Q1.1

For every index i in the vector x, softmax is defined as:

$$softmax(x_i) = \frac{e^{x_i}}{\Sigma_j e^{x_j}}$$

$$softmax(x_i + c) = \frac{e^{x_i + c}}{\Sigma_j e^{x_j + c}}$$

$$softmax(x_i + c) = \frac{e^{x_i} e^{c}}{\Sigma_j e^{x_j} e^{c}}$$

Since c is a constant, we don't have to consider e^c in the summation

$$softmax(x_i + c) = \frac{e^{x_i} e^{c}}{e^{c} \Sigma_j e^{x_j}}$$

$$softmax(x_i + c) = \frac{e^{x_i}}{\Sigma_j e^{x_j}}$$

Thus we have proven that softmax is invariant to translation.

Using c = -max x_i would mean that the largest possible value of $x_i + c$ is zero. This means that the largest possible value of the numerator would be 1, which would eliminate the possibility of overflow. Similarly, at least one term in the denominator will have a value of 1, which rules out the possibility of underflow in the denominator (would lead to a division by zero).

Q1.2

As discussed above, the maximum possible value for $x_i + c$ is zero, which would mean the maximum value of the numerator of softmax is 1 and the minimum asymptotically approaches zero. This means that the range of the output of softmax is (0,1].

One could say that softmax takes an arbitrary real valued vector x and turns it into a probability distribution.

The three steps for softmax are as follows:

(1) $s_i = e^{x_i}$
This step calculates the frequency of the given value by taking its exponential.

(2) $S = \Sigma s_i$
This step calculates the sum of all frequencies.

(3) $softmax(x_i) = \frac{s_i}{S}$
This outputs the normalized frequency of x_i, which is its probability. Performing this normalization over all x_i results in a vector of the probability distribution of x

Q1.3 Show that multi-layer neural networks without a non-linear activation function are equivalent to linear regression.

A pre- and post- activation for the first layer of a neural network are defined as

$$\mathbf{a}^{(1)}(\mathbf{x}) = \mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(t)}$$

$$\mathbf{h}^{(1)}(\mathbf{x}) = \mathbf{g}(\mathbf{a}^{(1)}(\mathbf{x}))$$

Where **g** is a non-linear activation function. In the absence of that linear activation function, then $\mathbf{h}^{(1)}(\mathbf{x})$ would be equal to $\mathbf{a}^{(1)}(\mathbf{x})$. If this is the case, then the output of each successive layer would take the form

$$\mathbf{h}^{(t)}(\mathbf{x}) = \mathbf{a}^{(t)}(\mathbf{x}) = \mathbf{W}^{(t)}\mathbf{h}^{(t-1)} + \mathbf{b}^{(t)}$$

To solve for each of these outputs would be equivalent to solving a linear regression problem.

Q1.4

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

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

$$\frac{\partial \sigma}{\partial x} = \frac{1}{1+e^{-x}}\frac{e^{-x}}{1+e^{-x}}$$

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

$$\frac{\partial \sigma}{\partial x} = \frac{1}{1+e^{-x}}\left(\frac{1+e^{-x}}{1+e^{-x}} - \frac{1}{1+e^{-x}}\right)$$

$$\frac{\partial \sigma}{\partial x} = \frac{1}{1+e^{-x}}\left(1 - \frac{1}{1+e^{-x}}\right)$$

$$\frac{\partial \sigma}{\partial x} = \sigma(x)(1-\sigma(x))$$

So the gradient of the sigmoid function can be written as a function of $\sigma(x)$ without having access to x directly.

Q1.5

$$y_i = \sum_{j=1}^{d} x_j W_{ij} + b_i$$

$$\frac{\partial J}{\partial y} = \delta \in \mathbb{R}^{k\times 1}, \quad W \in \mathbb{R}^{k\times d}, \quad x \in \mathbb{R}^{d\times 1}, \quad b \in \mathbb{R}^{k\times 1}$$

$$\frac{\partial y_i}{\partial W_{ij}} = \frac{\partial}{\partial W_{ij}}(x_1 W_{i1} + x_2 W_{i2+\cdots+} x_d W_{id})$$

In the above case, all terms become zero except for the case when the column index of W equals j. In that case, $\frac{\partial}{\partial W_{ij}}(W_{ij}) = 1$ and

$$\frac{\partial y_i}{\partial W_{ij}} = x_j$$

$$\frac{\partial J}{\partial W_{ij}} = \frac{\partial J}{\partial y_i}\frac{\partial y_i}{\partial W_{ij}} = \sum_{m=1}^{k} \delta_m \frac{\partial y_m}{\partial W_{ij}} = \delta_i x_j$$

$$\boldsymbol{\frac{\partial J}{\partial W} = \delta x^T}$$

$$\frac{\partial y_i}{\partial x_j} = \frac{\partial}{\partial x_j}(x_1 W_{i1} + x_2 W_{i2+\cdots+} x_d W_{id})$$

In the above case, all terms become zero except for the case when the index of x equals j. In that case, $\frac{\partial}{\partial x_j}(x_j) = 1$ and

$$\frac{\partial y_i}{\partial x_j} = W_{ij}$$

$$\frac{\partial J}{\partial x_j} = \frac{\partial J}{\partial y_i}\frac{\partial y_i}{\partial x_j} = \sum_{m=1}^{k} \delta_m \frac{\partial y_m}{\partial x_j} = \delta_i W_j$$

$$\boldsymbol{\frac{\partial J}{\partial x} = W^T \delta}$$

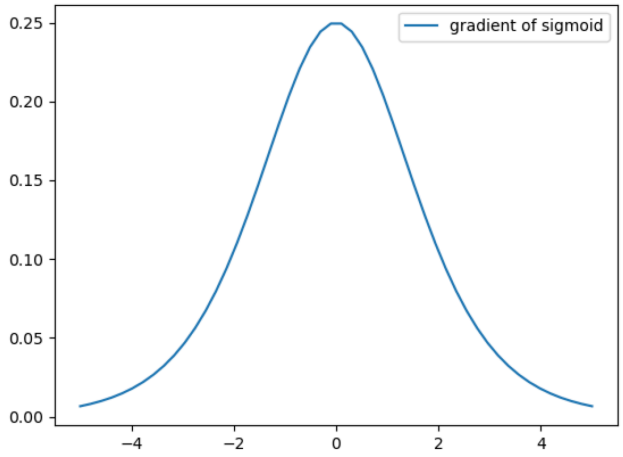
$$\frac{\partial y_i}{\partial b_i} = 1$$

$$\frac{\partial J}{\partial b_i} = \frac{\partial J}{\partial y_i}\frac{\partial y_i}{\partial b_i} = \sum_{m=1}^{k} \delta_m \frac{\partial y_i}{\partial b_i}$$

$$\boldsymbol{\frac{\partial J}{\partial b} = \delta}$$

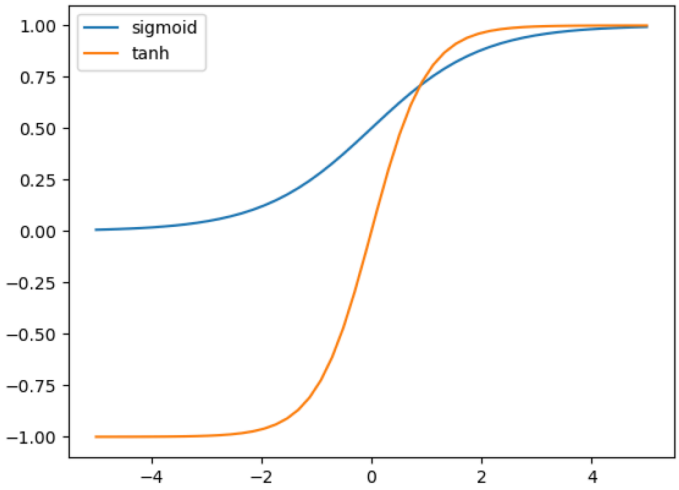
Q1.6

1. As stated in the writeup, the gradient of the activation function scales the backpropogation update. A plot of the gradient of sigmoid is shown below.



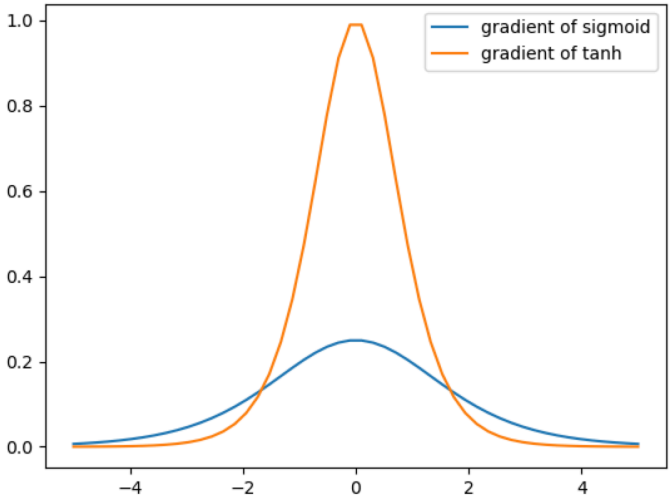
So, at best, the sigmoid activation function will lead to a backpropagation scaling of 0.25. When applied to many successive iterations, this multiplication by such a small fraction will effectively reduce the update factor to zero, hence the term "vanishing gradient."

2. A plot of tanh and sigmoid is shown below.



The output range of tanh is (-1,1) and the output range of sigmoid is (0,1). tanh might be preferable because negative inputs are assigned strongly negative values, inputs close to zero are assigned values near zero, and positive inputs are assigned strongly positive values. This would especially be useful when trying to map between 2 classes. Also, tanh is better because there is less of a vanishing gradient problem, which is discussed further in question 3 below.

3. Below is a plot of the gradients of sigmoid and tanh.



tanh has less of a vanishing gradient problem because for inputs between -2 and 2, the gradient of tanh is much higher than the gradient of sigmoid. So, the propagation update will be scaled by a value closer to one, meaning that the update term will not disappear nearly as quickly as when using sigmoid.

4.

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

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

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

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

$$\tanh(x) = 2\sigma(2x) - 1$$

Q2.1.1

Let's assume that all layers in a network (weights and biases) are zero. Then, going through the math in the appendix of the writeup, we would get a preactivation output of zero, which would mean our postactivation output would be the same value for all inputs. Because these would all be the same output, performing backpropagation would lead to the same gradient calculation and subsequently the same parameter updates. We don't want all parameters to be updated the same way, because each has a specific function. Therefore it is not a good idea to initialize a network with all zeros.

Q2.1.3

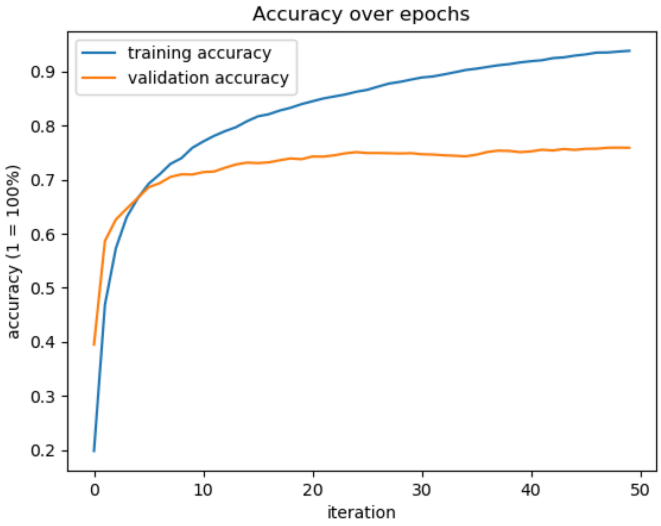
We initialize with random numbers for a similar reason that is discussed in 2.1.1. If all of the values are initialized to be the same, then all parameters will be updated in the same way, and no learning can occur. Randomization is necessary to "break the symmetry" so that each parameter can be updated independently to produce a good network.

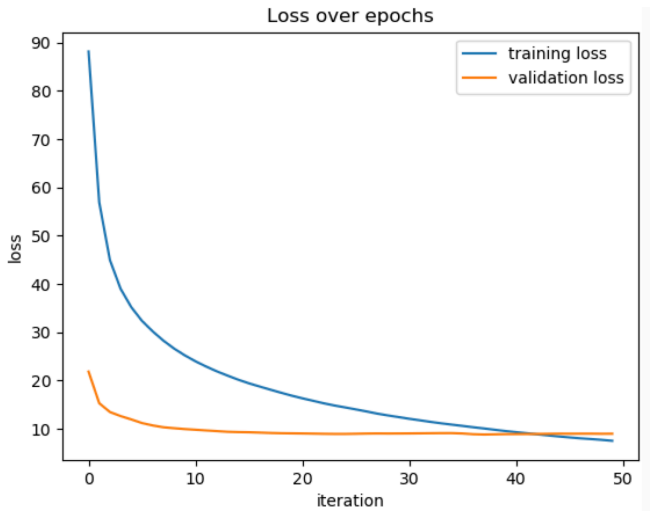
By scaling the initialization based on layer size, we reduce the activation variance between consecutive layers (Fig 6 in the paper), as well as the gradient variance between consecutive layers (Fig 7 in the paper). Because the gradients are all around the same value, it eliminates the possibility of the vanishing gradient problem discussed above.

Q3.1.2

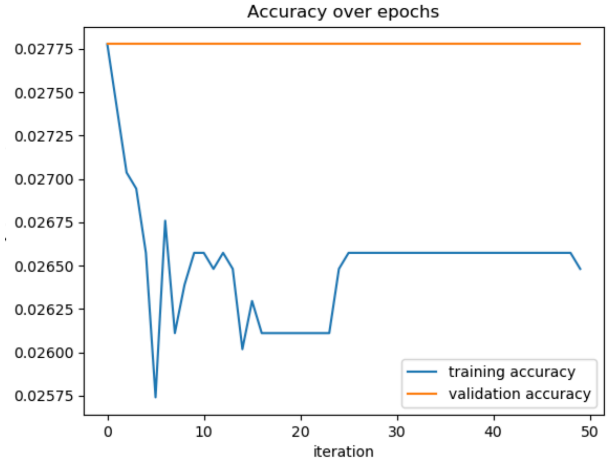
For all 3 examples, I set my batch size to 30 and my number of epochs to 50.

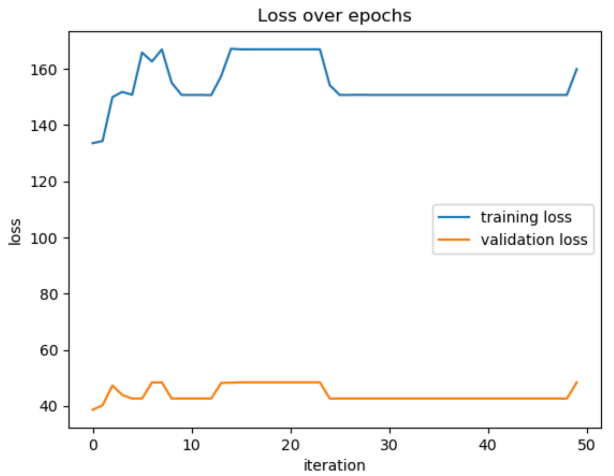
Below are the plots for my best network, with a learning rate of 0.01. In this network, I was able to achieve a validation accuracy of 75.92%.



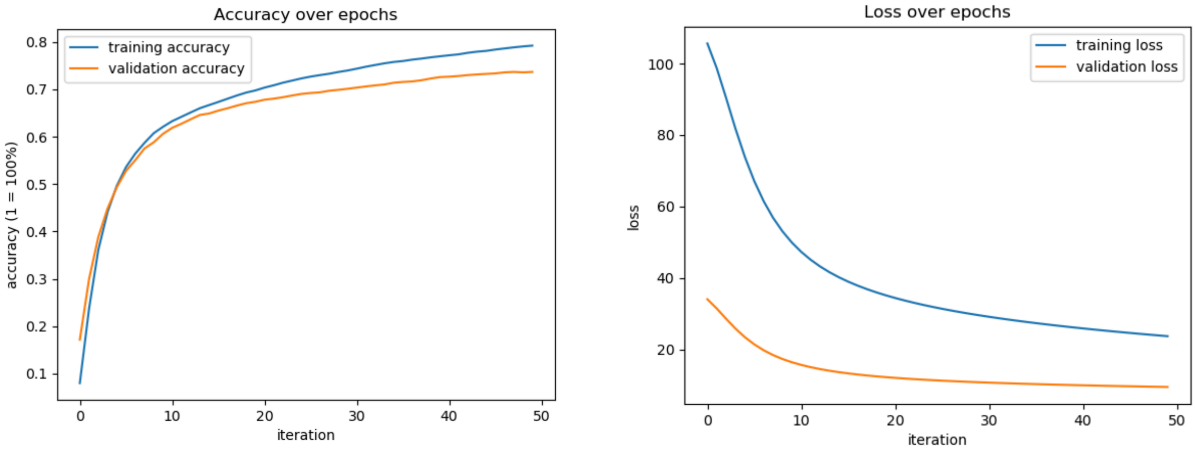


Below are the plots for the network with a learning rate of 0.1, ten times the learning rate of the best network. In this network, I was only able to achieve a validation accuracy of 2.78%.





Below are the plots for a learning rate of 0.001, one-tenth the value of my best learning rate. In this network, the validation accuracy was 73.67%.

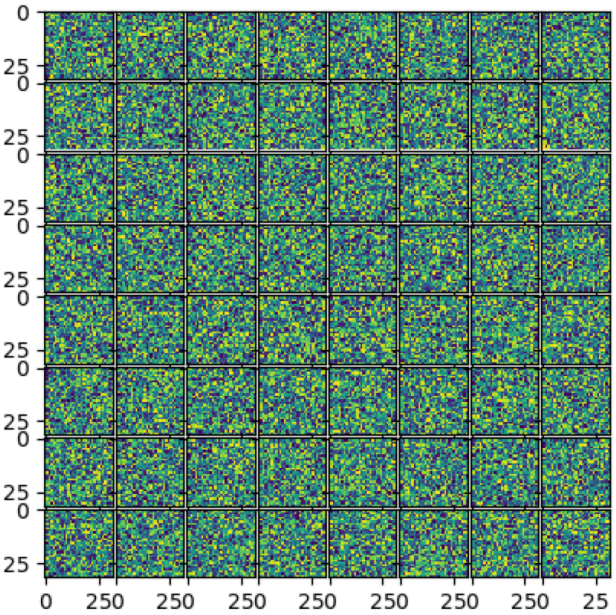


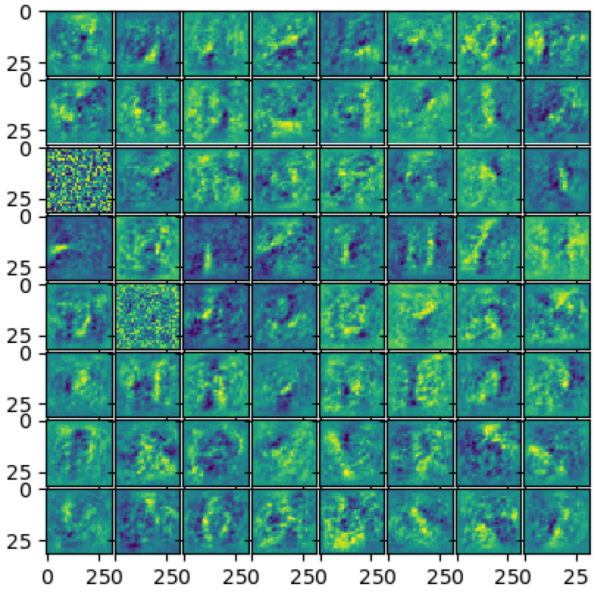
The learning rate is effectively proportional to the size of the step that you take in each iteration during the gradient descent. In theory, a higher learning rate means that you will converge to the optimal point faster. However, as is shown in the case of the network with learning rate of 0.1, if the learning rate is too large, the jumps may be so big that the local minimum is repeatedly jumped over, so the accuracy and loss will never converge to an optimal point. In the case of the very small learning rate (0.001), we see a smoother validation accuracy curve, but the accuracy takes longer to reach the desired value of 75% than the network with learning rate of 0.01.

The accuracy on the test set for the best network (learning rate 0.01) was 76.28%.

Q3.1.3

Shown below are the weights in layer 1 of the network right after initialization, and after training.

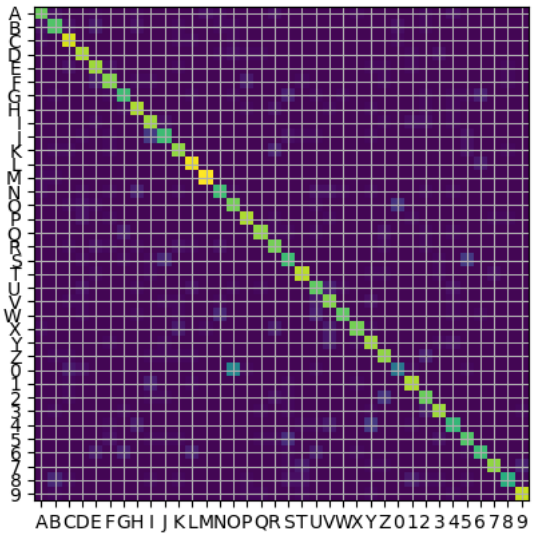




The weights immediately after initialization (shown on the left) are randomly generated from a uniform distribution, so it makes sense that they just show random patterns. After training, the weights (shown on the right) form some noticeable patterns.

Q3.1.4

Below is the confusion matrix for my best model (epochs = 50, batch size = 30, learning rate = 0.01) on the test data. The test accuracy was 76.28%.



As we can see, most of the classes were classified correctly. The most common mixed up classes were the letter “O” and the digit “0”. This is not surprising, as even humans mix these two characters up very easily. Two more classes that were often misclassified were the letter “Z” and the digit “2”. Again, these two have very similar shapes, so it is understandable why the model often had trouble classifying between the two. This model proves the usefulness of writing the number “0” like “Ø”, and writing the letter “Z” like “Ƶ”.

Q4.1

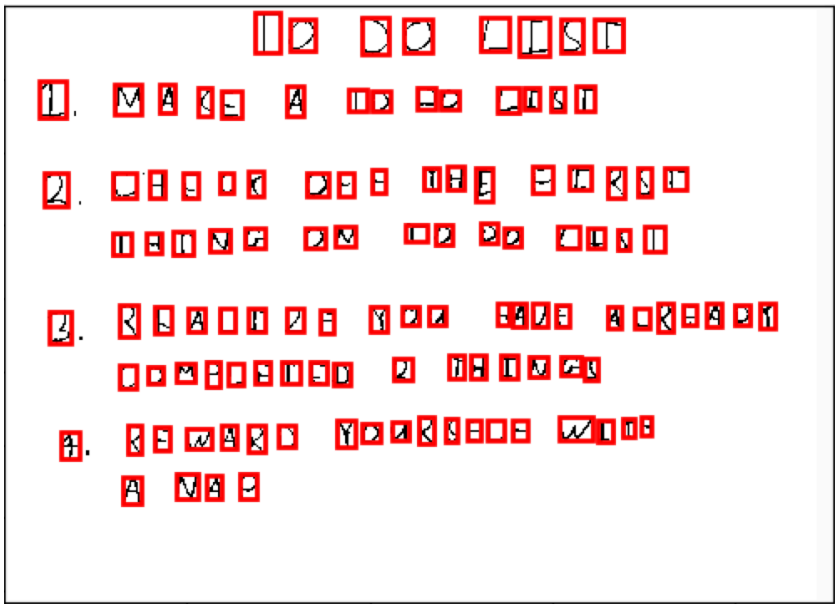
One big assumption that this model makes is that all letters are connected. The labeling method we use gives adjoining regions the same label. This would be a problem if letters were disconnected, like the two examples shown below. The letter detector would find four "letters" in this image because there are 4 continuous black regions.

By the same token, this method would not work if letters were connected to each other. Oftentimes handwriting can be messy, and that means that characters are rarely completely separated from adjacent characters. The following example would create some problems with our classifier. The letter detector we are using would think that this example only contains 3 letters, since there are only 3 continuous black regions.

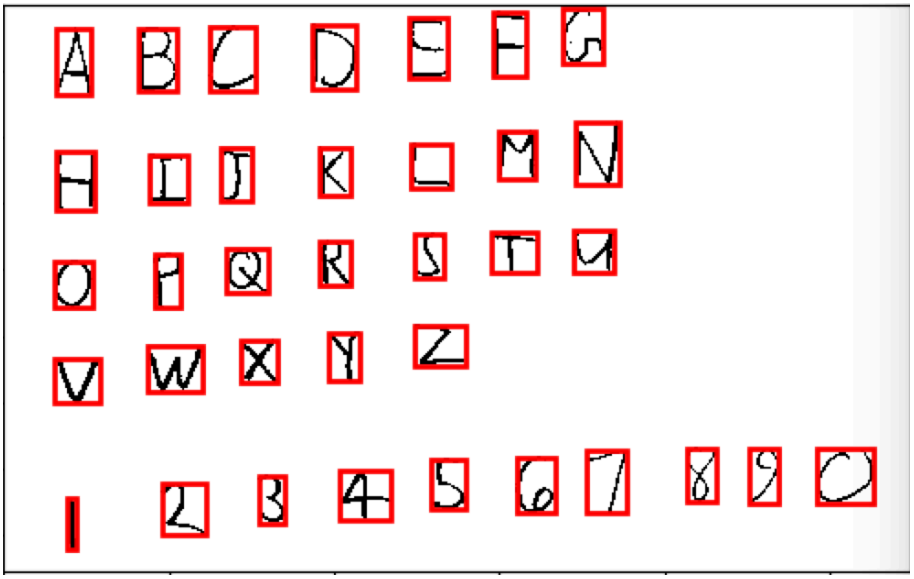
Q4.3

Below are all of the images with bounding boxes over the letters.

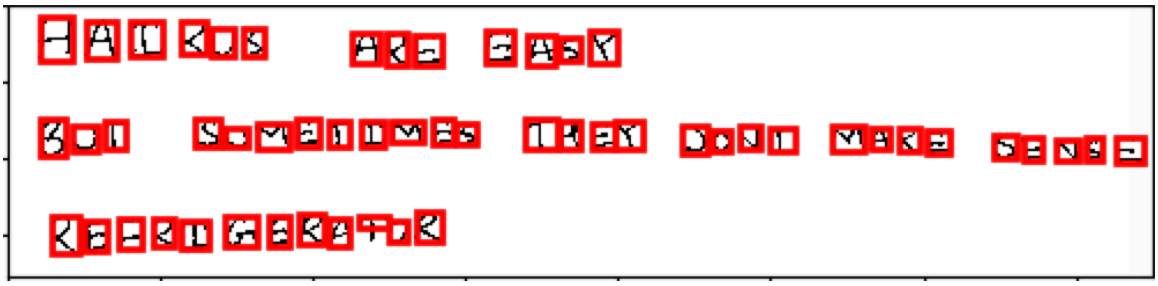
01_list.jpg



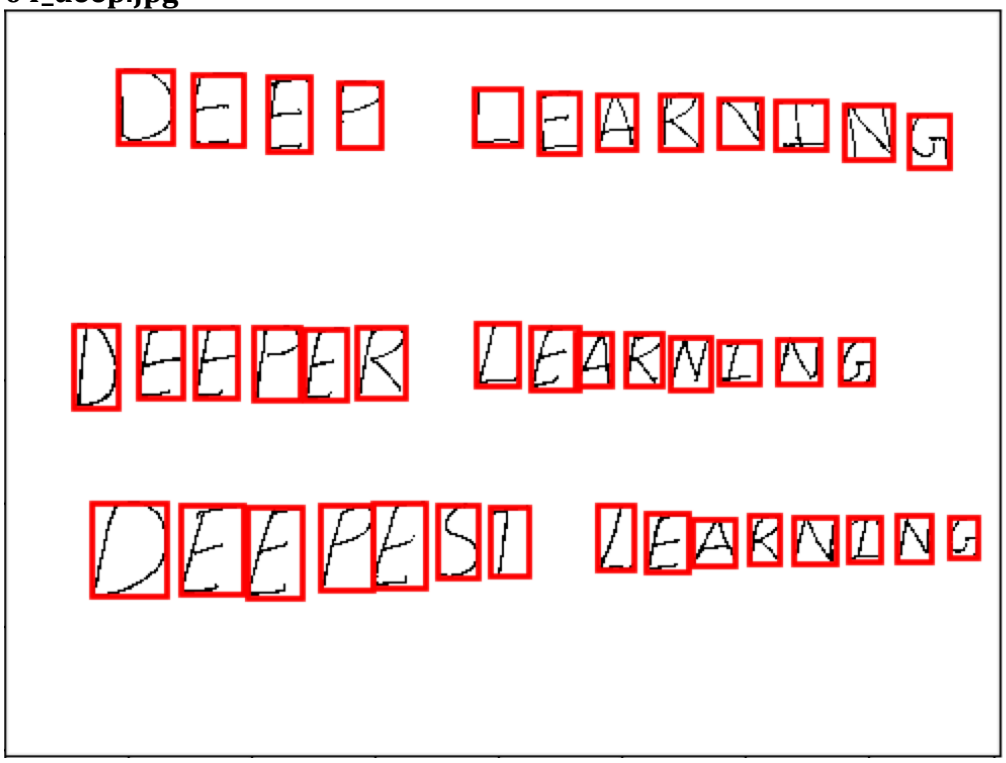
02_letters.jpg



03_haiku.jpg



04_deep.jpg



Q4.4

Below is the predicted output, actual output, and accuracy of the text in each image.

01_list.jpg

```
NN OUTPUT:
TO DO LIST
Z MAKE A T0 2Q LIST
2 CHECK QFF THE FIRST
THING 0H T0 D0 LIST
3 RLALIZE Y0U HAVE ALREADY
C0MPLETED 2 YHINGS
4 REWARD Y0URSELF WITH
A NAP

EXPECTED OUTPUT:
TO DO LIST
1 MAKE A TO DO LIST
2 CHECK OFF THE FIRST
THING ON TO DO LIST
3 REALIZE YOU HAVE ALREADY
COMPLETED 2 THINGS
4 REWARD YOURSELF WITH
A NAP

01_list.jpg accuracy: 0.8782608695652174
```

02_letters.jpg

```
NN OUTPUT:
ABCDGFG
HIJKLMN
QFQRSTU
VWXYZ
QZ3GSG7X9Q

EXPECTED OUTPUT:
ABCDEFG
HIJKLMN
OPQRSTU
VWXYZ
1234567890

02_letters.jpg accuracy: 0.7222222222222222
```

03_haiku.jpg

```
NN OUTPUT:
HAIKUS ARE HASY
BUT SQMETIMES TBEY DONT MAK2 BENGE
RBGRIGBRAMQR

EXPECTED OUTPUT:
HAIKUS ARE EASY
BUT SOMETIMES THEY DONT MAKE SENSE
REFRIGERATOR

03_haiku.jpg accuracy: 0.7962962962962963
```

04_deep.jpg

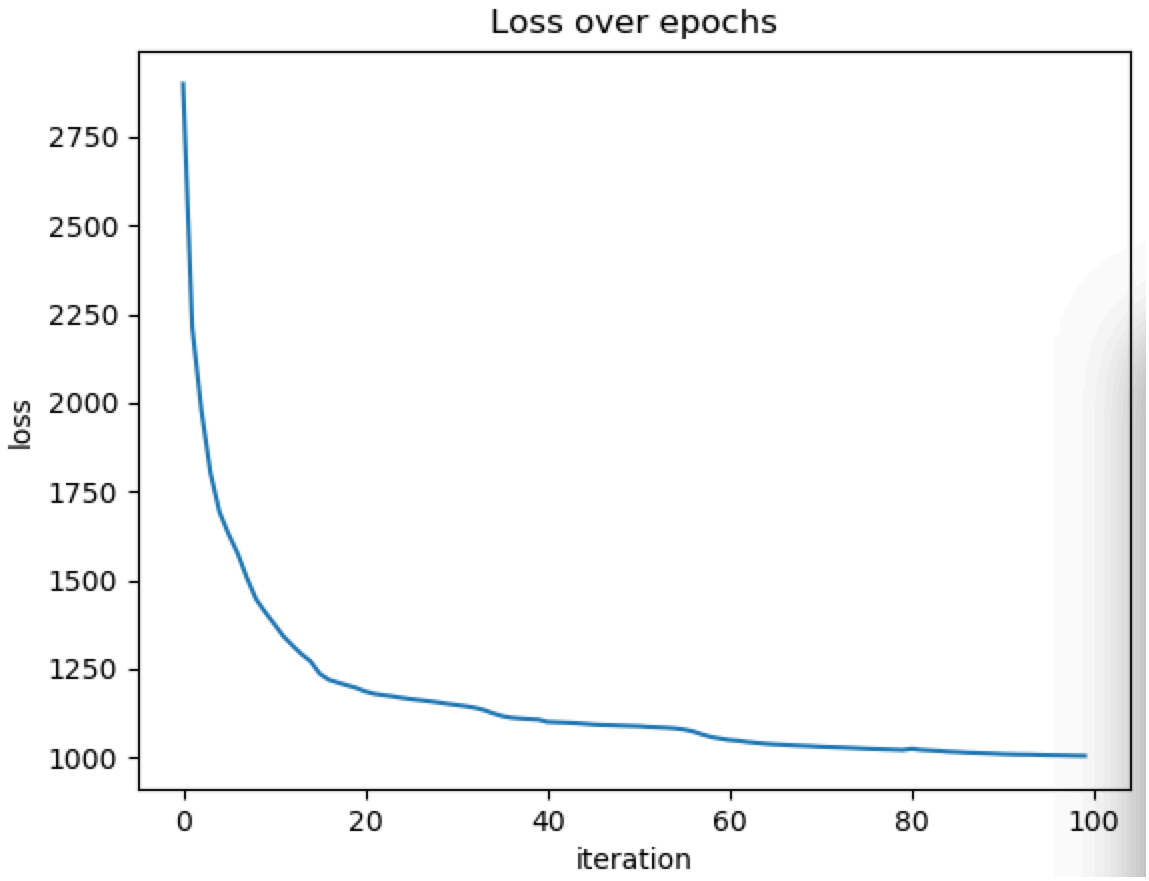
```
NN OUTPUT:
CHHF LHAKMINU
DGHTER LEARHING
SEKFHSI LEARNING

EXPECTED OUTPUT:
DEEP LEARNING
DEEPER LEARNING
DEEPEST LEARNING

04_deep.jpg accuracy: 0.5853658536585366
```

Q5.2

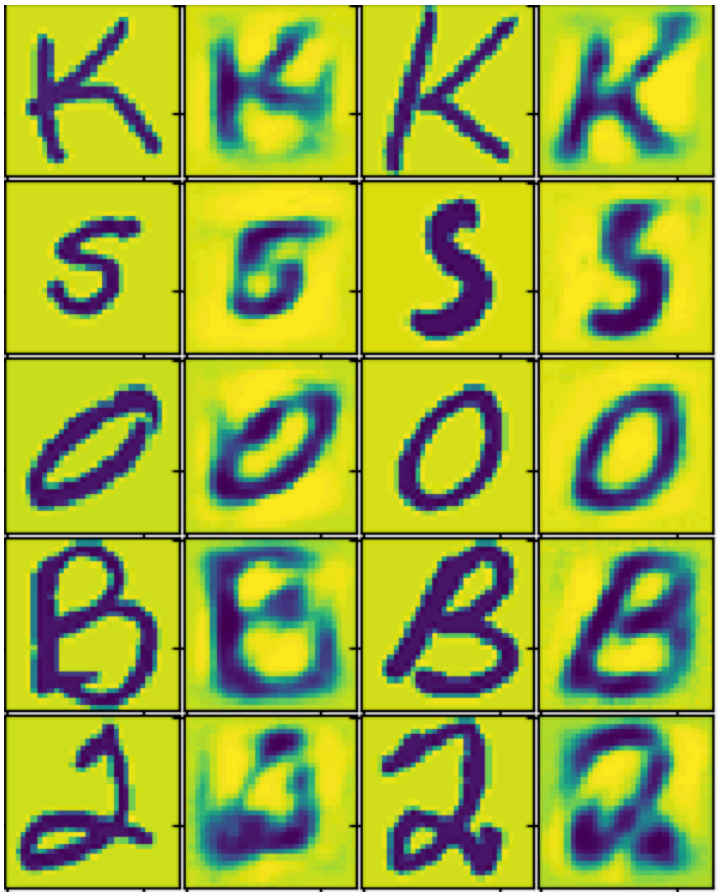
Below is the graph of loss (mean squared error) in the training set over 100 epochs.



We see the loss decrease very rapidly for the first 20 epochs, and then we see the loss function decrease asymptotically from there. This shows that we have found a minimum in our gradient descent.

Q5.3.1

Below are the images of 5 character classes before and after going through the autoencoder.



We can see that the autoencoder does a decent job of reconstructing the original images. The characters are a little more blurry/noisy than their originals, but it is still very clear what the original character was in most cases.

Q5.3.2

The average Peak Signal-to-Noise Ratio across all validation images is **15.9644**

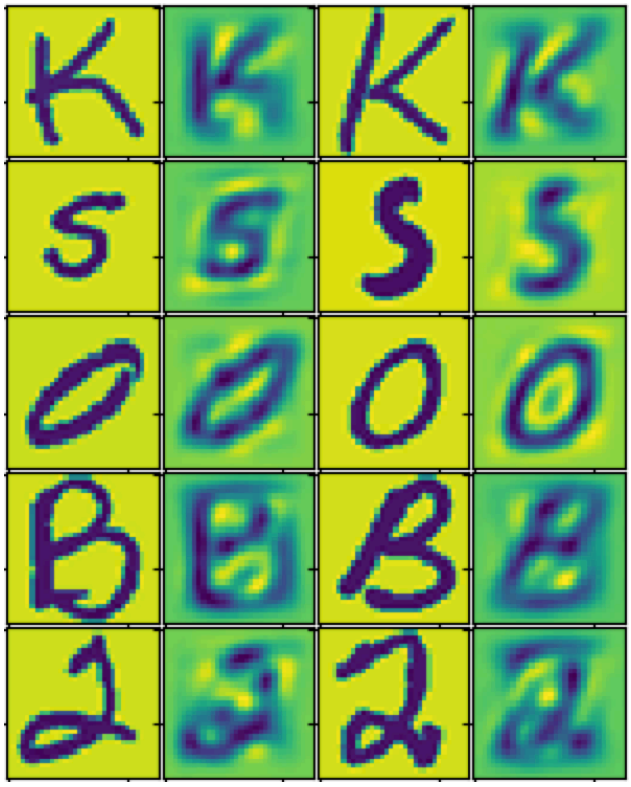
Q6.1

The size of the projection matrix is 1024 x 32. The rank of the matrix is 32, because all of the columns are linearly independent.

Multiplying our image data by this matrix reduces the dimensionality of the image from 1024 dimensions to 32 dimensions. Then multiplying that product by the transpose of the projection matrix projects the image data back to its initial dimensions.

Q6.2

Below are the same 5 letters visualized after reconstruction using Principal Component Analysis

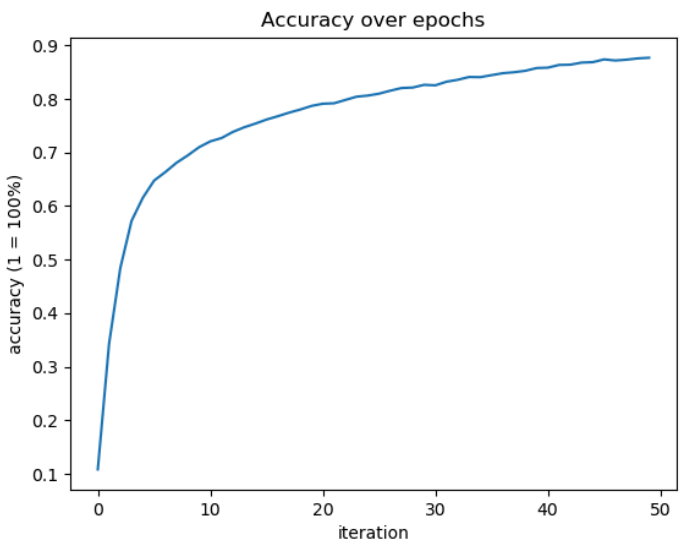


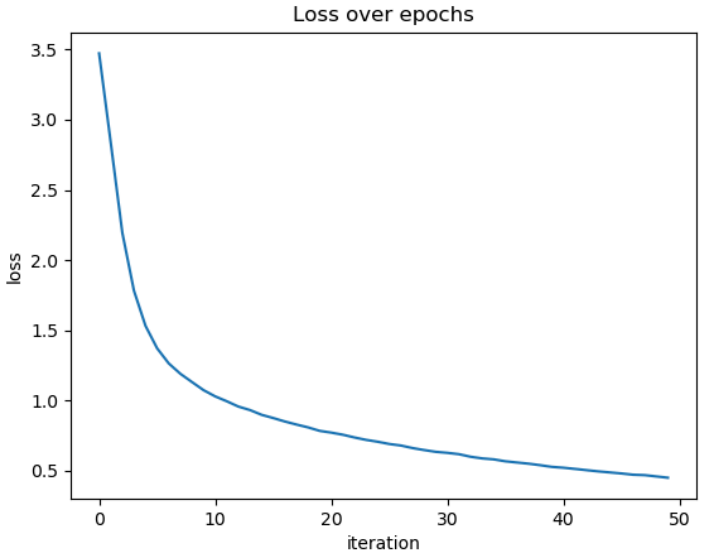
Here we see that the actual letters themselves look much clearer after PCA aas compared to the autoencoder. The drawback is that there is much more noise in the background after PCA. This makes sense because PCA only uses the most prominent information and effectively discards the rest. In this case, the most prominent information is the character in the picture, so that is reconstructed accurately.

We can prove that PCA in this case is a better reconstruction method because the PSNR is greater than the PSNR for the autoencoder. The PSNR for PCA is **16.3482**

Q7.1.1

Below are the graphs for training accuracy and loss for a fully connected network on NIST36 dataset.

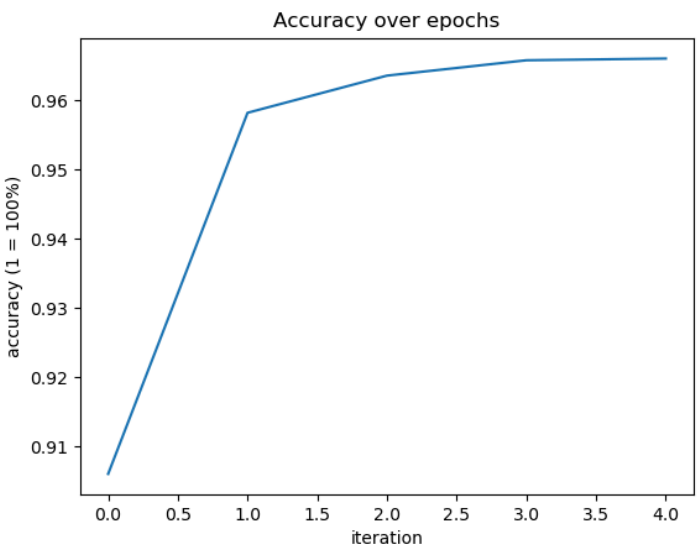


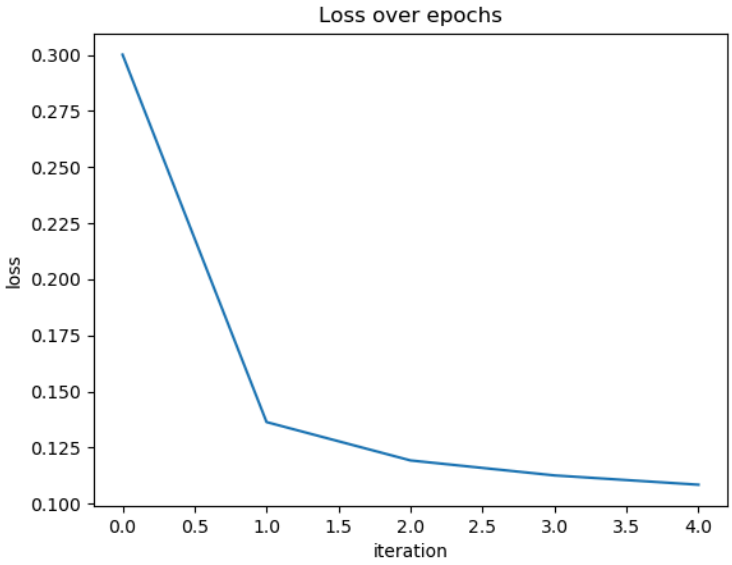


The test accuracy for this network was **79.67%**

Q7.1.2

Below are the graphs for training accuracy and loss for my convolutional neural network on the MNIST dataset.

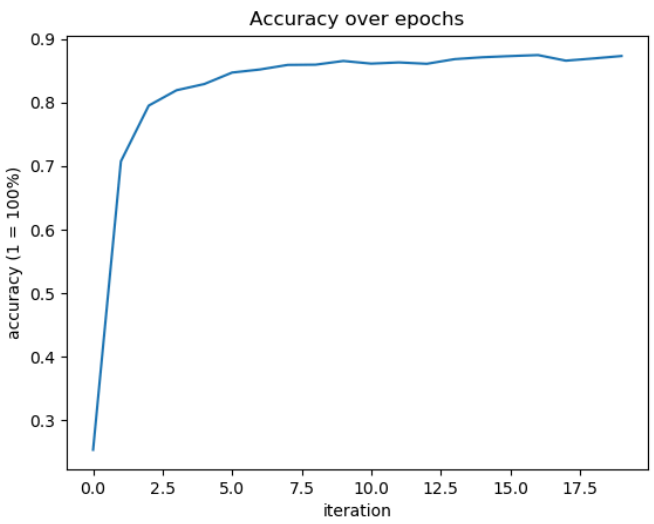


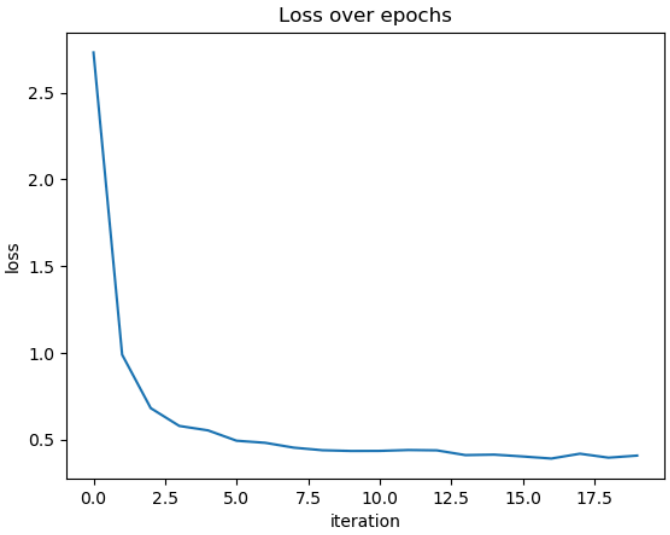


This network was only trained for 5 epochs because it was slow to train, and the training accuracy reached a very high value very quickly. Even after 5 epochs, the test accuracy for this network was **97.12%**

Q7.1.3

Below are the training accuracy and loss graphs for my convolutional neural network on the NIST36 dataset.

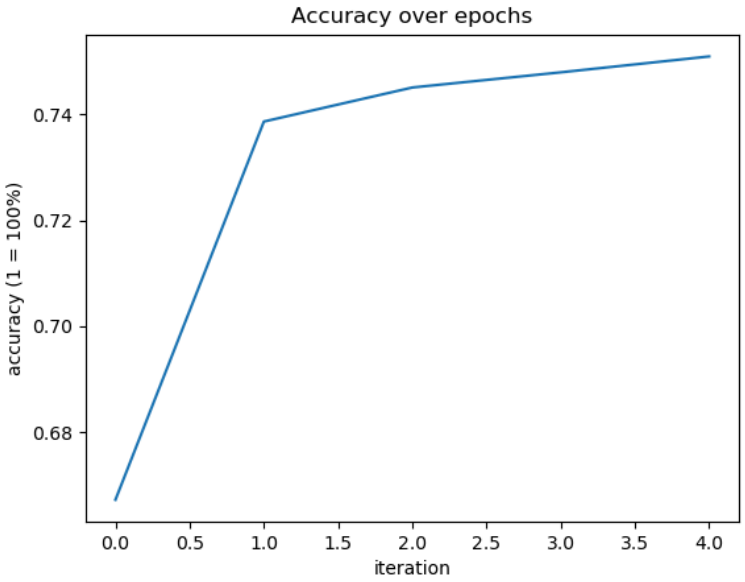


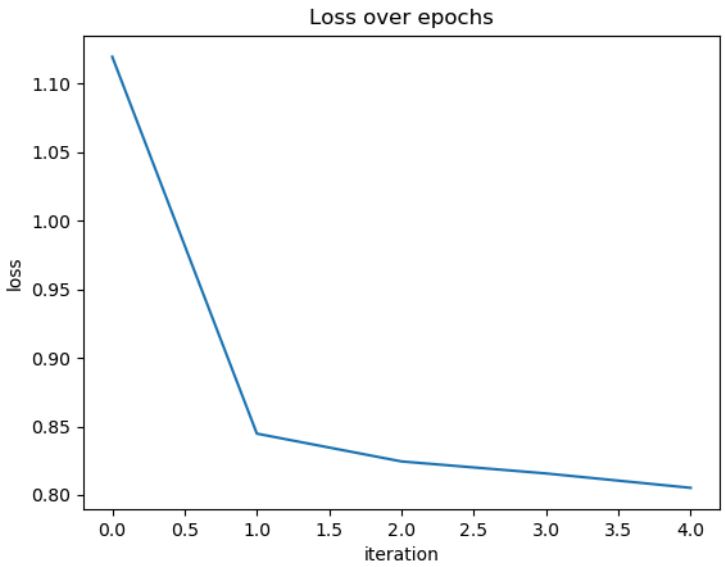


I only trained this network for 20 epochs and got a test accuracy of **85.56%**

Q7.1.4

Below are the training accuracy and loss graphs for my convolutional neural network on the EMNIST dataset.





This network was only trained for 5 epochs due to the long compute time. Even with so few training iterations, the test accuracy was **74.68%**
Below is the predicted output, actual output and accuracy of the text in each image.

01_list.jpg

```
NN OUTPUT:
T0 D0 LIST
I MAKE A T0 DQ LIST
2 CHECK DFF THE FIqST
IHIMG 0N TO DO LFST
3 RFALIZE YOU HAVE hLREADY
COMPLFTrD 2 THINGS
4 RFWARD YOURSEUF WTTH
A NAP

EXPECTED OUTPUT:
TO DO LIST
1 MAKE A TO DO LIST
2 CHECK OFF THE FIRST
THING ON TO DO LIST
3 REALIZE YOU HAVE ALREADY
COMPLETED 2 THINGS
4 REWARD YOURSELF WITH
A NAP

01_list.jpg accuracy: 0.8434782608695652
```

02_letters.jpg

```
NN OUTPUT:
ABCDFFG
HIJKLYN
0PQRSTW
VWXYZ
rZ3fSG789Q

EXPECTED OUTPUT:
ABCDEFG
HIJKLMN
OPQRSTU
VWXYZ
1234567890

02_letters.jpg accuracy: 0.7222222222222222
```

03_haiku.jpg

```
NN OUTPUT:
HAIKUS ARE EASX
3JT SOMETIMES THEY DONT MAKE SEMSE
REFRIGERArOR

EXPECTED OUTPUT:
HAIKUS ARE EASY
BUT SOMETIMES THEY DONT MAKE SENSE
REFRIGERATOR

03_haiku.jpg accuracy: 0.9074074074074074
```

04_deep.jpg

```
NN OUTPUT:
IEtP LtAKMLNG
UEEYEK LEAKNIWG
LtErESL LEAKMIMG

EXPECTED OUTPUT:
DEEP LEARNING
DEEPER LEARNING
DEEPEST LEARNING

04_deep.jpg accuracy: 0.5609756097560976
```