Computer Vision, 16720A - Homework #5

Neeraj Basu
neerajb@andrew.cmu.edu

November 26, 2020

Q1.1

$$\sigma(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}} \tag{1}$$

Our objective is to prove that $\sigma(x + c) = \sigma(x)$. Writing the function out with an input of x + c:

$$\sigma(x + c) = \frac{e^{x_i + c}}{\sum_j e^{x_j + c}} \tag{2}$$

We can use the fact that e^{x+c} can be rewritten as $e^x e^c$ to rewrite equation (2) as:

$$\sigma(x + c) = \frac{e^{x_i} e^c}{\sum_j e^{x_j} e^c} \tag{3}$$

Since both the numerator and denominator have been multiplied by e^c, the ratio of the equation does not change. Therefore proving $\sigma(x + c) = \sigma(x)$.

$$\sigma(x_i + c) = \frac{e^{x_i}}{\sum_j e^{x_j}} = \sigma(x_i) \tag{4}$$

The reason we use the c=max(x_i) trick is to keep us from assigning too high of an exponent to e. By scaling x by the largest value within x, this will keep the calculation from overflowing.

Q1.2

The softmax function has a range of [0,1] and the sum over all of the elements is 1.

The softmax function takes an arbitrary real valued vector x and turns it into a vector of values between zero and one that sum to one. This resulting vector can be thought of as a set of probabilities.

$s_i = e_i^x$ maps numbers into positive values while preserving the order. $S = \sum s_i$ calculates the value which will normalize the vector. $\frac{1}{s} s_i$ normalizes the vector so every value falls between 0 and 1.

Q1.3

Writing out the equation for the forward propagation of a multi-layer neural network, the format would look similar to:

$$y = f_3(w_3 f_2(w_2 f_1(w_1 x + b_1) + b_2) + b_3) \tag{5}$$

Here the function, f is assumed to be a non-linear activation function which can be thought of as additional matrix multiplications and additions that do not change the linearity of the output. Therefore, we can rewrite equation (5) as:

$$sy = \prod_{i=1}^{n} w_i x + \sum_{i=1}^{n} \left(\prod_{j=1}^{n} w_j \right) b_i \tag{6}$$

It becomes very clear that the output is a result of linear regression. The output consists of nothing more than matrix multiplication and addition when the activation function is a linear function.

Q1.4

Given the equation for the the sigmoid function:

$$\sigma(x) = \frac{1}{(1 + e^{-x})} \tag{7}$$

Taking the gradient of the function:

$$\frac{d}{dx} \frac{1}{(1 + e^{-x})} = \frac{e^{-x}}{(1 + e^{-x})^2} \tag{8}$$

Expanding out the denominator:

$$\frac{1}{(1 + e^{-x})} - \frac{1}{(1 + e^{-x})^2} \tag{9}$$

We now have the gradient in terms of exclusively $\sigma(x)$.

$$\frac{d}{dx}\sigma(x) = \sigma - \sigma^2 \tag{10}$$

Q1.5

In order to solve for $\frac{dJ}{dW}$, $\frac{dJ}{dx}$ and $\frac{dJ}{db}$, we need to define the intermediate step of $\frac{dJ}{dy}$ which is denoted as $\delta \in R^{kx1}$. To begin, we can solve for $\frac{dJ}{dW}$ using $\frac{dJ}{dy}\frac{dy}{dW}$. We can solve for $\frac{dy}{dW}$ using the following equation:

$$y = Wx + b \tag{11}$$

$$\frac{dy}{dW} = x \tag{12}$$

Where $x \in R^{dx1}$. Therefore, prior to matrix multiplication between x and δ we must transpose x. This can be rewritten in matrix form as:

$$\frac{dJ}{dW} = \begin{bmatrix} \delta_1 x_1 & \delta_1 x_2 & ... & \delta_1 x_d \\ \delta_2 x_1 & \delta_2 x_2 & ... & \delta_2 x_d \\ \vdots & \ddots & \ddots & \vdots \\ \delta_k x_1 & \delta_k x_2 & ... & \delta_k x_d \end{bmatrix} \tag{13}$$

Applying this same logic to $\frac{dJ}{dx}$, we can find our intermediate step of $\frac{dy}{dx}$ needed for $\frac{dJ}{dy}\frac{dy}{dx}$ as follows:

$$\frac{dy}{dx} = W \tag{14}$$

In this case, $W \in R^{kxd}$ and $\delta \in R^{kx1}$ so we must transpose W prior to matrix multiplication.

$$\frac{dJ}{dx} = \begin{bmatrix} W_1 \delta_1 \\ W_2 \delta_2 \\ \vdots \\ W_d \delta_d \end{bmatrix} \tag{15}$$

Finally for $\frac{dJ}{db}$ our intermediate step of $\frac{dy}{db}$ needed for $\frac{dJ}{dy}\frac{dy}{db}$:

$$\frac{dy}{db} = 1 \tag{16}$$

Since b is a scalar of 1, we can represent $\frac{dJ}{db}$ using only δ:

$$\frac{dJ}{db} = \begin{bmatrix} \delta_1 \\ \delta_2 \\ \vdots \\ \delta_k \end{bmatrix} \tag{17}$$

Converting back to scalar form, we end with the following equations:

$$\frac{dJ}{dW} = \delta x^T \tag{18}$$

$$\frac{dJ}{dx} = W^T \delta \tag{19}$$

$$\frac{dJ}{db} = \delta \tag{20}$$

Q1.6.1

Since the sigmoid activation function compresses a large input space into an input between 0 and 1, a large change in the input function will cause a small change in output. The maximum possible derivative of the activation is very small ($\sim$.25). Since the gradient is found during backpropogation, for multi-layered networks where we are multiplying these derivatives together, the gradient will decrease exponentially as we propagate through the layers, ultimately resulting in a vanishing gradient.

Q1.6.2

The output range of tanh is [-1,1] while the output range of sigmoid is [0,1]. Therefore tanh has a larger range of potential input values including negatives. We might prefer the tanh function over the sigmoid due to the fact that the derivative of the tanh function is higher than that of the sigmoid which may help in regards to the vanishing gradient problem. The figure below does a good job visualizing how the derivative of the tanh function has a higher maximum value than that of the sigmoid function (source).

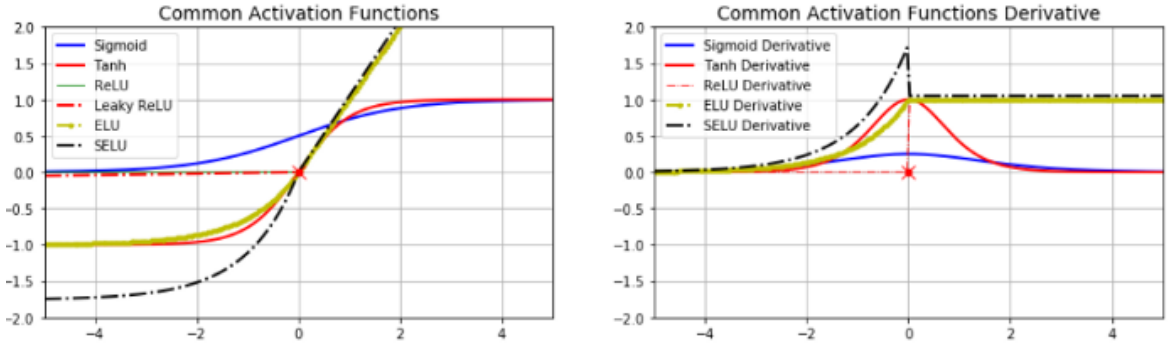

Figure 1: Derivative of Sigmoid vs. Tanh

Q1.6.3

Since the derivative of the tanh function is highter than that of the sigmoid, you will minimize your cost function faster. This will also help with the vanishing gradient problem layed out in 1.6.2. The following graph (source) does a good job of displaying the derivative of the tanh function and showing how it's max value is 1 which is higher when compared to the derivative of sigmoid:

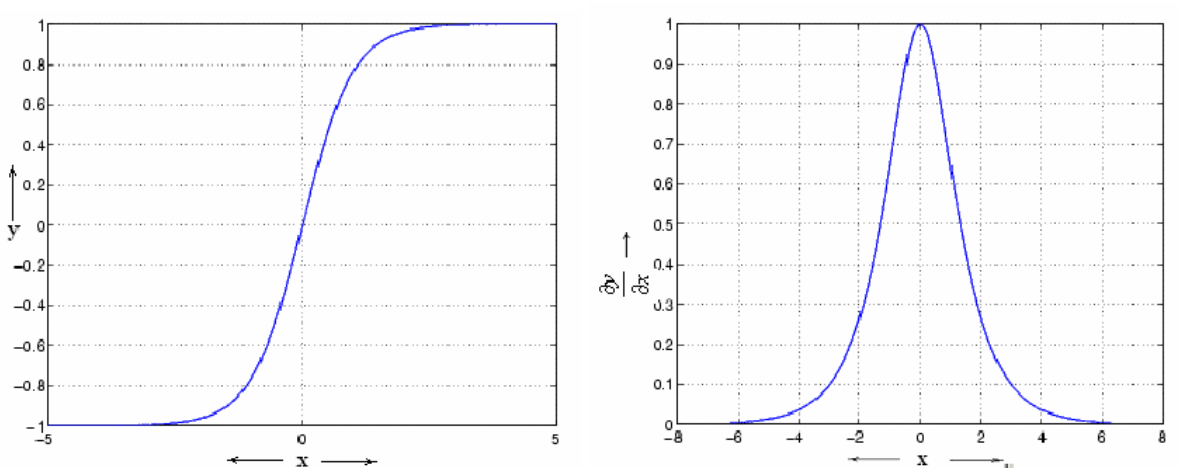

Figure 2: Derivative of Sigmoid vs. Tanh

Q1.6.4

By observation of figure (1), it becomes clear that tanh is nothing more than a scaled version of sigmoid. Since sigmoid is symmetric about x and returns values between 0 and 1, we can rewrite the equation as:

$$1 - \sigma(x) = \sigma(-x) \tag{21}$$

Using the equation of sigma defined in (7), this equation will then look like:

$$1 - \frac{1}{(1 + e^{-x})} = \frac{1}{(1 + e^{x})} \tag{22}$$

Writing the equation for tanh(x) out:

$$\frac{e^{x} - e^{-x}}{e^{x} + e^{-}x} \tag{23}$$

We can simplify this equation to the form:

$$1 - \frac{2}{e^{2x} + 1} \tag{24}$$

Looking at equation (24) and expressing it in terms of the sigmoid function:

$$tanh(x) = 1 - \frac{2}{e^{2x} + 1} = 1 - 2\sigma(-2x) \tag{25}$$

Since the function is symmetric, it can be represented as:

$$tanh(x) = 2\sigma(2x) - 1 \tag{26}$$

Here we can see the tanh function is nothing more than a scaled version of the sigmoid function.

Q2.1.1

By initializing every weight to zero, all nodes of the hidden layers will have the same weights. As an example, in the case where our activation function is the sigmoid function, the forward propagation would return a value of 0.5. Then during back propagation, the derivative of the weights in respect to the loss function will be uniform. Therefore each node will end up with the same value and the neural network will end up learning a single function when the objective is to learn multiple functions. In general, it is a bad idea to initialize your weights to the same values. If all weights are updated by the same value, the weights will remain equivalent which is eliminating the non-linearization of the model. Ultimately your network will end up thinking every outcome is equally as probable and therefore the output of the trained network is futile.

Q2.1.3

In order for the stochastic gradient descent algorithm to work, the network needs to be passed small random wights when initialized. This also keeps the set of weights non-uniform which will keep many of the issues I laid out in 2.2.1 from manifesting (loss of non-linearization, failure to train, etc.).

We scale the weights in accordance to layer size to keep the variance of the input data to each layer from becoming too large. If the weights are too big, we end up on the other side of vanishing gradient problem, where our gradients are so large they become useless. Since the sigmoid function is flat for larger values, the gradient would be close to zero as the activation becomes saturated and this gradient would propagate through the network creating meaningless results.

Q3.1/3.2

The best accuracy I was able to achieve was 77.66% with a learning rate of 0.3e-2 and a batch size of 15.

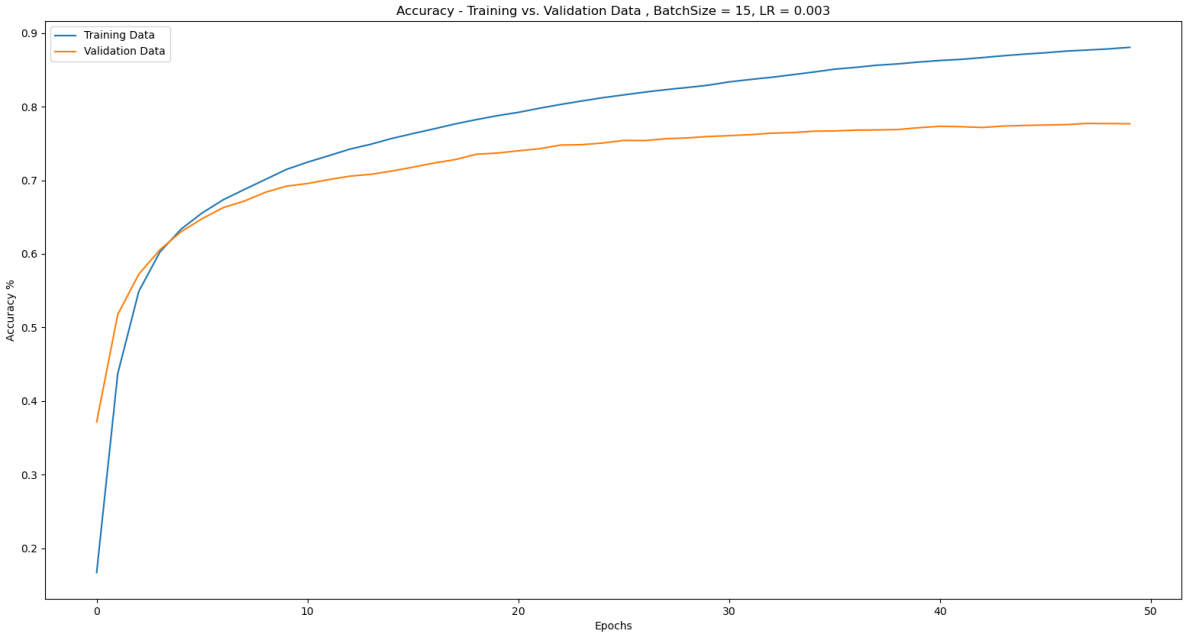

Figure 3: Accuracy, Batch Size = 15, LR = 0.003

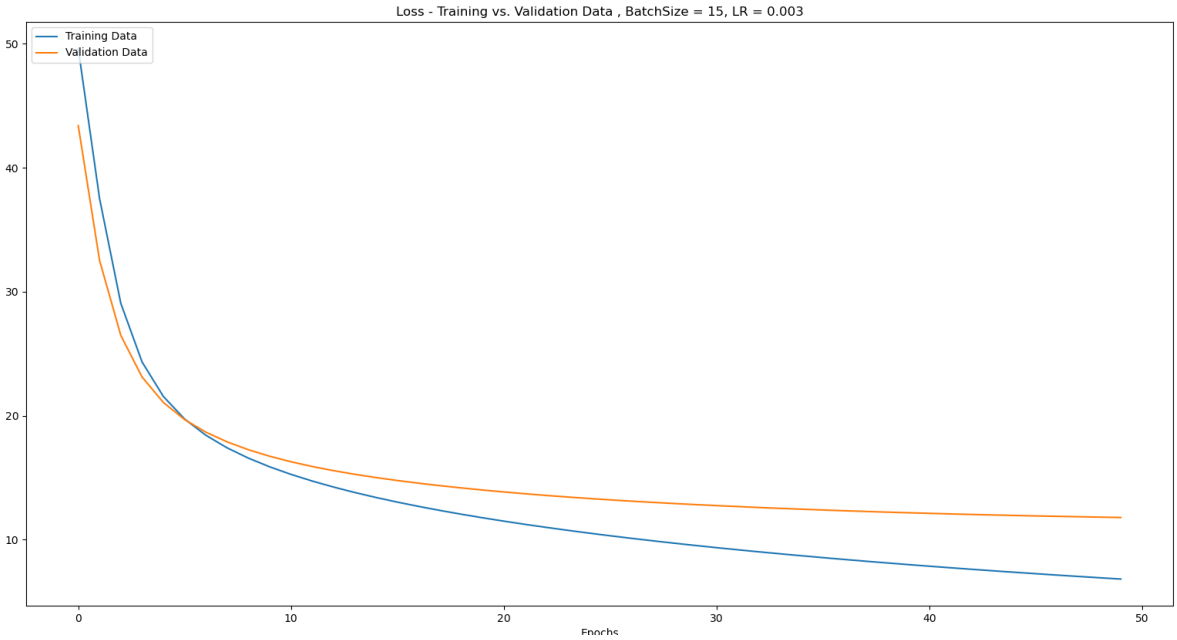

Figure 4: Loss, Batch Size = 15, LR = 0.003

Since the learning rate controls how aggressively the neural network adjusts when responding to the estimated error, a learning rate which is too high will cause the system to adjust aggressively. After multiplying my best learning rate by 10 my new learning rate was 0.3e-1. With this new learning rate, I achieved an accuracy of 52.23%. With this increased learning rate, my neural net preformed significantly worse. This is likely due to the fact that my network was missing the local minima due to aggressive jumps in the gradient. This is also visually represented in the plots as we see jagged changes as we progress through the epochs. The loss only reached a plateau of $\sim$25 compared to my best run which was $\sim$10. This worse performance is likely due to the model missing the local minima with each adjustment therefore never truly minimizing error.

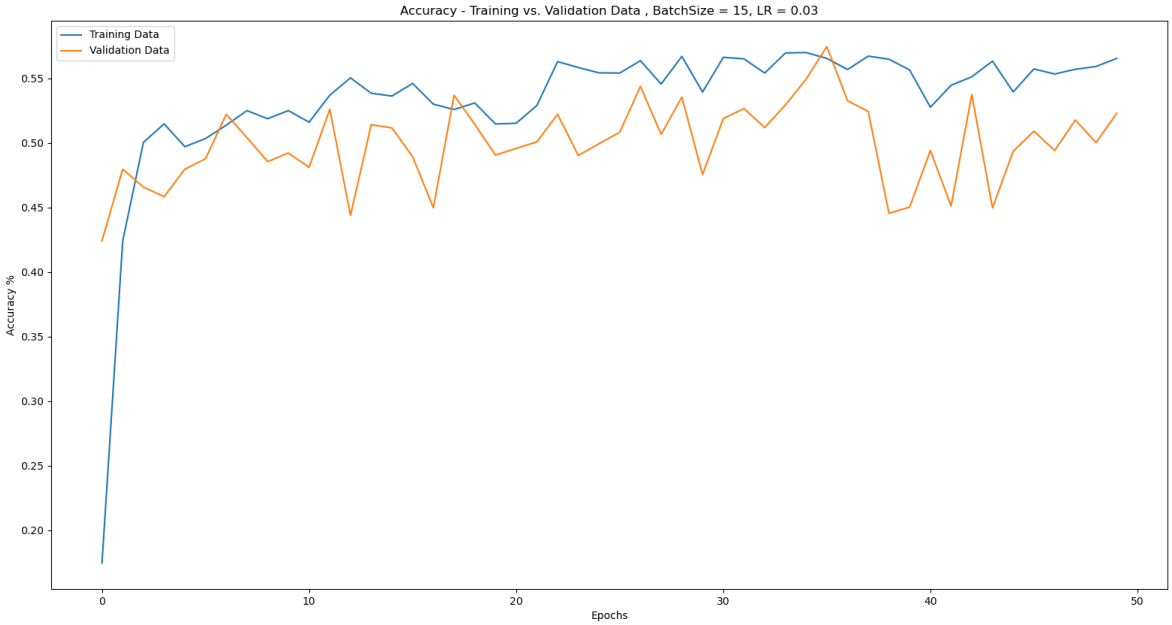

Figure 5: Accuracy, Batch Size = 15, LR = 0.03

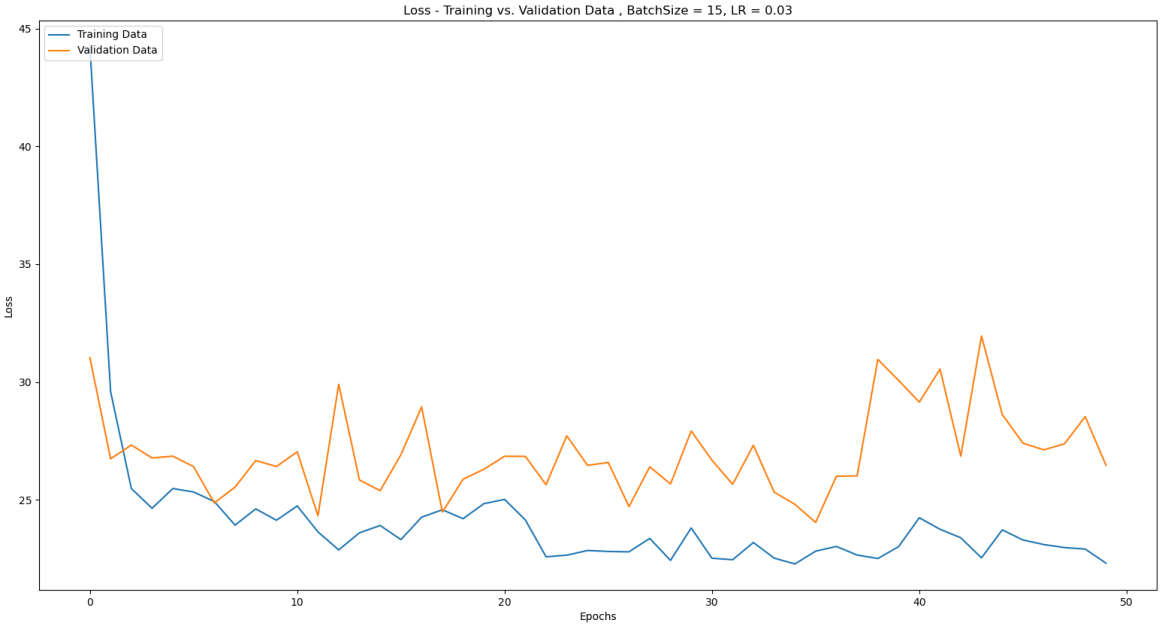

Figure 6: Loss, Batch Size = 15, LR = 0.03

Conversely, when multiplying my learning rate by $\frac{1}{10}$, the resulting response was much smoother. My decreased learning rate was .3e-3. With this new learning rate, I achieved an accuracy of 66.05%. With a smaller learning rate, we don't see the same jagged curve which we saw with the increased learning rate. In this scenario the gradient is in much smaller steps, therefore decreasing the probability of overshooting the local minima. While the loss does not plateau as low, if we were to increase the number of epochs, the system would more likely out preform my best run proposed above. However, this would be at the expense of high computational cost and time.

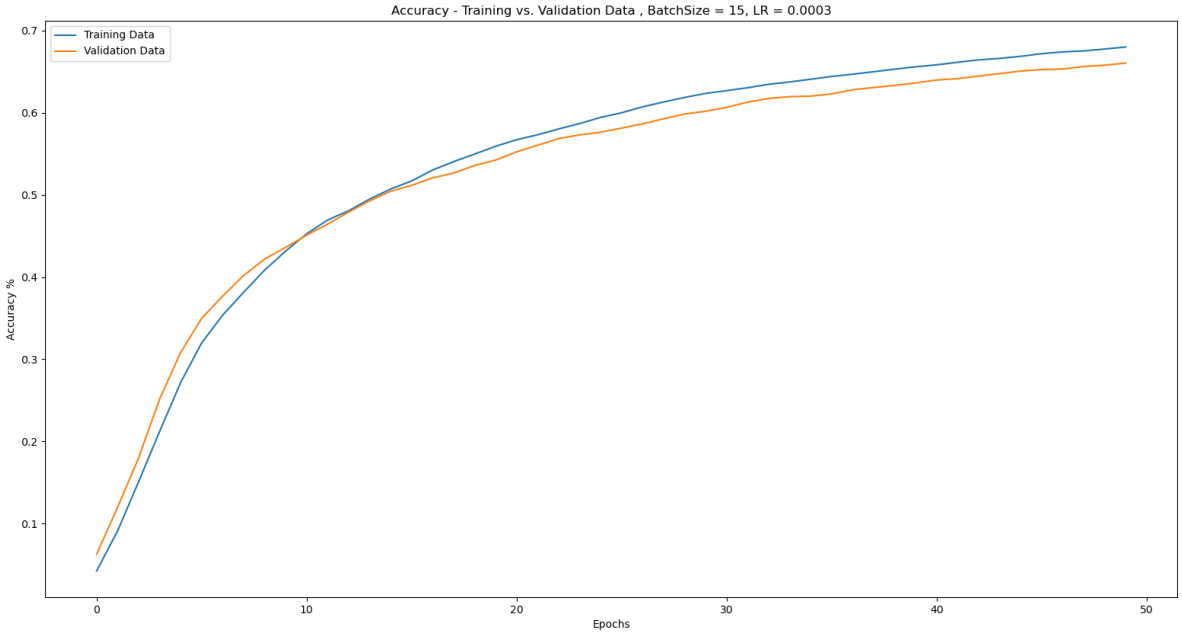

Figure 7: Accuracy, Batch Size = 15, LR = 0.0003

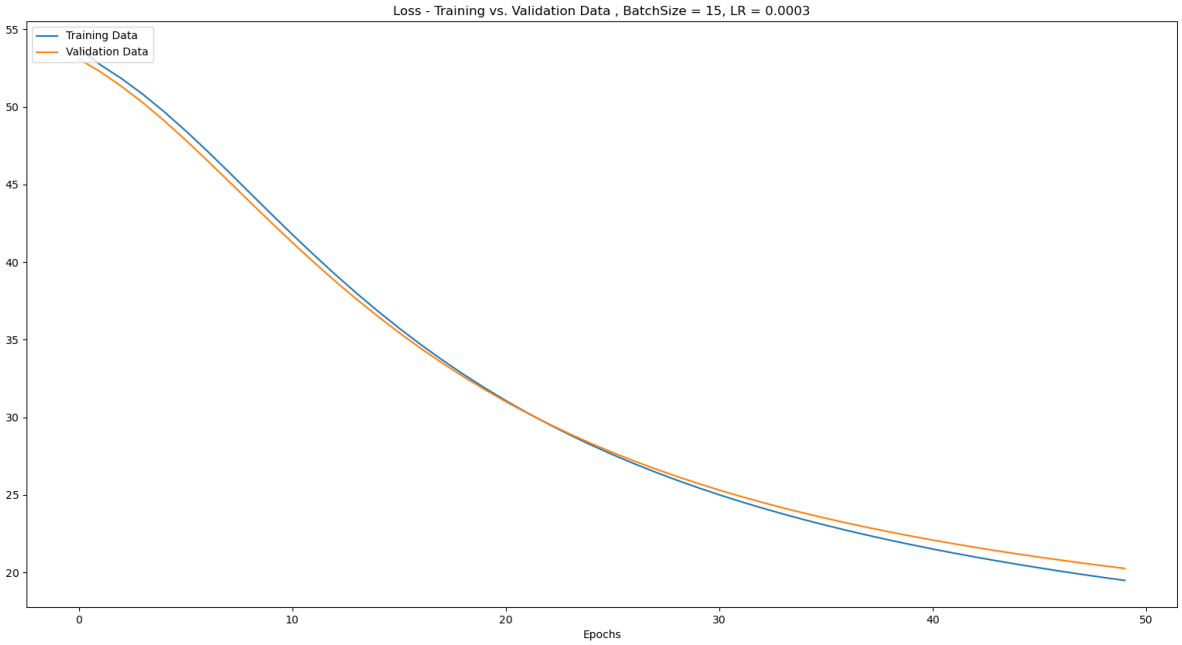

Figure 8: Loss, Batch Size = 15, LR = 0.0003

Q3.3

Prior to training, each nueron's weight is clearly randomized. However after training, you can very vaguely see which neurons are activating during the network's learning. The grouping of weights is representative of an interest feature the neuron is learning. In theory if we were to visualize the weights after more layers, we could start to make out what shapes each neuron is 'looking' for as it learns.

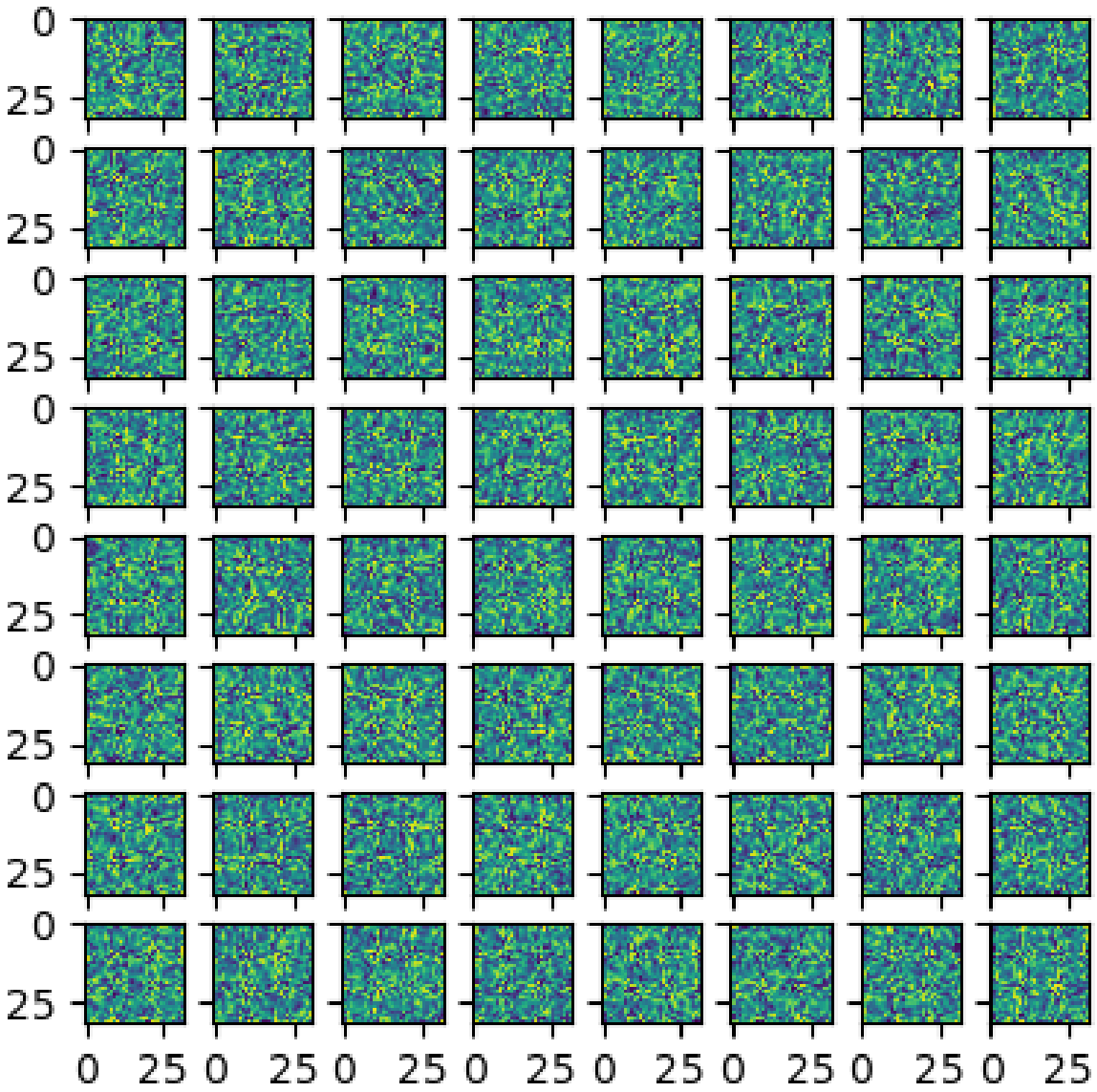

Figure 9: ImageGrid of Network Weights Before Training for each Neuron

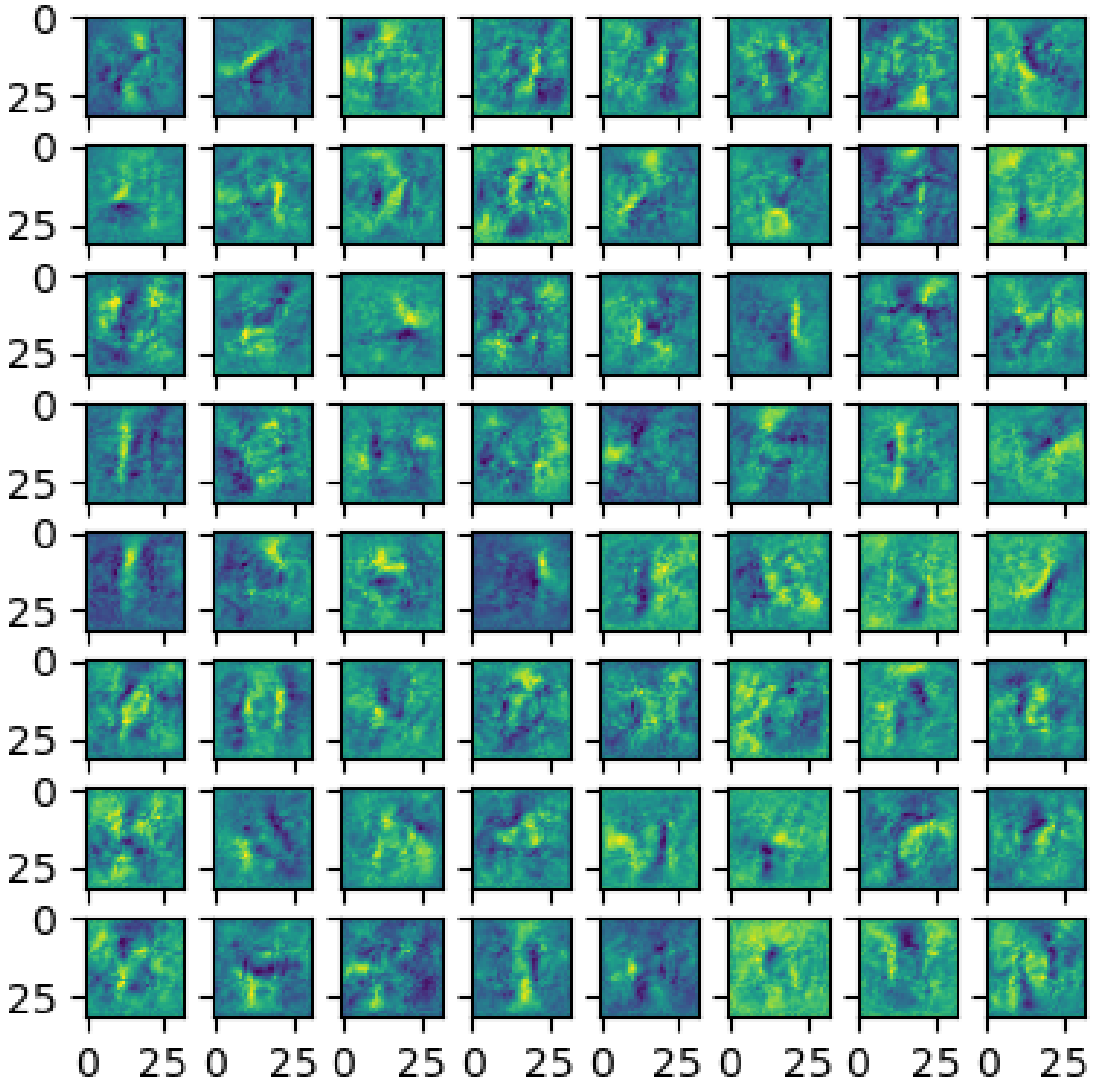

Figure 10: ImageGrid of Network Weights After Tuning for each Neuron

Q3.4

As expected, the confusion matrix visualizes the same points of confusion we encounter as humans. The most common mistakes represented by the confusion matrix:

0 (zero) vs. O (Capital 'oh')
S vs. 5
Z vs. 2
1 vs. I
F vs. P

We will explicitly see in problem 4 how these mix-ups manifest themselves during character recognition.

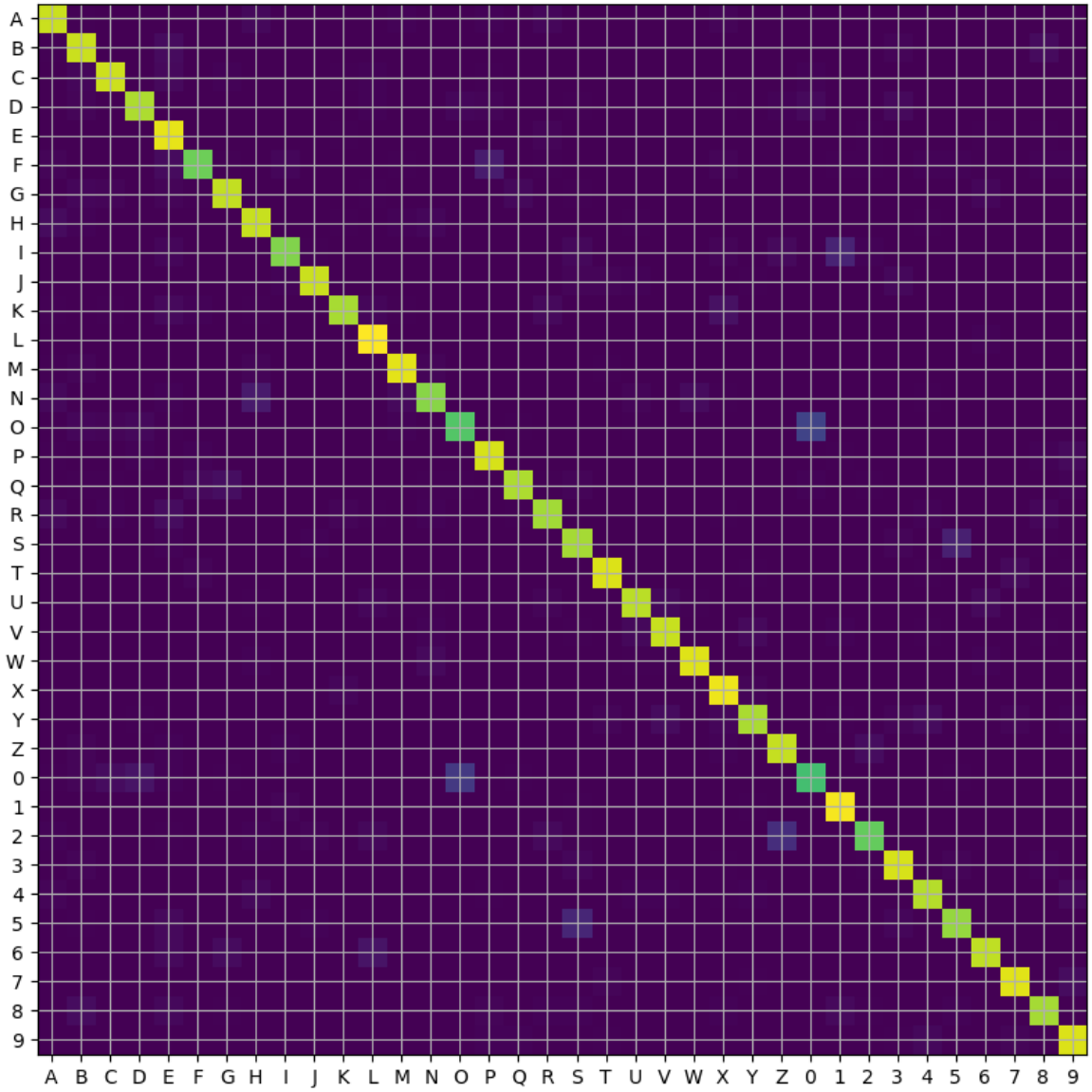

Figure 11: Confusion Matrix

Q4.1

1. The first assumption we make is that we are being passed well written differentiable characters:

a. Characters are well spaced out from each other and not overlapping
b. Characters are fully 'closed'
c. Characters do not resemble other characters

We can see there are a number of instances where these assumptions are not held true and therefore result in incorrect bounding boxes/letter recognition.

Figure 12: A z which looks like a 2

Figure 13: An unclosed O

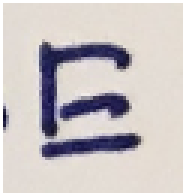

Figure 14: An unconnected E

2. The second assumption is that images are easily 'segment-able' given that they are on plain backgrounds. Fortunately all of the images we dealt with met this assumption, therefore making this method viable for assigning bounding boxes.

Q4.3

The results of findLetters is shown below. As expected, there are a number of extra bounding boxes which were a result of unconnected letter segments.

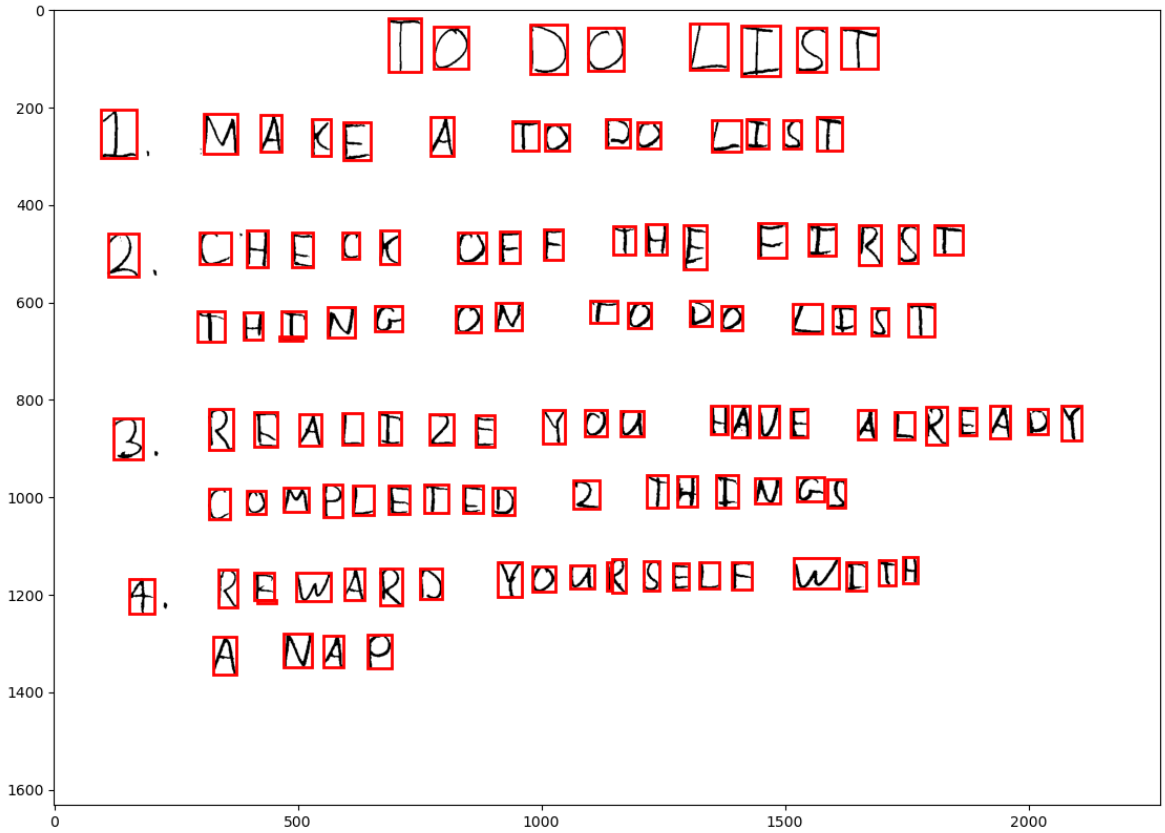

Figure 15: Q43 - List with Bounding Boxes

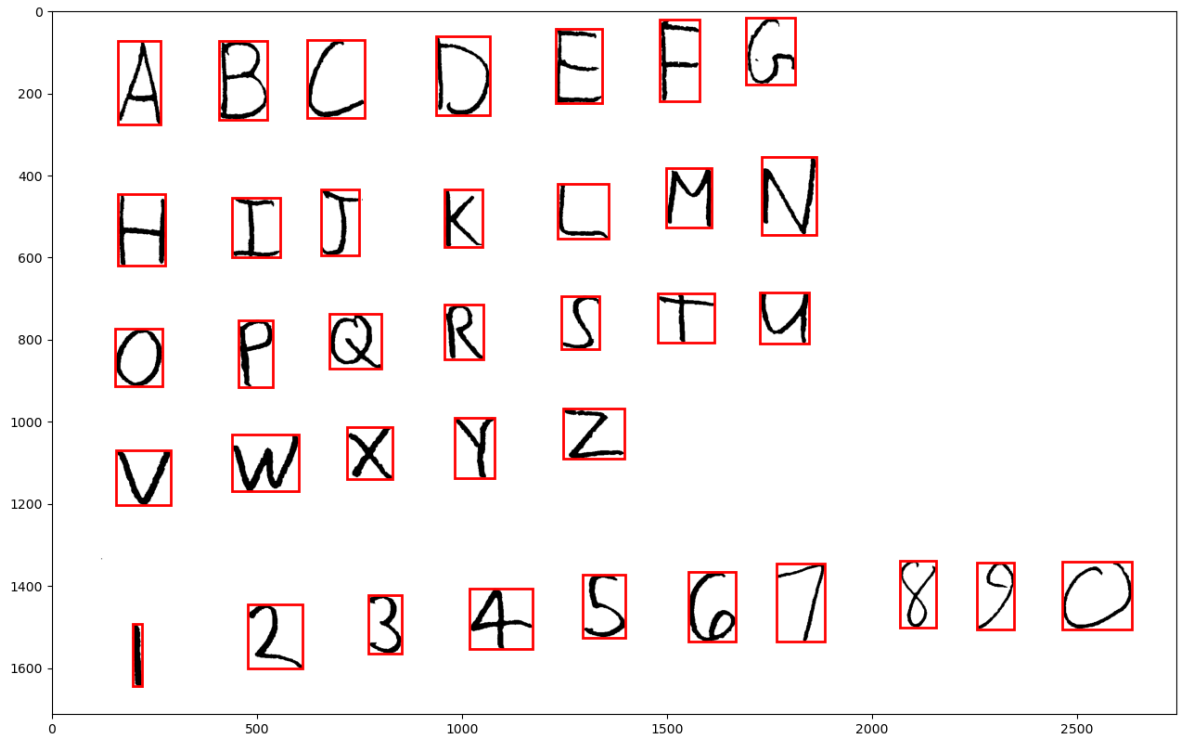

Figure 16: Q43 - Letters with Bounding Boxes

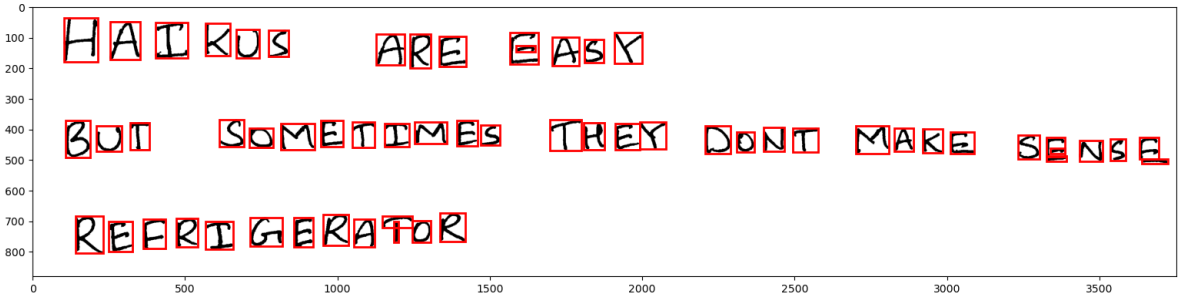

Figure 17: Q43 - Haiku with Bounding Boxes

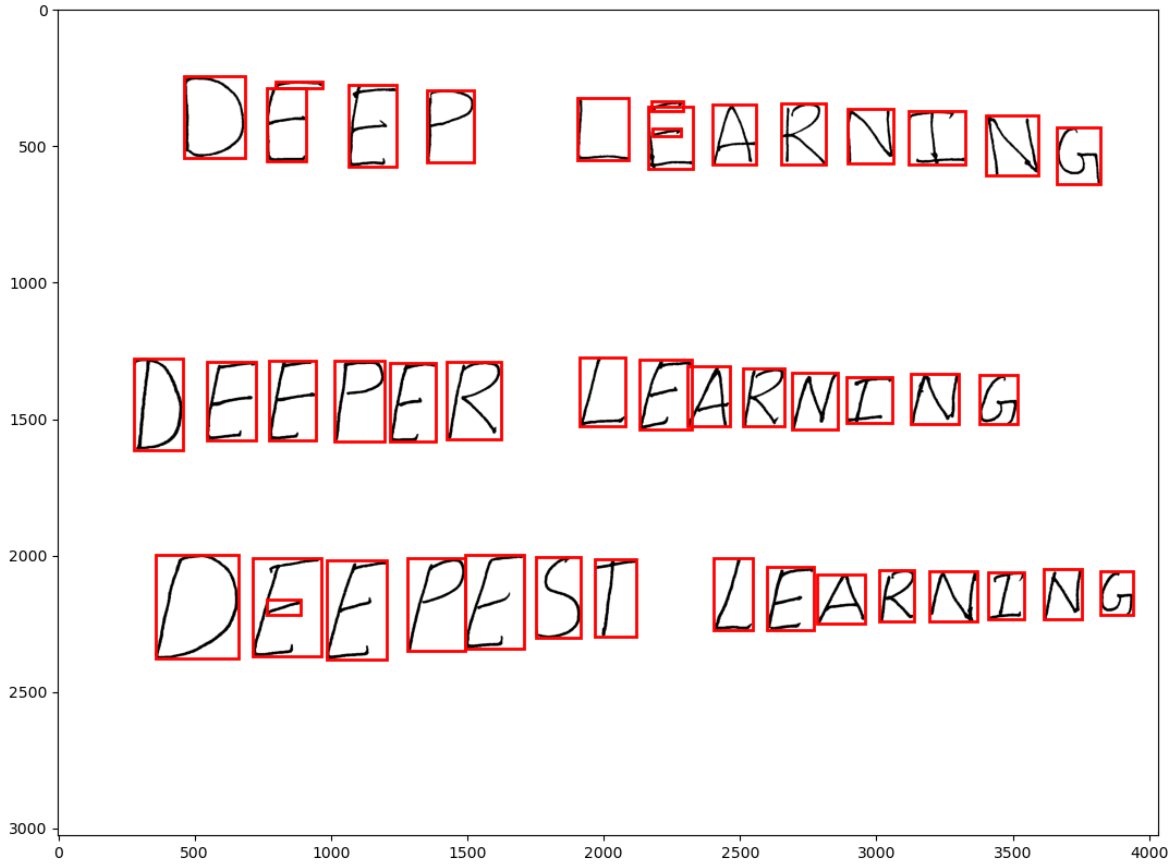

Figure 18: Q43 - Deep with Bounding Boxes

Q4.4

The results of the detected images are as below. It becomes very apparent that the error prone character pairs shown in the confusion matrix manifest themselves in the actual recognition.

Results for list:
T 0 D 0 L I S T
I M A X E A T 0 D 0 L I S T
2 C H E E K 0 F E T H E F I R S T T H F T N G 0 N T 0 D 0 L I S T
3 R E A L I Z E Y 0 U H A V E A L R E A D T E 0 M P L E T E D Z T H I N G S
9 R F F W A R D X 0 U 1 1 S E L F W I T H A N A P

Results for letters:
2 B L D E F G
H I J K L M N
0 P Q K S T U
V W X Y Z
1 Z 3 G S G 7 8 9 Q

Results for haiku:
H A I K U S A R E E H A S X
B U T S Q M E T I M E S T R E X D D N T M A K G S F H M G F 4 4
R E F R I G E R A M 1 0 R

Results for deep:
D C 4 E P L L M H 2 R M I N G
D E 8 P E R L E A R A I N G
D E M 1 P E S T L E A R N I N G

Q5.2

The autoencoder, has a different loss profile compared to that of the question 3. The network seems to do do most of it's learning within the first 20 epochs and then plateaus.

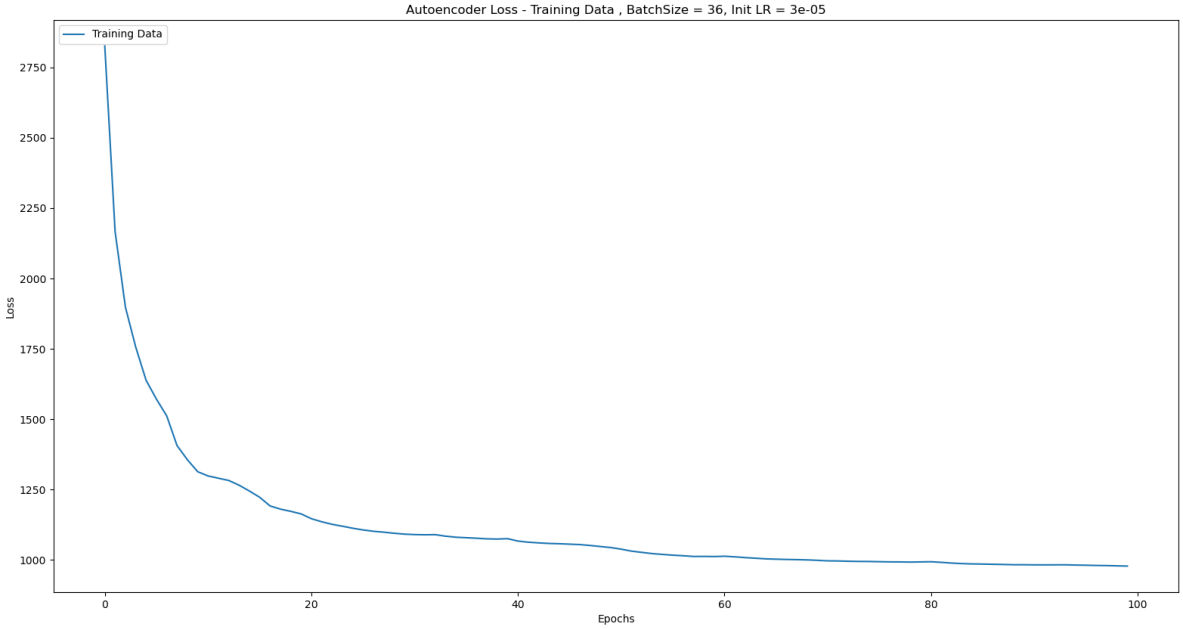

Figure 19: Q52 - Loss of Autoencoder

Q5.3.1

The resulting images from the autoencoder network are essentially lower fidelity images of the input images. This is rational as the network is taking the most primitive features encoded in each class and doing its best to recreate the class in a lower dimensional space. Therefore it is expected that the fidelity of the recreated images will be lower and the network will only prioritize the dominating features per class.

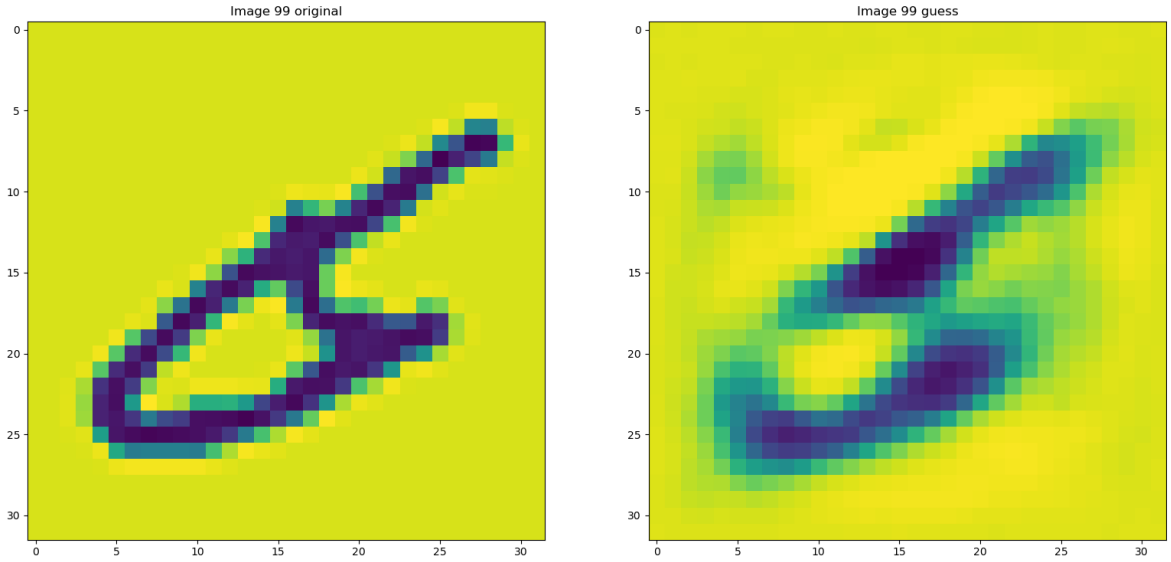

Figure 20: Q53 - Image 99

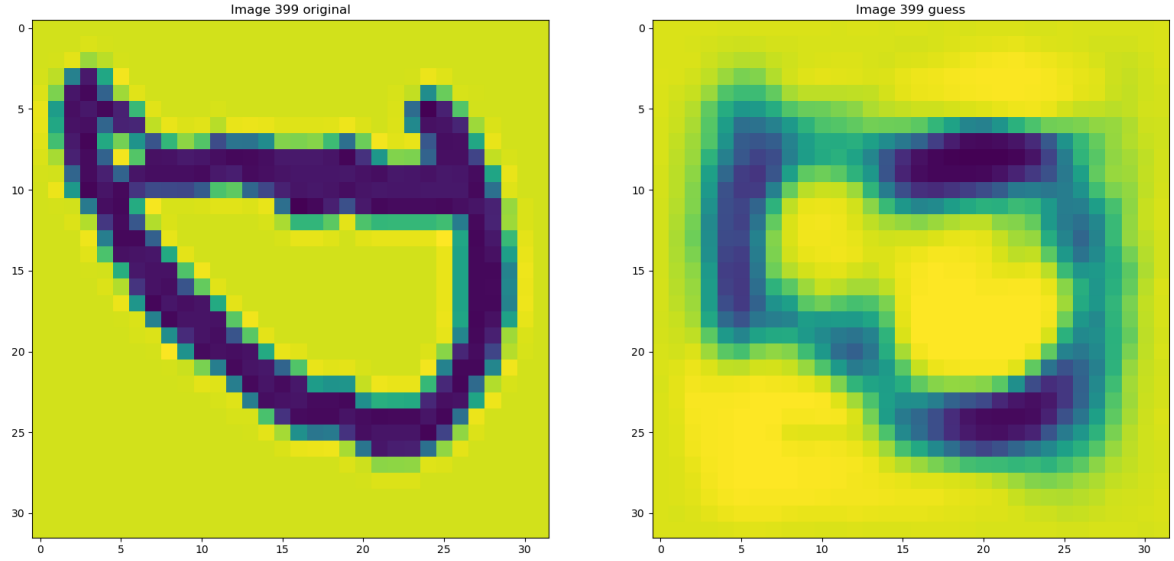

Figure 21: Q53 - Image 399

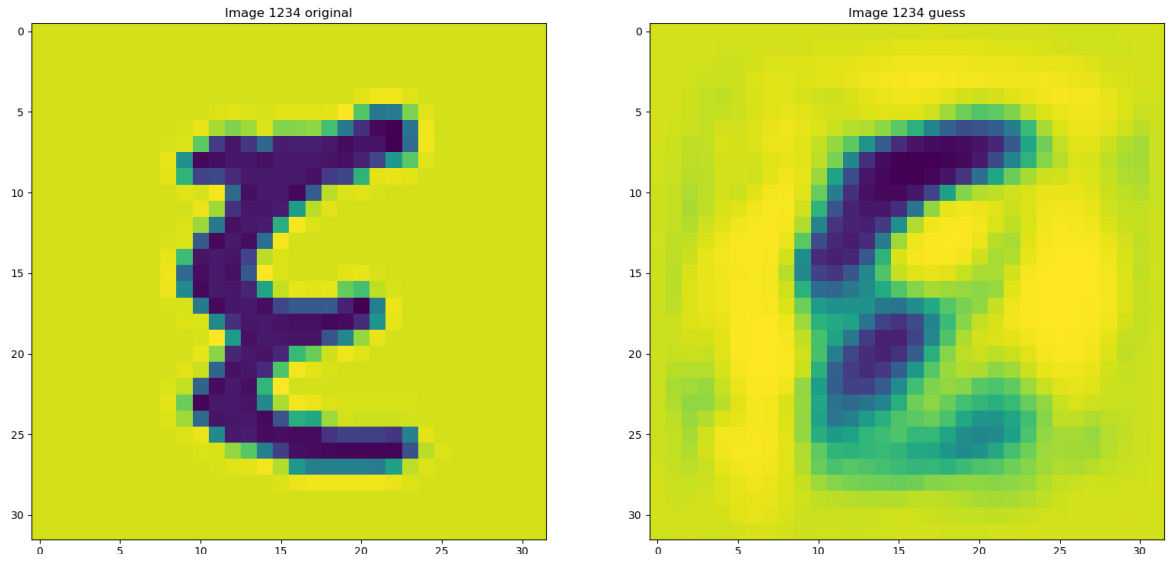

Figure 22: Q53 - Image 1234

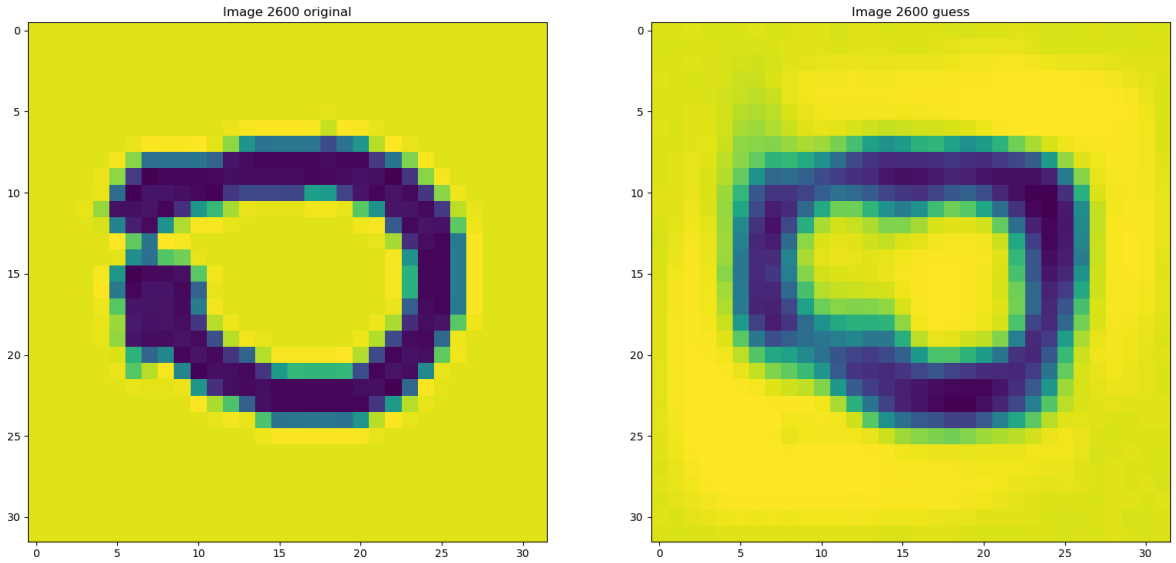

Figure 23: Q53 - Image 2600

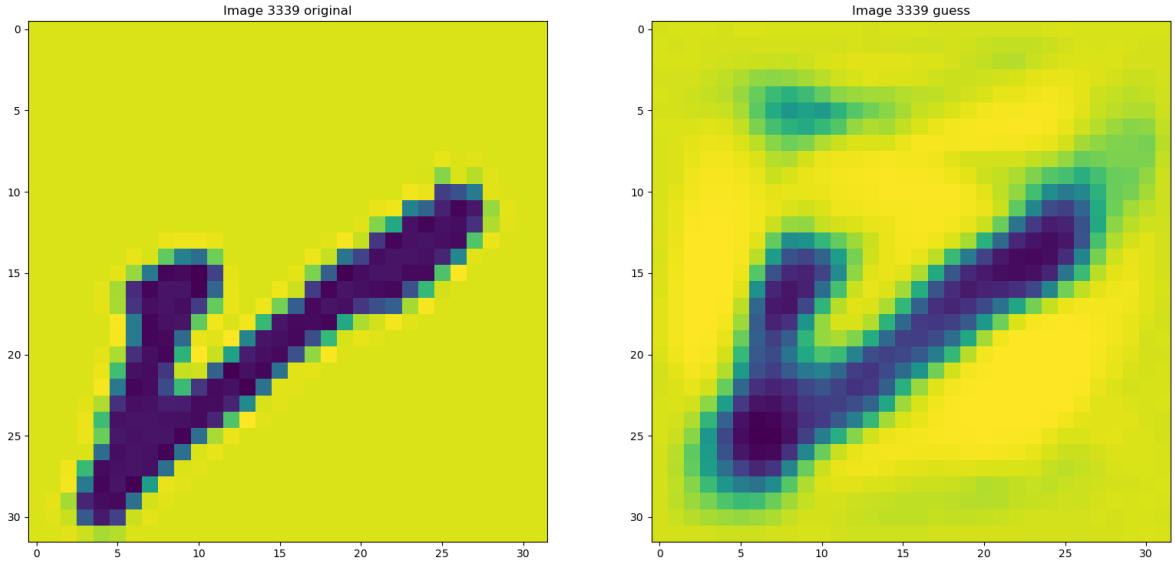

Figure 24: Q53 - Image 3339

Q5.3.2

The highest PSNR ratio I was able to achieve was 15.63%.