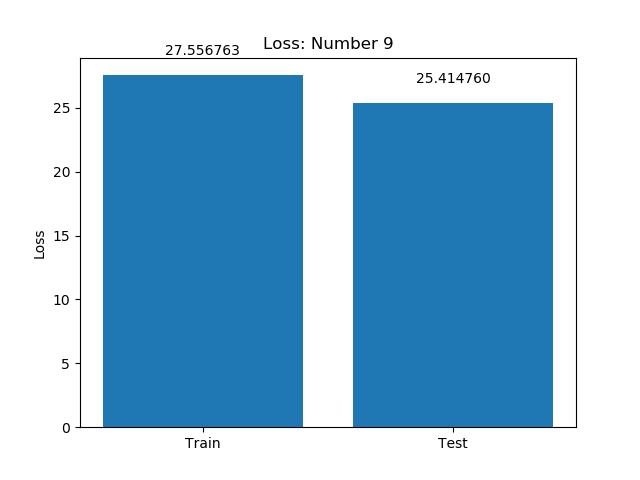
**Figure 10: Loss For 5’s**

**Figure 11: Loss For 6’s**

****

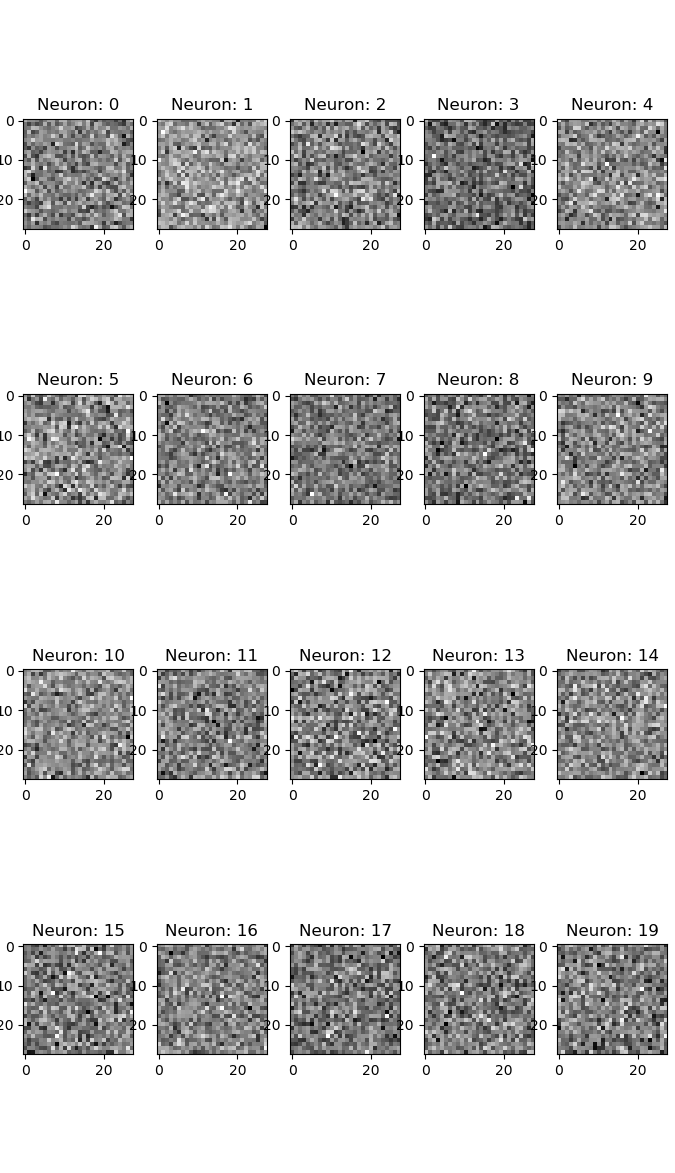
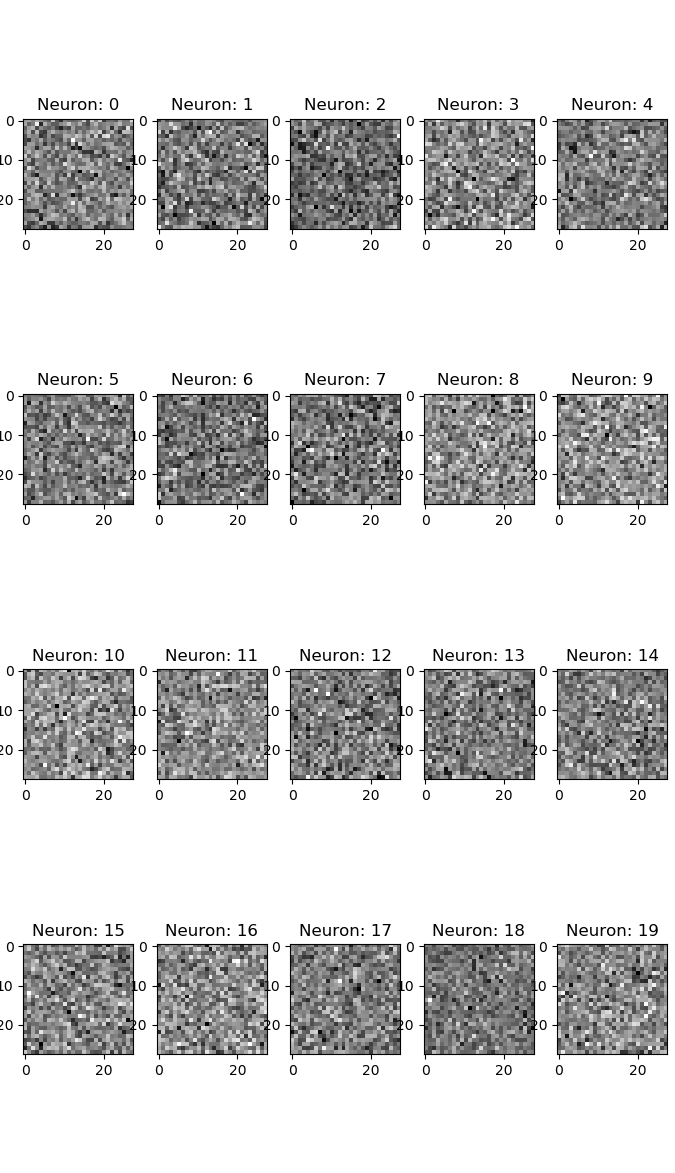
**Figure 13: Loss For 8’s**

**Figure 12: Loss For 7’s**

****

**Figure 14: Loss For 9’s**

**Features:**

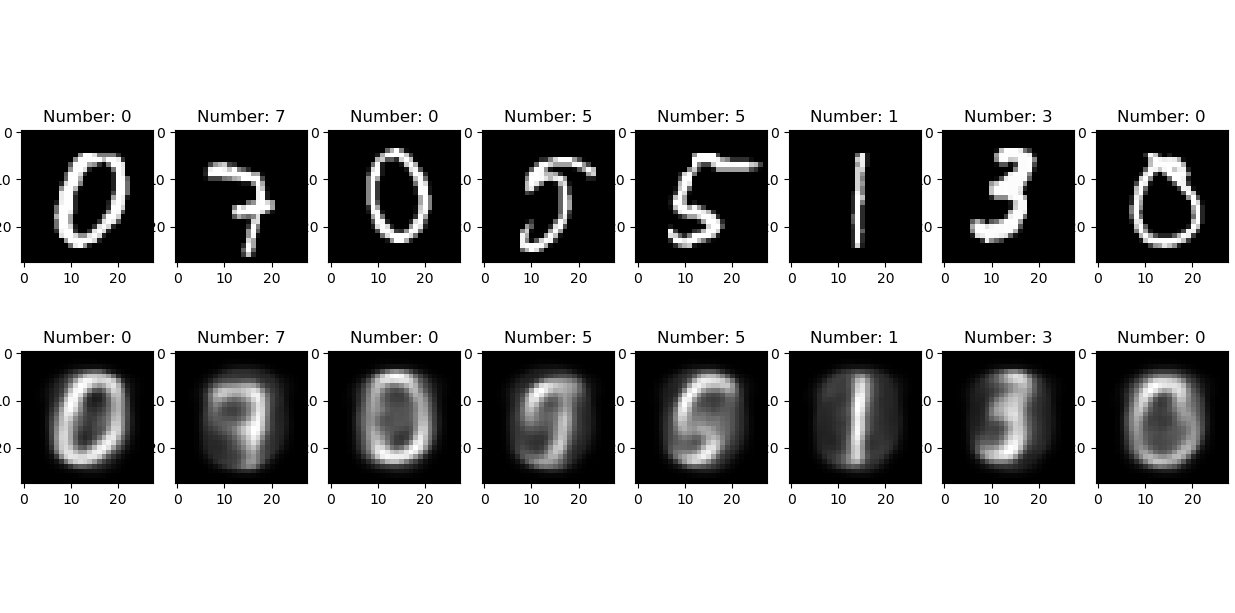
****Below one can see neuron feature maps for the weights connecting the hidden layer. Below are features maps for both the MLP and the Autoencoder. While the neuron selection for the features were picked randomly, the neurons that were selected were selected for both of them. While the second set of weights could have been chosen for the autoencoder, in order to compares the MLP with the Autoencoder the first set of weights were chosen

**Figure 16: Feature Map For Autoencoder**

**Figure 15: Feature Map For MLP**

These feature maps are mostly noise. However, I do notice a few patterns between them all. It seems like a general rule is that the neurons are the same darkness in the feature maps. Additionally, the really dark pixels, which would correspond to 1’s, seems to appear in the same locations for both feature maps. Having worked previously with neural networks I expected this, especially for the first layer. As more layers are added I would expect to see more details stick out. I think it would be interesting to try an adversarial attack on these feature maps and try to enter them into the network as I have read about.

**Sample Outputs:**

Below are 8 random picture samples from the test data that were inserted into the network and the network tried to recreate them. The top images are the ones inserted into the network and the bottom images are the ones that the network first encoded, then decoded. An application for this would be to feed in a random 200 long array and, if trained well the autoencoder would be able to make an image of a number.

**Figure 16: Sampled Outputs From The Autoencoder**

**Analysis:**

The autoencoder overall did a good job at learning the different images and recreating them. Looking at the bar graphs, some of the digits had a lower or higher loss than the average. I think the reason for this is that some digits, like the 1 are very distinct, and were probably easier to learn. Conversely 2 had a higher loss. This makes sense to me, as a poorly drawn 2 can look like a 5 and other digits. When the digits have the possibly to look like the others, the network has to learn more from them before it is able to easily distinguish. Once the encoding is better, the decoding will be better and the loss will lower.

In the features, I noticed a pattern of “hot spots” between the two networks. The dark spots in the images would result in a 1, meaning that neuron weight was highly active. Since the encoding process for the two network is the same dimensions, it makes sense that some same things are learned in both networks creating these shared hotspots.

The digit that seemed the easiest to reconstruct was the 1. Some of the harder ones were the 7, and 3. I think the reason for this is clear. The one is very distinct, and hard for a human to add much variation to. However, 3s and 7s can vary wildly, leading it to be hard to recreate them.