Computer Vision: 16720-A

Kinjal Jain
Homework 5

April 21, 2020

1 Theory

Q1.1

$$softmax(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}} softmax(x_i + c) = \frac{e^{x_i+c}}{\sum_j e^{x_j+c}} \tag{1}$$

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

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

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

$$softmax(x_i + c) = softmax(x_i) \tag{5}$$

Therefore, $softmax(x + c) = softmax(x)\ \forall x \in R$.
Numbers are represented using 32 bits or 64 bits. So there is a limit on the value any number can take and also drawbacks in precision. If $c == 0$, the value of numerator can range from $[0, \infty)$. With $c = -max(x_i)$, the range will be $[0, 1]$ Thus, by restricting the value to be within the limits by subtracting with $c = -max(x_i)$, we can ensure numerical stability during the calculations by avoiding overflow errors.

Q1.2

- Range of the softmax is given by [0, 1] i.e. $Softmax(x) \in [0, 1]$
  Softmax denotes the probability of an event x_i. Sum over all elements (x_i) is equal to 1.

- Softmax takes and arbitrary real values vector x and turns it into a probability distribution. It assigns individual probability to all the events $x_i \in x$.

- 1. x_i can be the output of any non-linear function. The values can be positive or negative. $s_i = e^{x_i}$ converts all values to positive and so take exp value of all x_i.

  2. $S = \sum s_i$ denotes the sum of all values of x_i belonging to the class which is calculated by summing overall all positive values obtained in the previous step. This can be greater than 1 since the exponential values for events x_i are not bounded.

  3. $softmax(x_i) = \frac{s_i}{S}$ converts all values to probability distribution for that particular class by taking the ratio. The sum of all $softmax(x_i)$ values is equal to 1. Now, each value indicates the probability of an individual event.

Q1.3

The linear activation function on first layer can be represented as,

$$h_1 = a_1(W_1^T x_1 + b_1) + c_1 \tag{6}$$

where W_1 represents the weights corresponding to this layer, b_1 represents the bias corresponding to this layer, b_1 and c_1 represent the coefficients of linear activation
Without a non-linear layer, we pass the output of this layer to another linear layer. This can be represented as,

$$h_2 = W_2^T h_1 + b_2 \tag{7}$$

By substituting the value of h_1, we can simplify this as,

$$h_2 = W_2^T(a_1(W_1^T x_1 + b_1) + c_1) + b_2 \tag{8}$$

$$h_2 = (W_2^T.a_1.W_1^T)x_1 + (W_2^T.a_1.b_1 + W_2^T.c_1 + b_2) \tag{9}$$

We can write the above in the form as below,

$$h_2 = W_3^T + b_3 \tag{10}$$

We can say that this equation [10] is of the form [6] which represents a linear regression. We can see that this combination of 2 layers is also linear. This logic can be extended to any number of layers and so, we can say that without a non-linear layer, multi-layer neural networks are equivalent to linear regression.

Q1.4

Sigmoid activation function is given by:

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{11}$$

Then the derivative of [11] will be:

$$\frac{d}{dx}\sigma(x) = \frac{e^{-x}}{(1 + e^{-x})^2} \tag{12}$$

$$\frac{d}{dx}\sigma(x) = \frac{1}{(1 + e^{-x})}\frac{(e^{-x})}{(1 + e^{-x})} \tag{13}$$

$$\frac{d}{dx}\sigma(x) = \sigma(x)(1 - \sigma(x)) \tag{14}$$

So, we can represent the gradient of sigmoid as a function of sigmoid itself (without accessing x).

Q1.5

Given,

$$x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_d \end{bmatrix}$$

$$b = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_k \end{bmatrix}$$

$$w = \begin{bmatrix} w_{11} & w_{12} \dots w_{1k} \\ w_{21} & w_{22} \dots w_{2k} \\ \vdots & \vdots \\ w_{d1} & w_{d2} \dots w_{dk} \end{bmatrix}$$

and

$$y = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_k \end{bmatrix}$$

Given $y = w^T x + b$ and loss $= J$
We can write,

$$\frac{\partial J}{\partial y} = \begin{bmatrix} \delta_1 \\ \delta_2 \\ \vdots \\ \delta_3 \end{bmatrix}$$

All the outputs y_i can be represented by,

$$y_1 = x_1 w_{11} + ... + x_k w_{k1} + b_1 \tag{15}$$
$$y_2 = x_1 w_{12} + ... + x_k w_{k2} + b_2 \tag{16}$$
$$\tag{17}$$

and so on...

Then, the partial derivative of loss (J) with respect to the matrix w.

$$\frac{\partial J}{\partial w_{11}} = \frac{\partial J}{\partial y_1} \frac{\partial y_1}{\partial w_{11}} \tag{18}$$
$$\frac{\partial J}{\partial w_{11}} = \frac{\partial J}{\partial y_1} . x_1 \tag{19}$$

Similarly, $\frac{\partial J}{\partial w_{12}}$ will be,

$$\frac{\partial J}{\partial w_{12}} = \frac{\partial J}{\partial y_2}\frac{\partial y_2}{\partial w_{12}} \tag{20}$$

$$\frac{\partial J}{\partial w_{12}} = \frac{\partial J}{\partial y_2}.x_1 \tag{21}$$

To generalize, we can say,

$$\frac{\partial J}{\partial w} = \frac{\partial J}{\partial y}x^T \tag{22}$$

which in matrix form can be written as,

$$\frac{\partial J}{\partial w} = \begin{bmatrix} \frac{\partial J}{\partial y_1}x_1 & \frac{\partial J}{\partial y_2}x_1 \dots \frac{\partial J}{\partial y_k}x_1 \\ \frac{\partial J}{\partial y_1}x_2 & \frac{\partial J}{\partial y_2}x_2 \dots \frac{\partial J}{\partial y_k}x_2 \\ \vdots & \vdots \\ \frac{\partial J}{\partial y_1}x_d & \frac{\partial J}{\partial y_2}x_d \dots \frac{\partial J}{\partial y_k}x_d \end{bmatrix}$$

which is same as,

$$\frac{\partial J}{\partial w} = \begin{bmatrix} \delta_1 x_1 & \delta_2 x_1 \dots \delta_k x_1 \\ \delta_1 x_2 & \delta_2 x_2 \dots \delta_k x_2 \\ \vdots & \vdots \\ \delta_1 x_d & \delta_2 x_d \dots \delta_k x_d \end{bmatrix}$$

To calculate $\frac{\partial J}{\partial x}$, we need to compute the partial derivatives of J with respect to the vector x. So, for a scalar we can say that,

$$\frac{\partial J}{\partial x_1} = \frac{\partial J}{\partial y}\frac{\partial y}{\partial x_1} \tag{23}$$

$$\tag{24}$$

Then,

$$\frac{\partial J}{\partial x_1} = \frac{\partial J}{\partial y_1}\frac{\partial y_1}{\partial x_1} + \frac{\partial J}{\partial y_2}\frac{\partial y_2}{\partial x_1}... + \frac{\partial J}{\partial y_k}\frac{\partial y_k}{\partial x_1} \tag{25}$$

$$\frac{\partial J}{\partial x_1} = \frac{\partial J}{\partial y_1}w_{11} + \frac{\partial J}{\partial y_2}w_{12}... + \frac{\partial J}{\partial y_k}w_{1k} \tag{26}$$

which can be written in the matrix form as,

$$\frac{\partial J}{\partial x} = \begin{bmatrix} \frac{\partial J}{\partial y_1}w_{11} + \frac{\partial J}{\partial y_2}w_{12}... + \frac{\partial J}{\partial y_k}w_{1k} \\ \frac{\partial J}{\partial y_1}w_{21} + \frac{\partial J}{\partial y_2}w_{22}... + \frac{\partial J}{\partial y_k}w_{2k} \\ \vdots \\ \frac{\partial J}{\partial y_1}w_{d1} + \frac{\partial J}{\partial y_2}w_{d2}... + \frac{\partial J}{\partial y_k}w_{dk} \end{bmatrix}$$

which is same as,

$$\begin{bmatrix} \frac{\partial J}{\partial x_1} \\ \frac{\partial J}{\partial x_2} \\ \vdots \\ \frac{\partial J}{\partial x_d} \end{bmatrix} = \begin{bmatrix} w_{11} & w_{12} & \dots & w_{1k} \\ w_{21} & w_{22} & \dots & w_{2k} \\ \vdots & & & \\ w_{d1} & w_{d2} & \dots & w_{dk} \end{bmatrix} \begin{bmatrix} \frac{\partial J}{\partial y_1} \\ \frac{\partial J}{\partial y_2} \\ \vdots \\ \frac{\partial J}{\partial y_k} \end{bmatrix} \tag{27}$$

To simplify further, we can write it as,

$$\begin{bmatrix} \frac{\partial J}{\partial x_1} \\ \frac{\partial J}{\partial x_2} \\ \vdots \\ \frac{\partial J}{\partial x_d} \end{bmatrix} = \begin{bmatrix} w_{11} & w_{12} & \dots & w_{1k} \\ w_{21} & w_{22} & \dots & w_{2k} \\ \vdots & & & \\ w_{d1} & w_{d2} & \dots & w_{dk} \end{bmatrix} \begin{bmatrix} \delta_1 \\ \delta_2 \\ \vdots \\ \delta_k \end{bmatrix} \tag{28}$$

To generalize, we can say,

$$\frac{\partial J}{\partial x} = w \frac{\partial J}{\partial y} \tag{29}$$

which is same as,

$$\frac{\partial J}{\partial x} = W\delta \tag{30}$$

To calculate $\frac{\partial J}{\partial b}$, we need to compute the partial derivative of loss (J) with respect to the vector b.

We know that,

$$\frac{\partial J}{\partial b_1} = \frac{\partial J}{\partial y_1} \tag{31}$$

To generalize, we can say,

$$\frac{\partial J}{\partial b} = \frac{\partial J}{\partial y} \tag{32}$$

which can be represented in the matrix form as,

$$\frac{\partial J}{\partial b} = \begin{bmatrix} \frac{\partial J}{\partial y_1} \\ \frac{\partial J}{\partial y_2} \\ \vdots \\ \frac{\partial J}{\partial y_k} \end{bmatrix}$$

To simplify further, we can write it as,

$$\frac{\partial J}{\partial b} = \begin{bmatrix} \delta_1 \\ \delta_2 \\ \vdots \\ \delta_3 \end{bmatrix}$$

Q1.6

1. The maximum value of the derivative (gradient) of sigmoid is 0.25 which occurs when sigmoid is equal to 0.5. When back-propagation is performed, the gradients across the layers gets multiplied with each other as we progress, and thus as we propagate the gradients through multiple layers, the gradients start to decrease very rapidly. This effect of gradients becoming too small is known as vanishing gradient problem. And then, multiplying this reduced value with the learning rate results in an even smaller gradient for updating the weights of the network. In such a scenario the network takes a very long time to train and the convergence is not guaranteed.

2.

$$\text{Output range of tanh activation} = [-1, 1] \tag{33}$$

$$\text{Output range for sigmoid activation} = [0, 1] \tag{34}$$

The output space for tanh is twice as large as compared to sigmoid function. So, the gradients obtained using the tanh function are greater in magnitude than the gradients obtained using sigmoind activation function. Specifically, with sigmoid activation function, the exponential function converts the negative values of output to very small positive values. In this situation scenarios, using tanh activation function helps in overcoming such small values by mapping negative inputs to negative outputs and not very small positive outputs.

3. As mentioned in previous part, the output range of tanh function is twice as large as compared to the output space of sigmoid function. Thus, the gradients do not decay as rapidly as in sigmoid, it will take slightly longer for gradients to decay to that low a value in case of tanh. However, they do suffer from vanishing gradient problem as well.

4.

$$tanh(x) = \frac{1 - e^{-2x}}{1 + e^{-2x}} \tag{35}$$

$$tanh(x) = \frac{1 - e^{-2x}}{1 + e^{-2x}} + 1 - 1 \tag{36}$$

$$tanh(x) = \frac{1 - e^{-2x} + 1 + e^{-2x}}{1 + e^{-2x}} - 1 \tag{37}$$

$$tanh(x) = \frac{2}{1 + e^{-2x}} - 1 \tag{38}$$

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{39}$$

$$\sigma(2x) = \frac{1}{1 + e^{-2x}} \tag{40}$$

$$tanh(x) = 2\sigma(2x) - 1 \tag{41}$$

2 Network Initialisation

Q2.1.1

In neural networks, we want different neurons to compute different functions to capture different features to be learnt. However, If all the weights are initialized with zeros, all of the neurons will give the same output. In turn, the gradient of each neuron will also be the same. Across multiple epochs, this effect is repeated again and again and finally same weights are assigned to every neuron. This means that all the neurons end up learning the same function, which is opposite to why we want to use neural networks in the first place. The entire network will learn what could be learnt with just a single neuron. So, it is necessary that the weights are initialized randomly to solve this problem and make each neuron learn differently.

Q2.1.3

As mentioned in previous question, if all the weights are initialized to 0, we will face the problem where all the neurons in the neural network end up learning the same function. In such a scenario the network fails the learn effectively and results in poor performance. So, the main reason behind random initialization of weights and bias is to break this symmetry problem. If we scale the initial weights depending on the layer size, we can force the values obtained after applying activation function to lie within a certain range. This enables us to have high value of gradient while training the network and ultimately helps the network to learn faster and better.

3 Training Models

Q3.1

The optimal hyperparameters of the model when max iterations is 50 are:
Batch size = 8
Learning rate = 1e-2
Training Accuracy = 92%
Validation Accuracy = 78.92%
Test Accuracy = 78.77%

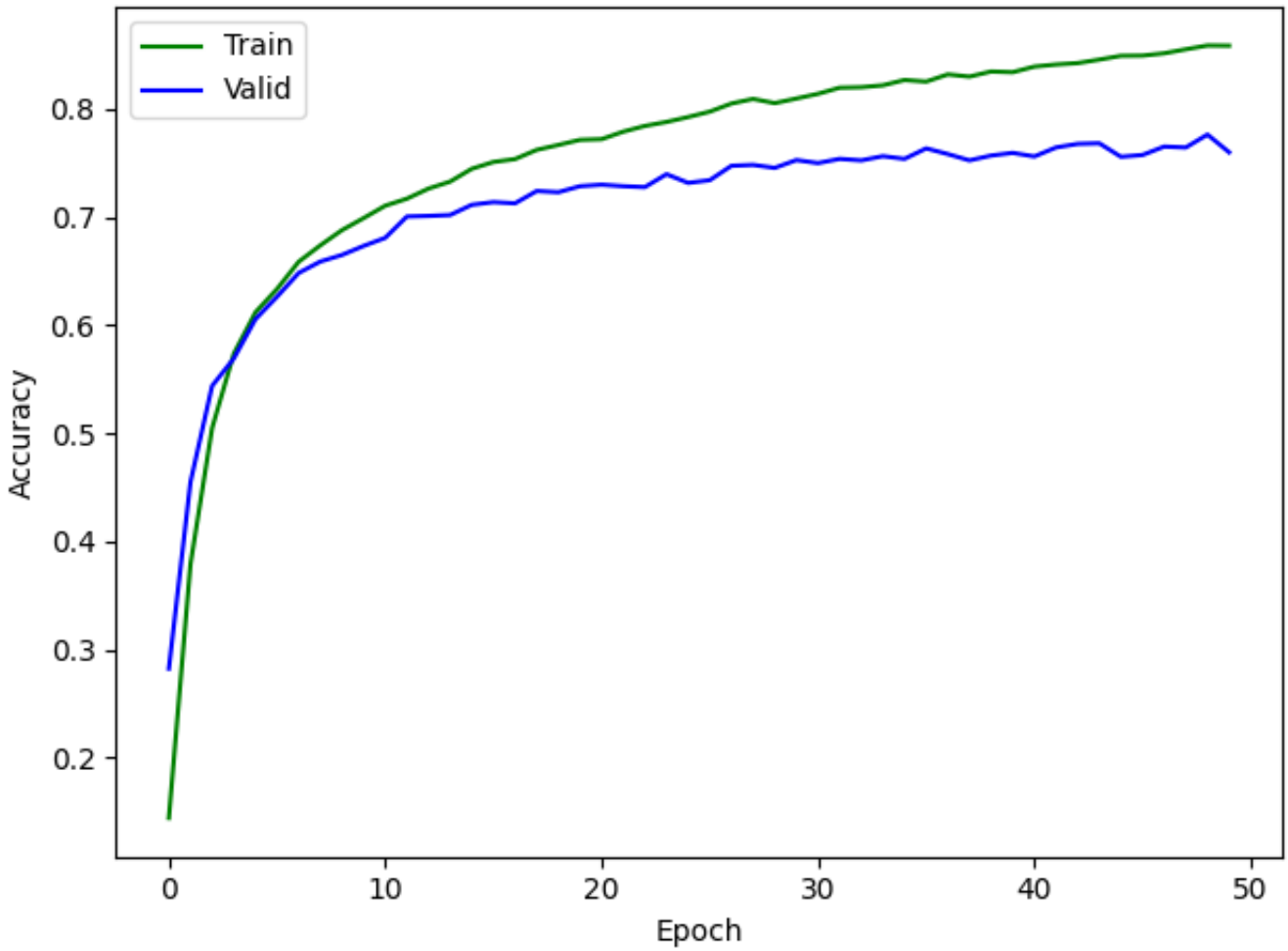


Figure 1: Accuracy vs Epochs for the best model

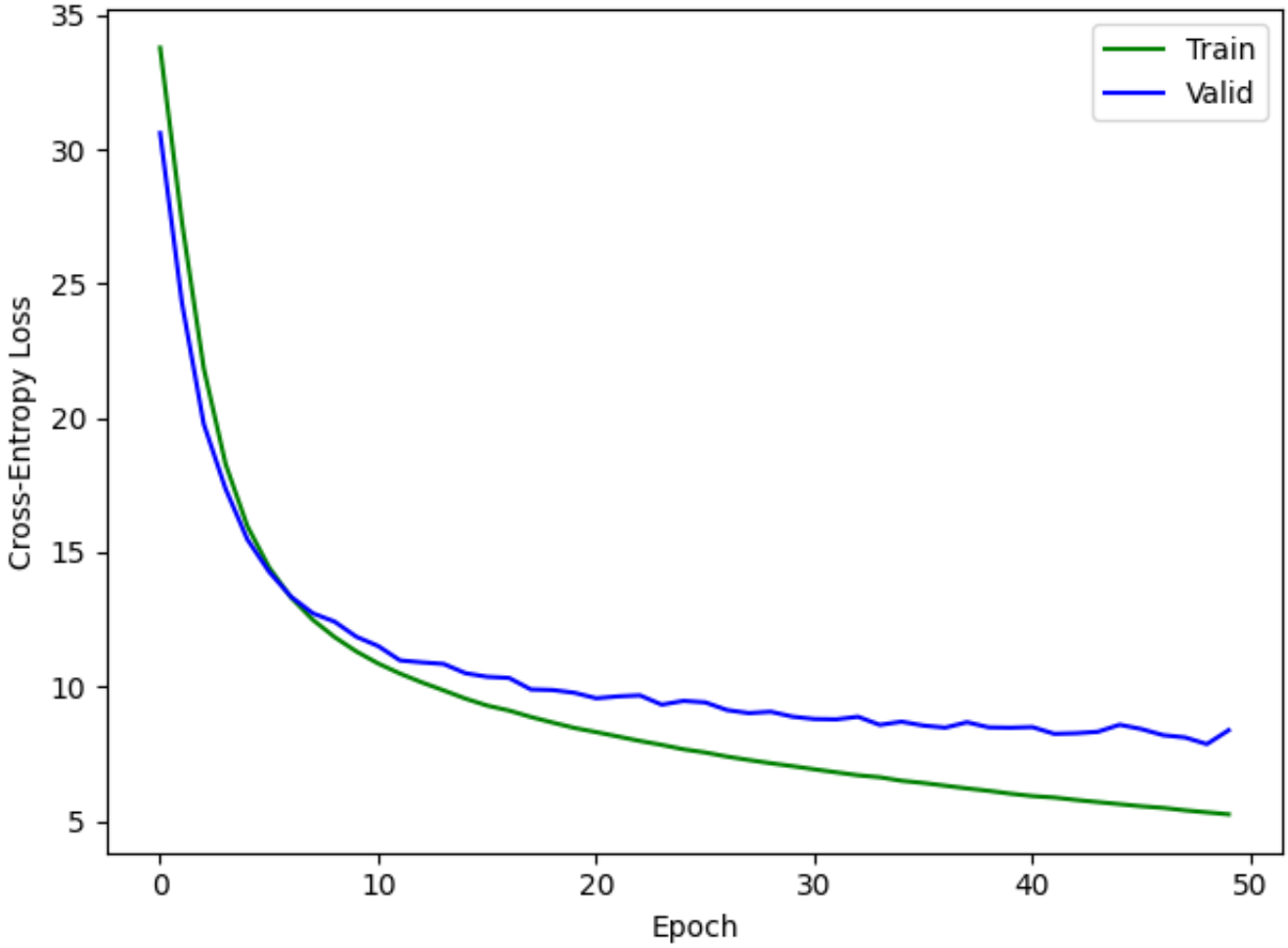


Figure 2: Loss vs Epochs for the best model

Q3.2

With 10 times LR, following metrics were obtained: Train accuracy = 33.33% Validation accuracy = 33.11% Test accuracy = 31.72%

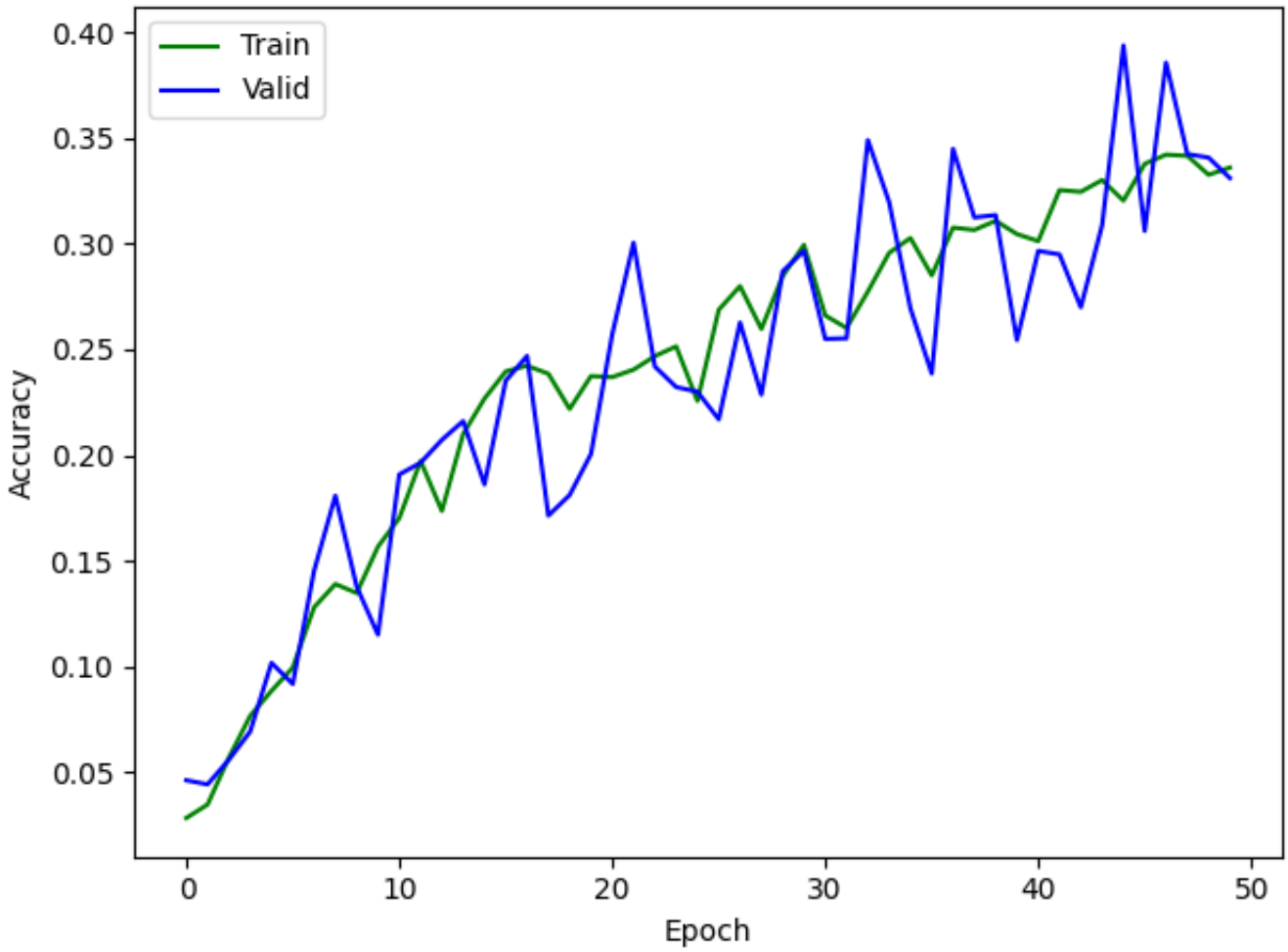


Figure 3: Accuracy vs Epochs for the model with 10 times the learning rate as the best model i.e. 0.1

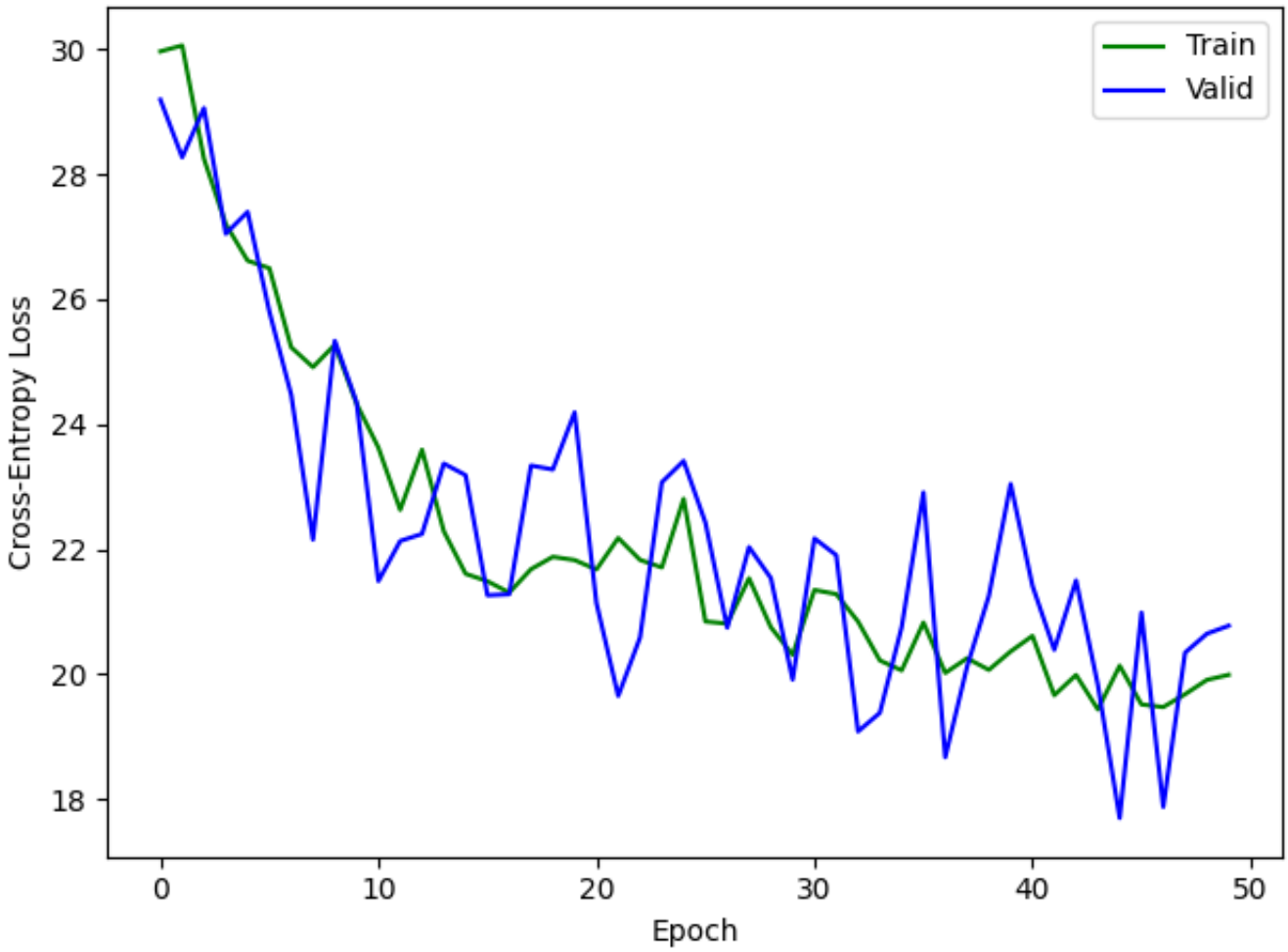


Figure 4: Loss vs Epochs for the model with 10 times the learning rate as the best model i.e. 0.1

With 0.1 times LR, following metrics were obtained: Train accuracy = 79.83% Validation accuracy = 73.66% Test accuracy = 74.88%

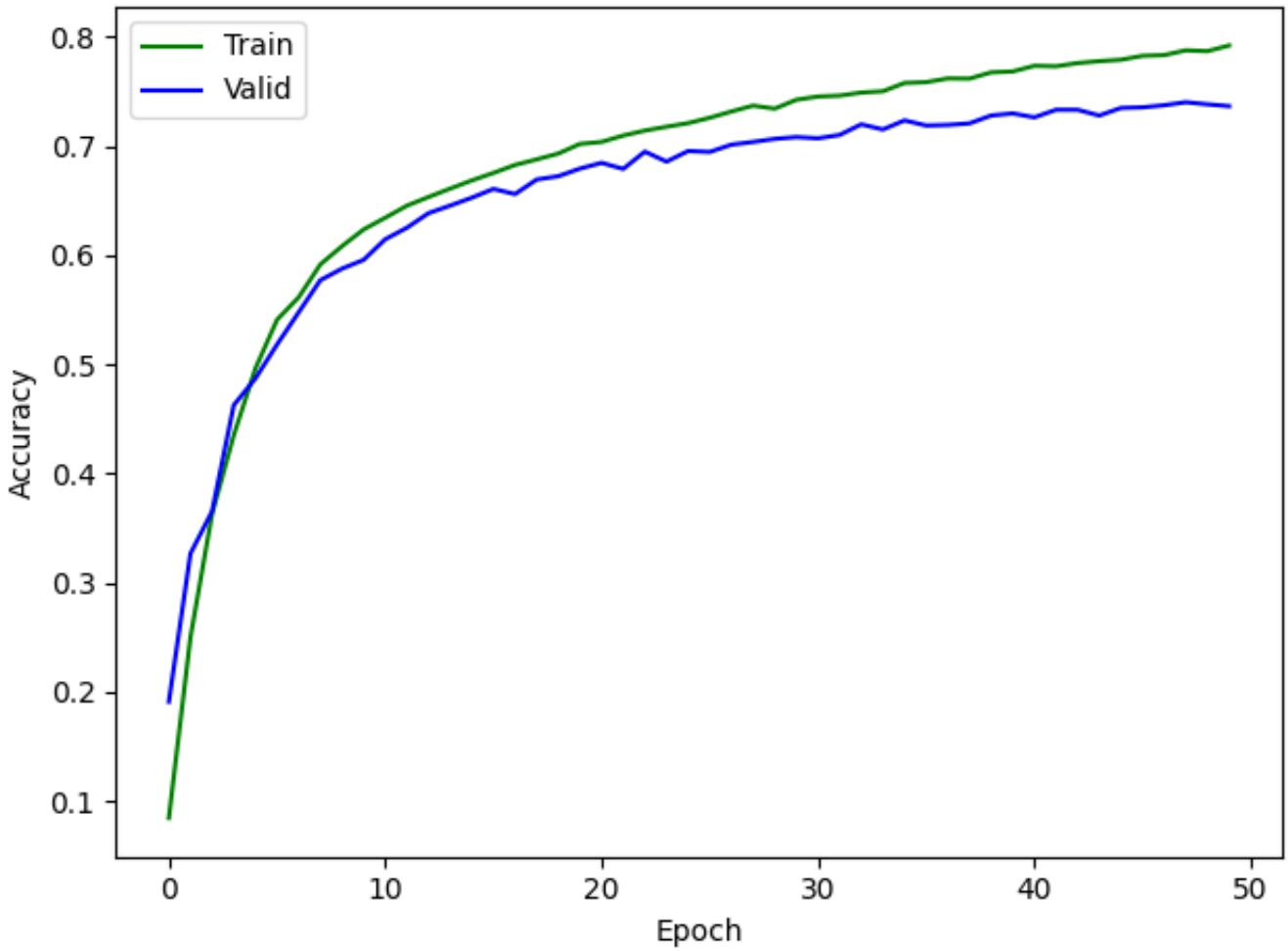


Figure 5: Accuracy vs Epochs for the model with 0.1 times the learning rate as the best model i.e. 0.001

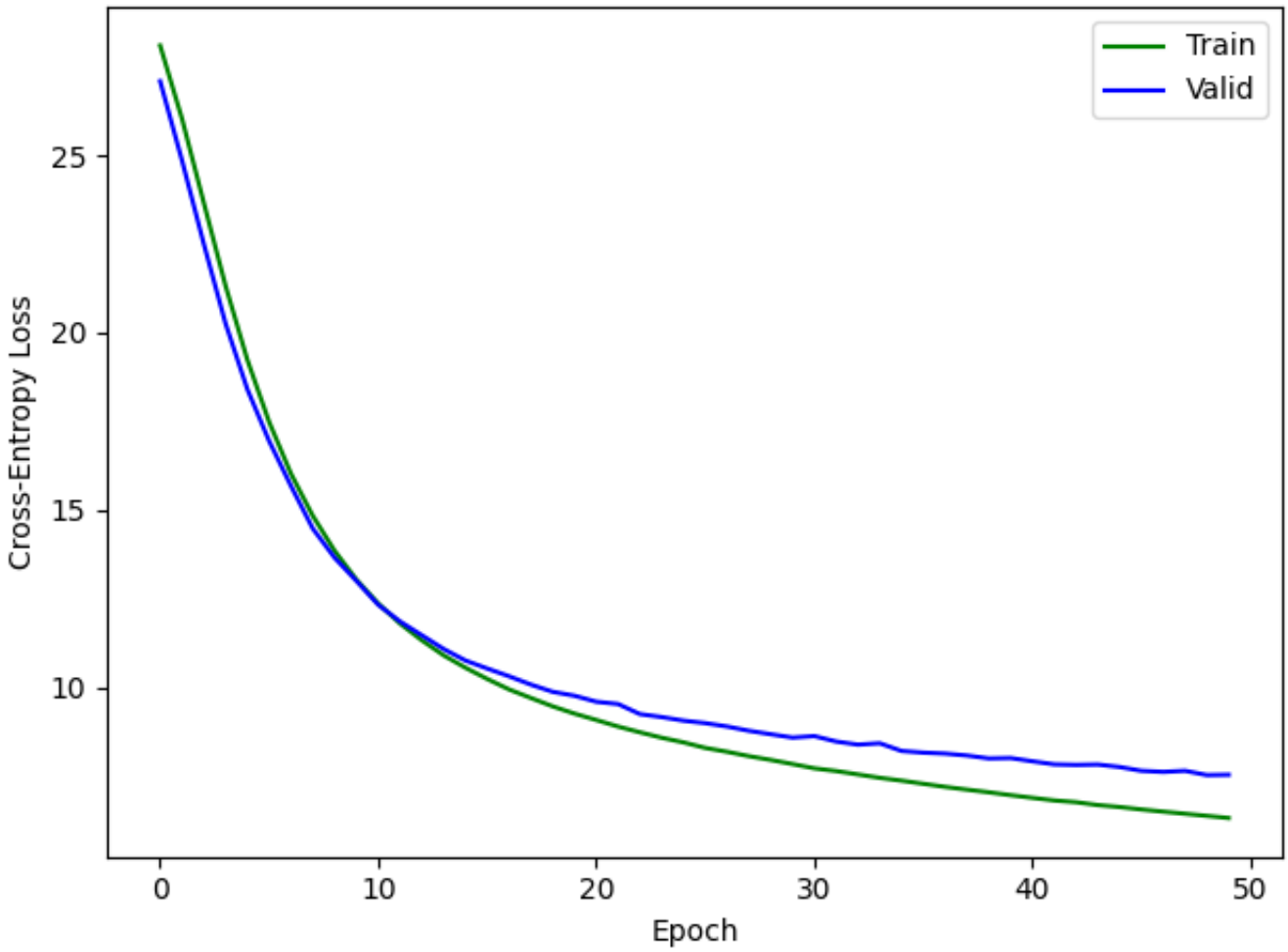


Figure 6: Loss vs Epochs for the model with 0.1 times the learning rate as the best model i.e 0.001

By observing the plot as an effect to how it varies with changing the learning rate, we can say that when learning rate is high, the network oscillates around the local optima and thus we can see many fluctuations in the plot when the learning rate is increased to 0.1 from 0.01. By increasing the learning rate 10 times, the network may not learn optimally as it might get stuck into local suboptimal minima. In that case the loss will not reduce optimally as can be seen in the above graphs.

When learning rate is reduced by 10 times i.e. from 0.01 to 0.001, the rate of training is slower as compared to before. The network takes small steps towards the optimal minima, but it may never reach it in the same number of epochs. It takes very long to train since the network is taking very small steps towards the minima because of the low learning rate.

Performance of the best network on the **test set: 78.7%**

Q3.3

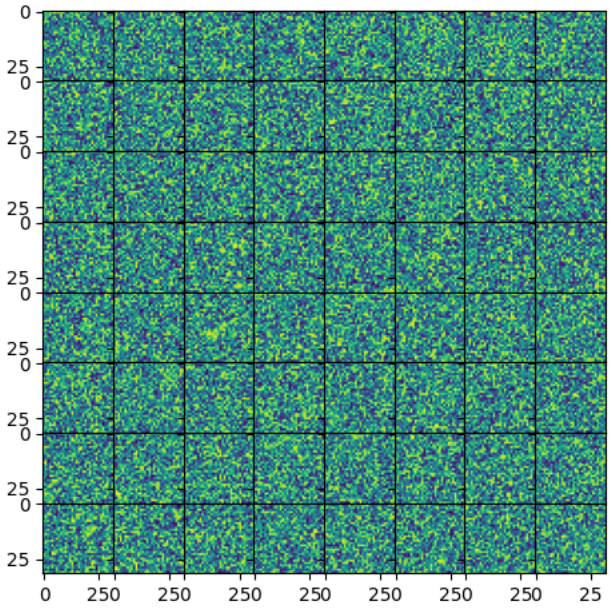


Figure 7: Visualization of weights with random initialization

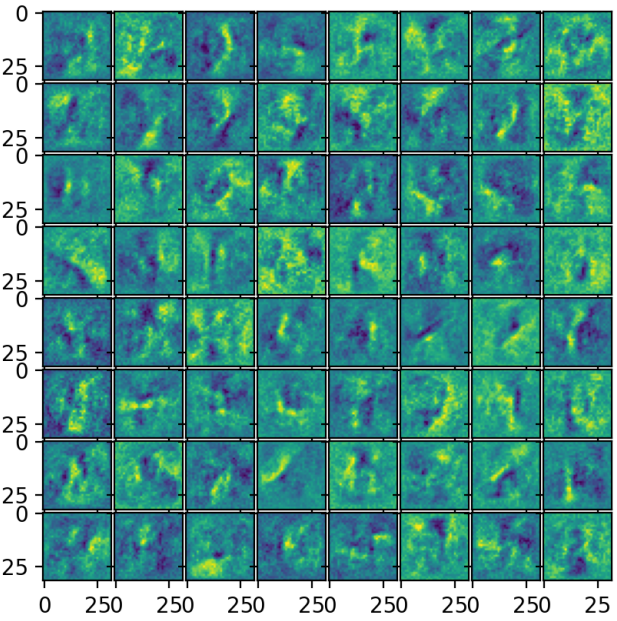


Figure 8: Visualization of weights after training the network for 50 epochs

When the weights are initialized randomly, we cannot see any pattern in the weights visualised above, it is completely random.
But, once the model is trained, the neural network is starting to learning features from the input data. So, we can see some features corresponding to certain patterns and edges that are usually present in the training images. These are small patterns that are detected by the network and combination of these help the network in the classification task overall.

Q3.4

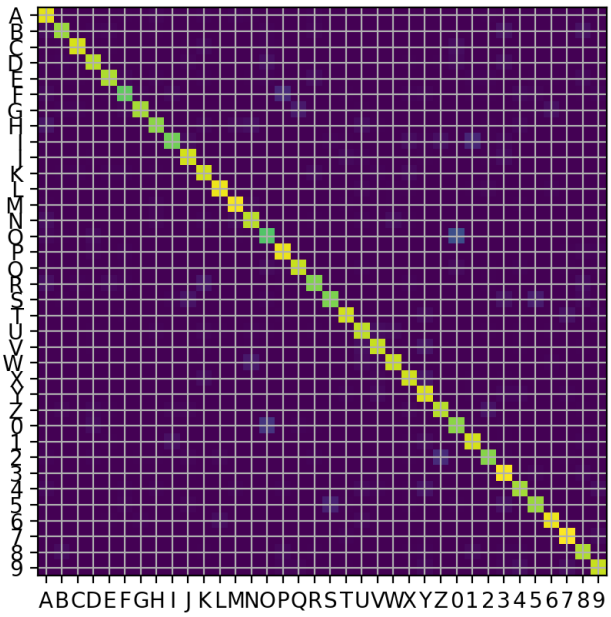


Figure 9: Confusion matrix on test data of the best model

From the figure [9], the top few pairs that are commonly mistaken are as follows:

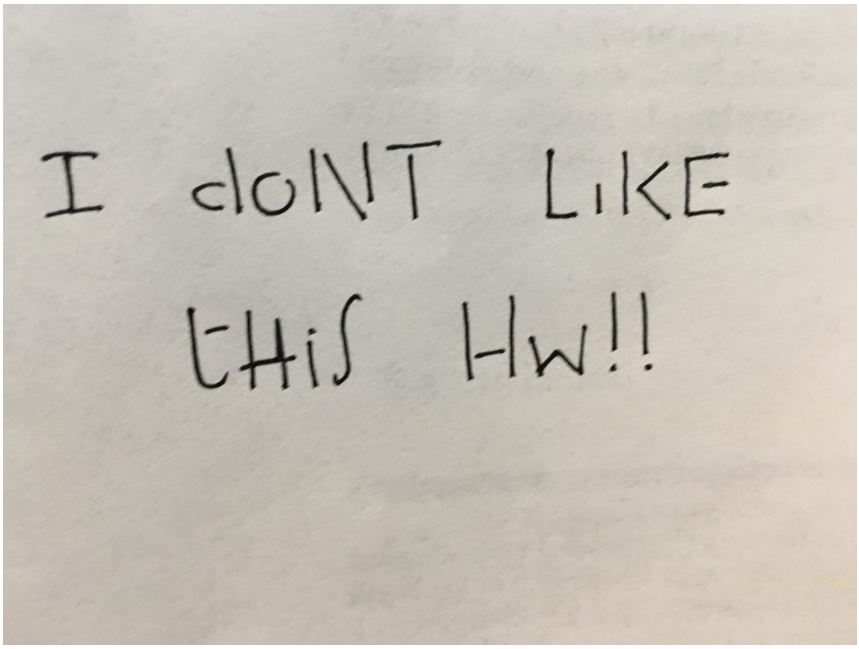
- O and 0: This is an obvious. Both the characters have a similar structure. It might even be difficult for humans to distinguish between these two in most instances.
- 2 and Z: The structure of the characters Z and 2 are similar, only difference being the sharpness of the corner in Z. Therefore, it is common for the network to not easily distinguish between them.
- I and 1: The structure of the characters I and 1 are similar as both the characters have a vertical line and horizontal line, only difference being the additional slanted line in case of 1 and horizontal line at the top in case of I. In regular handwriting these differences might not be so easily distinguishable, thus, it is difficult for the network to classify between these 2 classes.
- S and 5: The structure of the characters S and 5 are similar, only difference being the sharpness of the corner in 5. Therefore, it is common for the network to not easily distinguish between them.

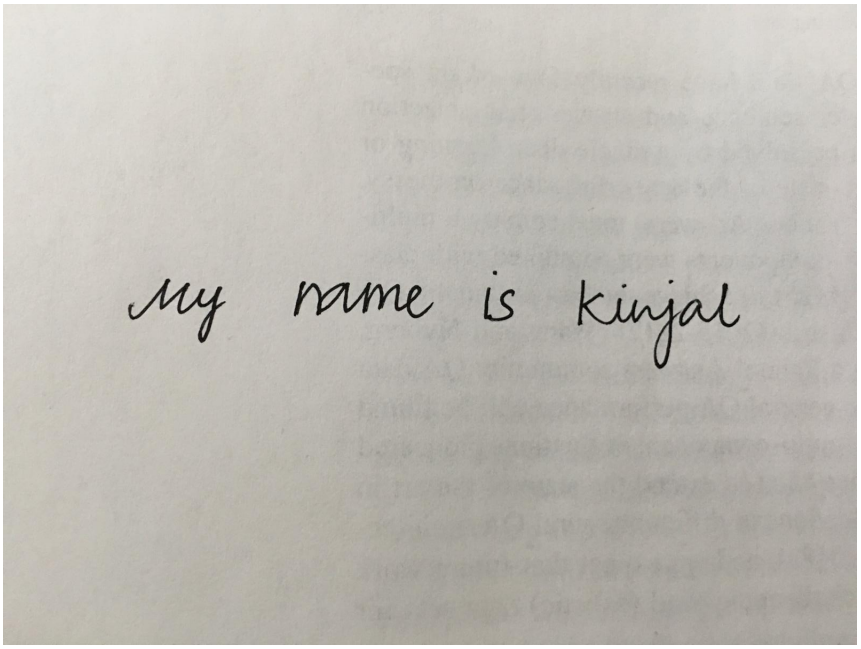
4 Extract Text from Images

Q4.1

According to me the two main assumptions are:

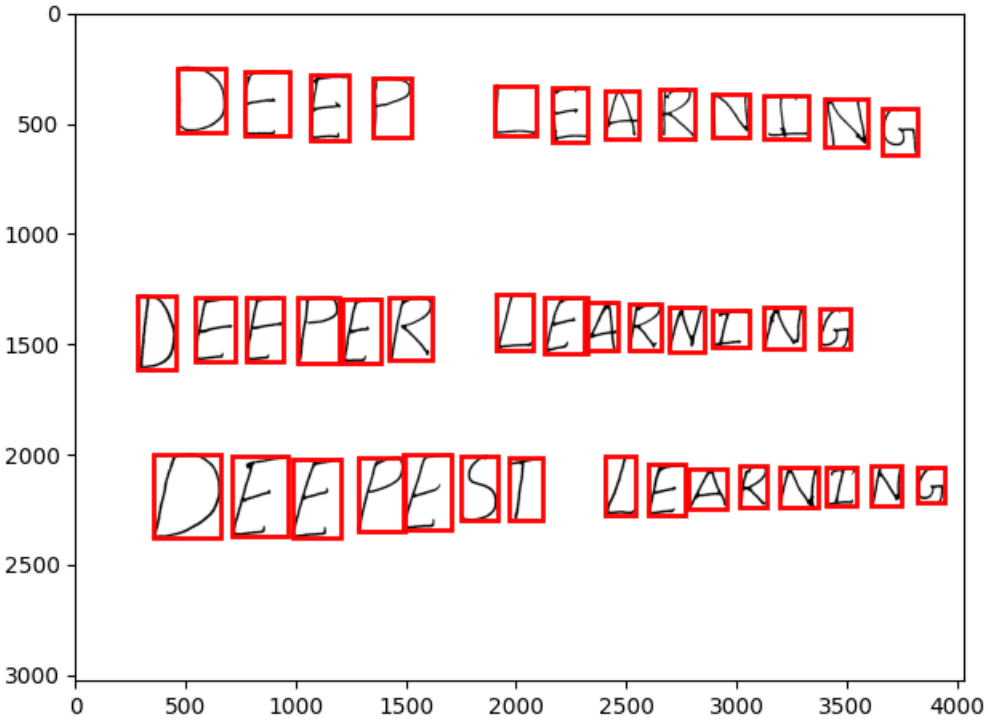
- The model assumes that all characters are completely disconnected. But in cursive handwriting this is not true. Also, sometimes people write the same character lines with separation. Often, the middle dash in E is disconnected. So, the assumption that all the lines/edges within the characters are connected is also made here. It will also not recognise the exclamation as non- character and detect it as L/ I.
- The detector assumes that all characters in the image are almost of the same size. But, this is not the case in most writing formats. People often capitalise to emphasise importance, etc.

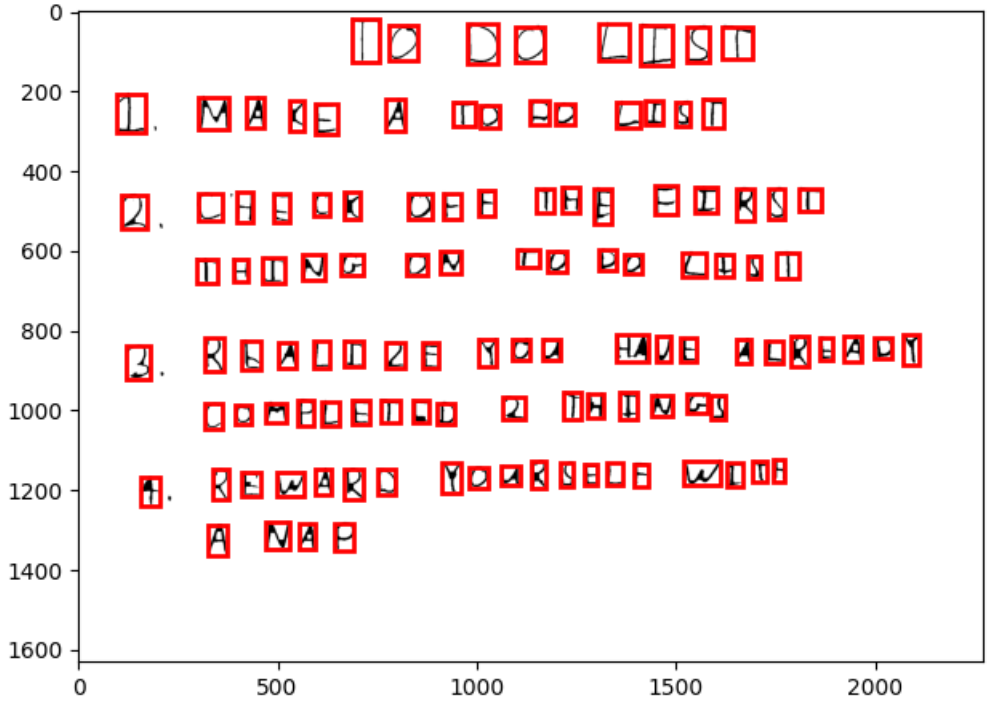


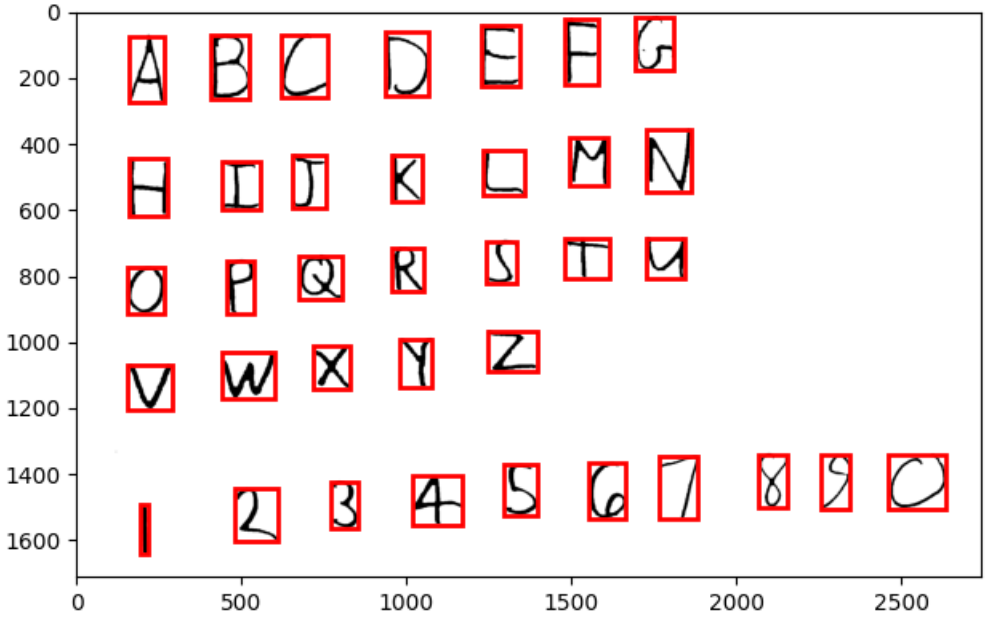


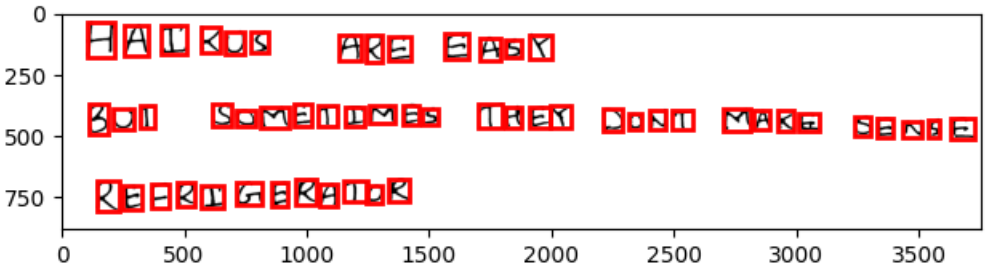
The detection fails in the examples given above. In the first example, it detects 'd' in 'dont' as 'c,l'. It also detects '!' as 'l'. In the second example, the character detection fails as it has characters which are connected in the words, so it cannot segment them properly.

Q4.3






0
200
400
600
800
1000
1200
1400
1600
0
500
1000
1500
2000
2500


0
250
500
750
0
500
1000
1500
2000
2500
3000
3500

Q4.4

- **01_list.jpg**

```
TC DO LIST
I MARE A TVDV 6IST
2 CH45R O4T THE FIR5T
2 T42N4 OH Y444 LXIT
3 RIALZZK YOV NVE AKRXAVT
5OM4LETVD 2 X4X44X
4 REHARD YO4RSELE WIXA
A NAP
```

- **02_letter.jpg**

```
A B D U E F G
H II K L M N
O P Q X S T 4
V W X Y Z
X 2 3 4 S 67 87 0
```

- **03_haiku.jpg**

```
HAIKUS ARE EASY
EUT SOMET2MES T4EY OKKT MAKE OENKE
REFRS6ERATOR
```

- **04_deep.jpg**

```
DEEP LEARMING
DEEPER LEARKING
DEEPEST LEARNING
```

5 Image Compression with Autoencoders

Q5.2

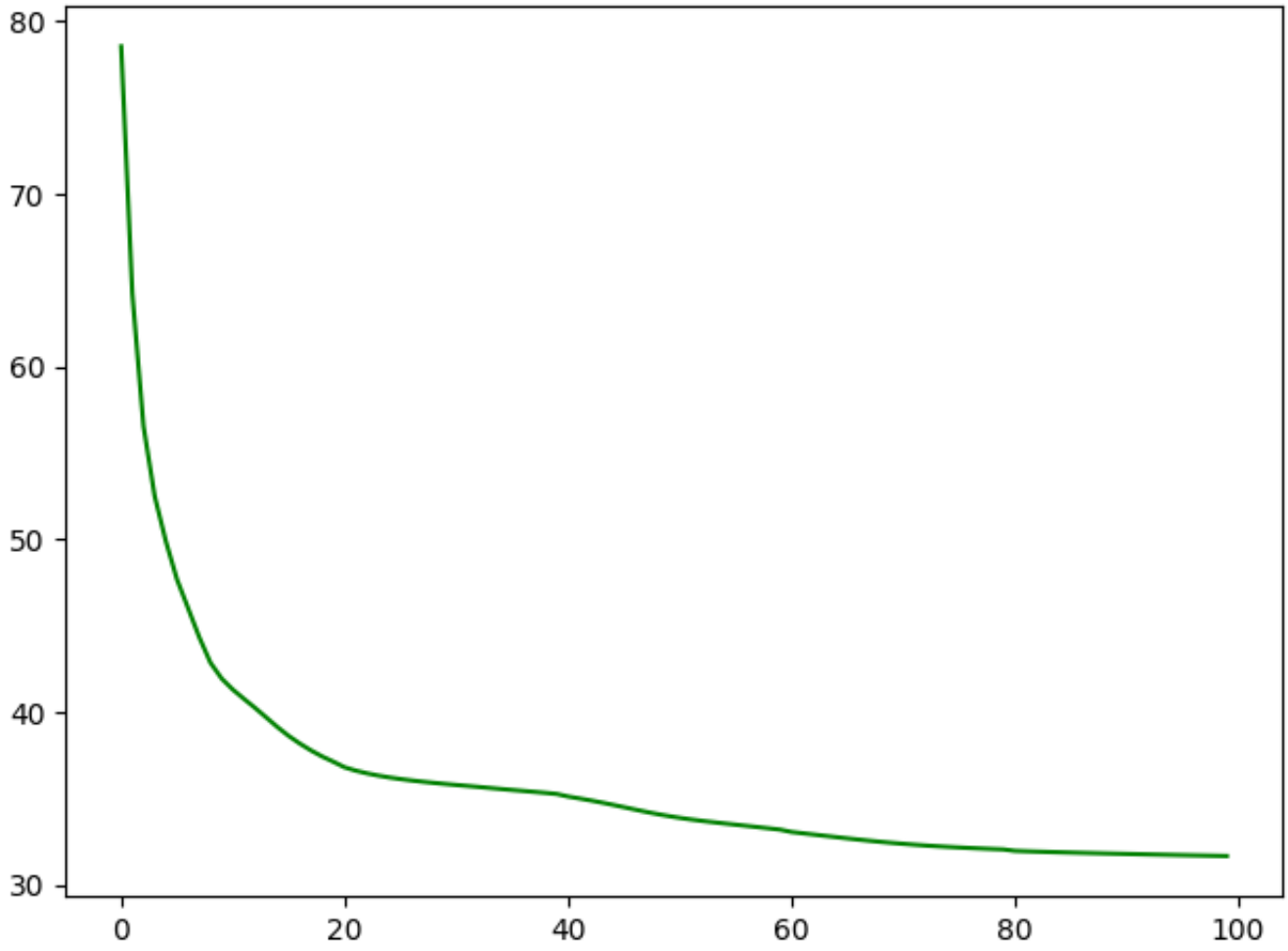


Figure 10: Training loss vs Epochs for autoencoder

Since the weights are initialized randomly, the squared error between the output and input image is large at the beginning. As the training progresses, the error decreases rapidly till about 20 epochs. After that, the loss decreases very slowly and gradually saturates around 85 epochs.

Q5.3.1

It can be observed that the reconstructed image is very similar to the input image. As the model tries to reduce the total squared error in every epoch, it results in outputting the reconstructed image which is very very similar to the input image. So, it can be said that auto-encoder tries to reproduce the input.
Since we are training the autoencoder for only 100 epochs, the reconstructed image does not match precisely with the input. The edges are not as sharp as in the input image. As we can see from the plotted curve of loss vs epochs, the error reduces in each epoch, but it does not produce zero loss. Thus the reconstructed image does not match the input image exactly.

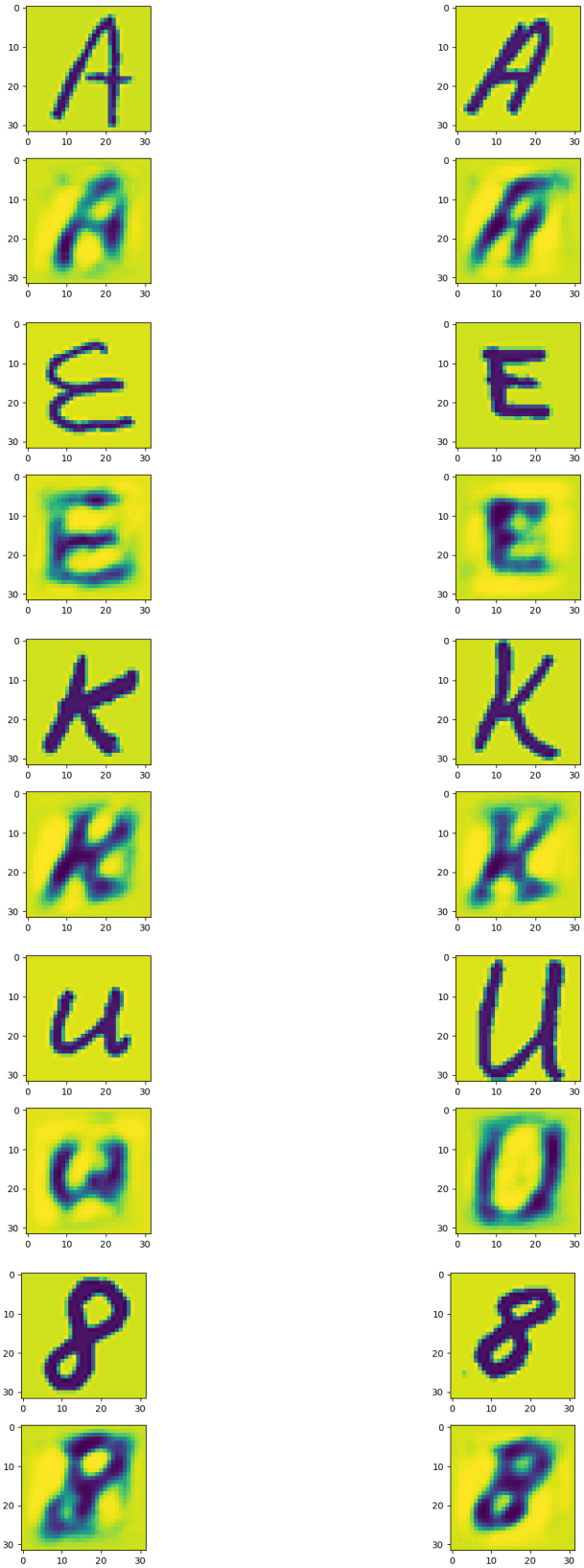


Figure 11: Output of Autoencoder for classes A, E, K, U, 8

Q5.3.2

Average PSNR obtained from the autoencoder across all images in validation set: 15.386008237936808.

6 PyTorch Extra Credit

Q6.1.1

The python code for this task is present in 'run_q6_1_1.py'.

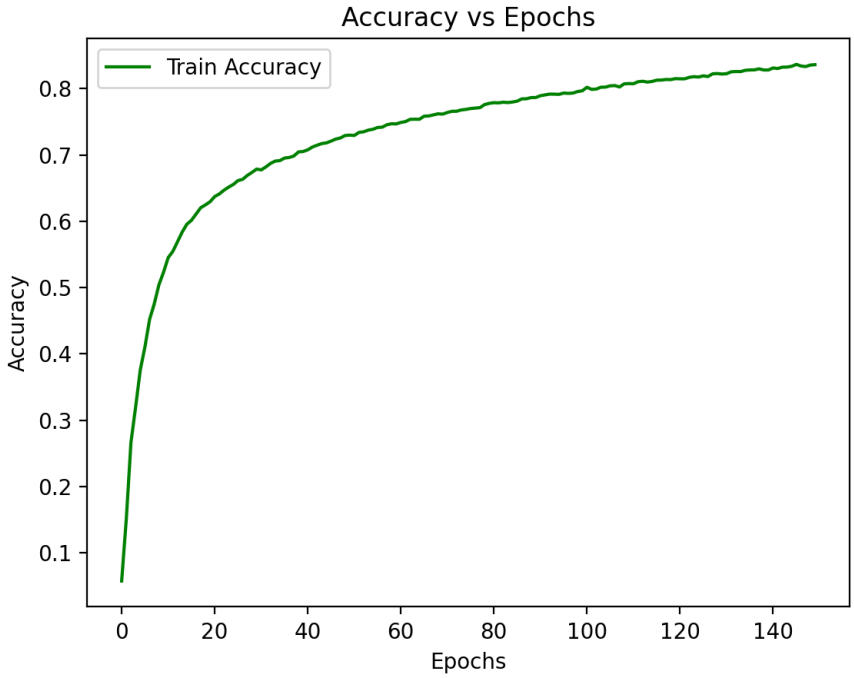


Figure 12: Training Accuracy vs Epochs for NIST36 - Fully Connected Network

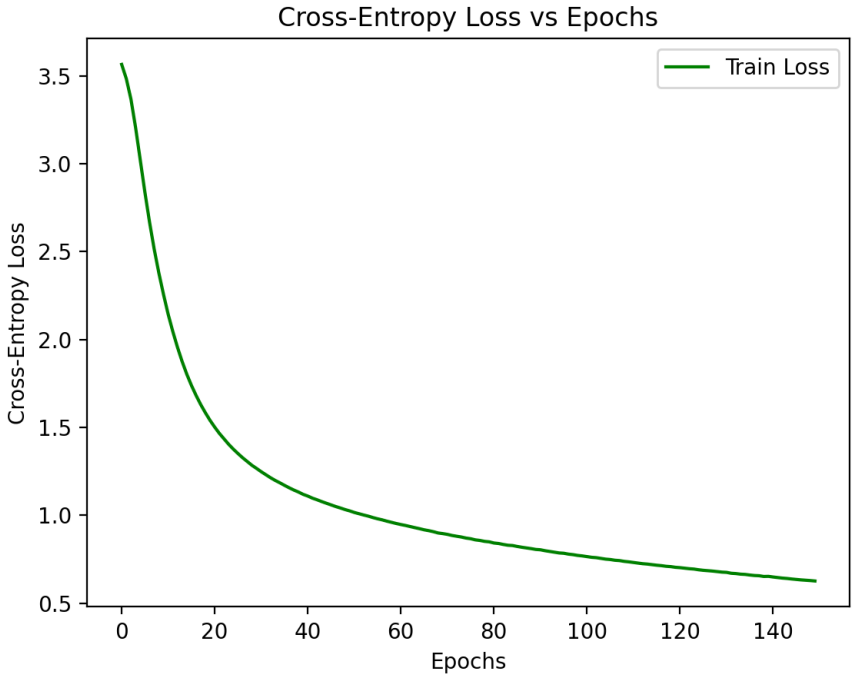


Figure 13: Training loss vs Epochs for NIST36 - Fully Connected Network

Q6.1.2

The python code for this task is present in 'run_q6_1_2.py'. Compared to previous fully-connected network CNN with 3 layers performs much better, and it learns much much faster i.e. only in 25 epochs compared to 150 epochs in Fully connected network. The training loss goes almost to 0.1 with CNN unlike 0.6 with Fully connected network.The training accuracy is also much higher with CNN which is 96.45% compared to 84.29% with just fully connected network from previous section.

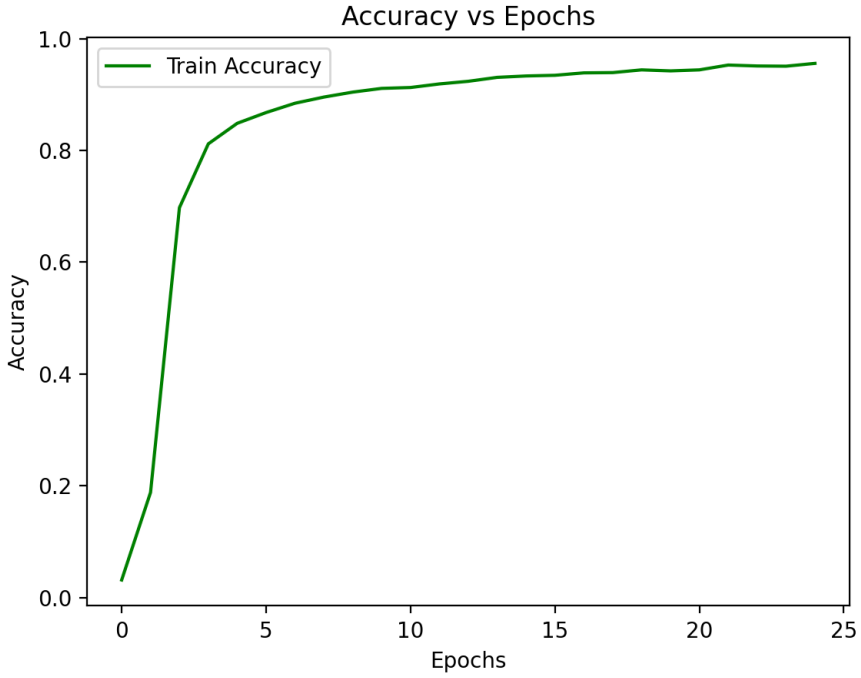


Figure 14: Training Accuracy vs Epochs for NIST36 - CNN

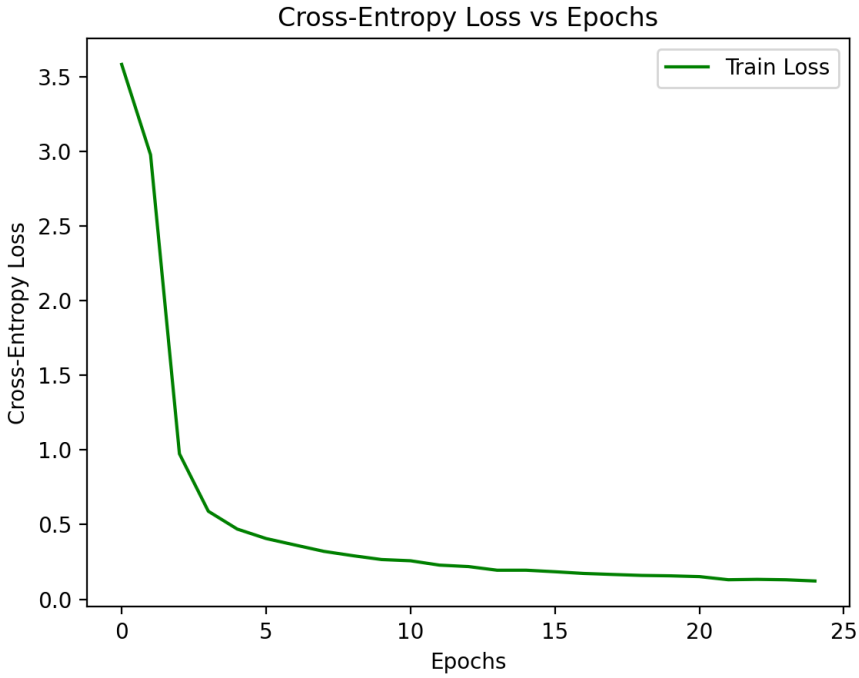


Figure 15: Training loss vs Epochs for NIST36 - CNN

Q6.1.3

The python code for this task is present in 'run_q6_1_3.py'.

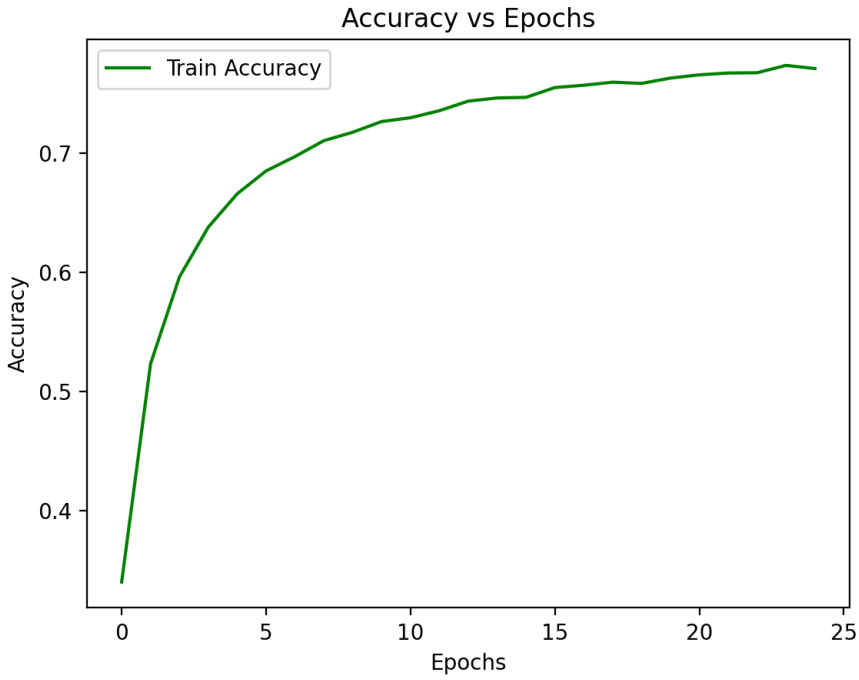


Figure 16: Training Accuracy vs Epochs for CIFAR-10 using CNN

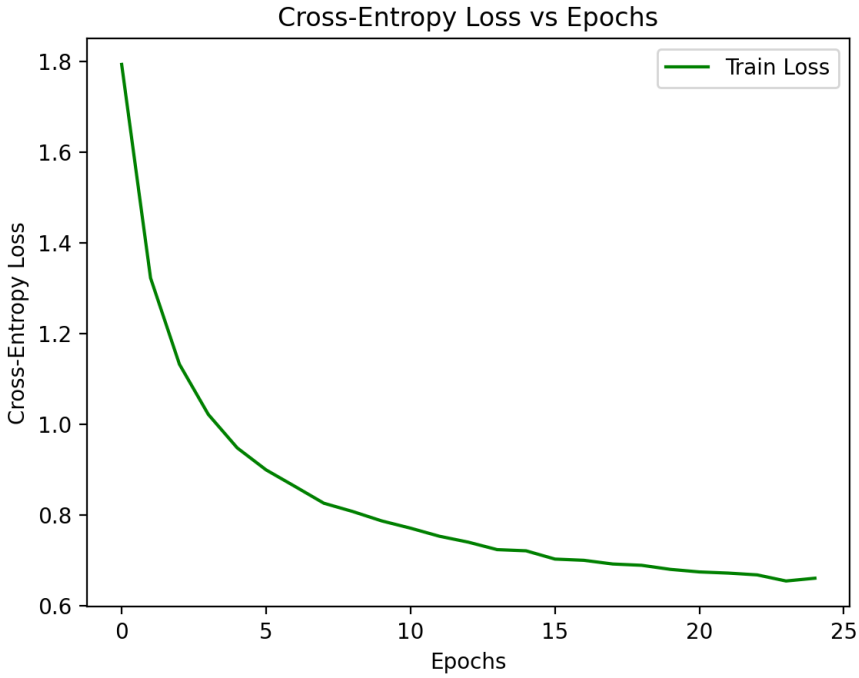


Figure 17: Training loss vs Epochs for CIFAR-10 using CNN

Q6.1.4

The python code for this task is present in 'q6_1_4.py'.

The best results obtained on HW1 on this dataset on test data was 65.25%. However, with CNN, the accuracy improved to 71% on test data and 74.4% on train data. This is mostly because CNN tries to detect features using 2D convolutions unlike just taking some features manually in HW1.

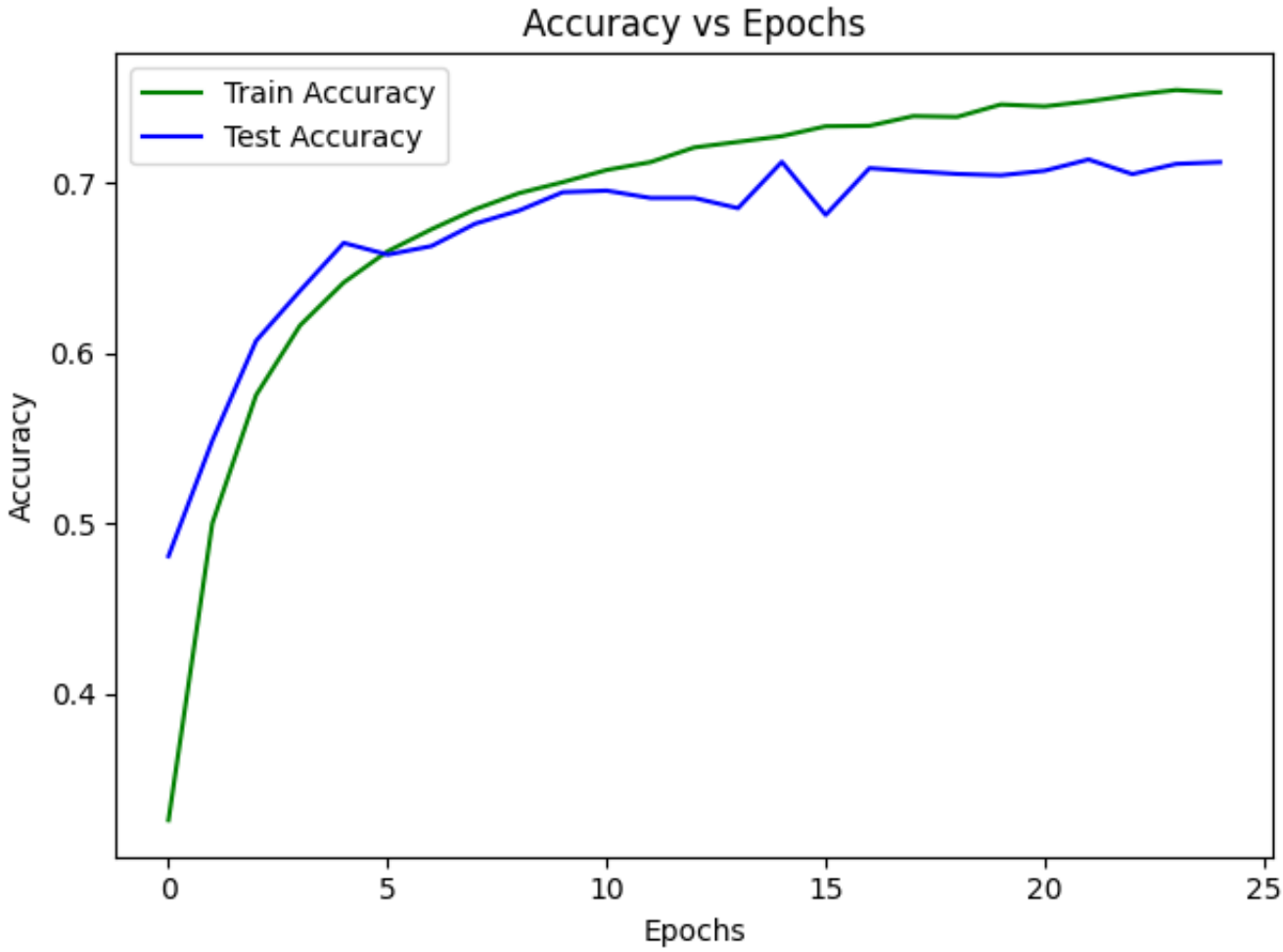


Figure 18: Training Accuracy vs Epochs for CNN on SUN dataset

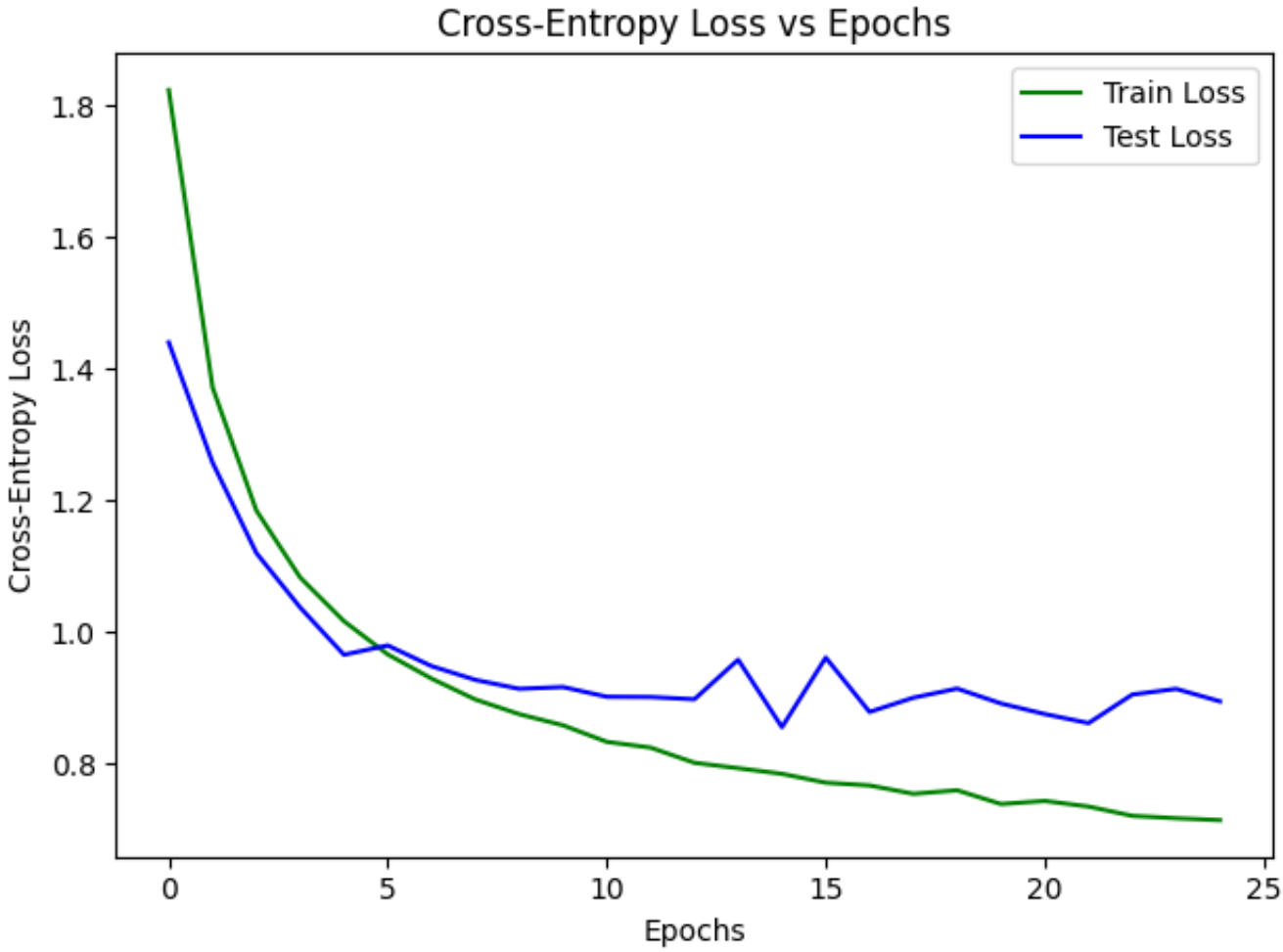


Figure 19: Training loss vs Epochs for CNN on SUN dataset

Q6.2

The python code for this task is present in 'run_q6_2.py'.

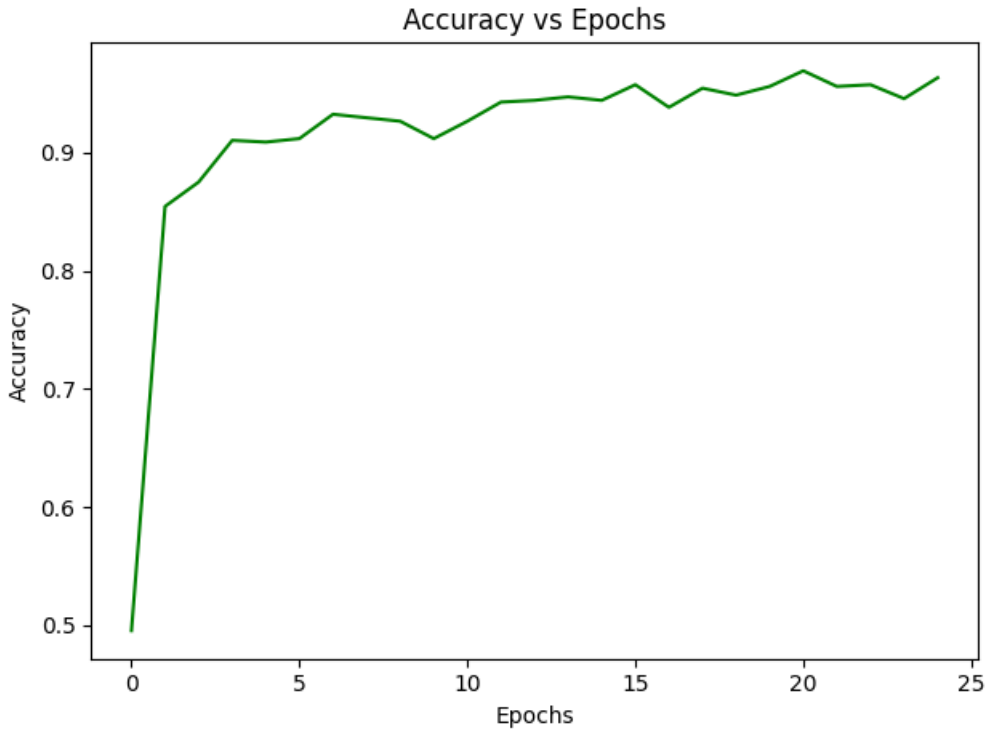


Figure 20: Training Accuracy vs Epochs on SqueezeNet fine-tuning on flowers 17 dataset

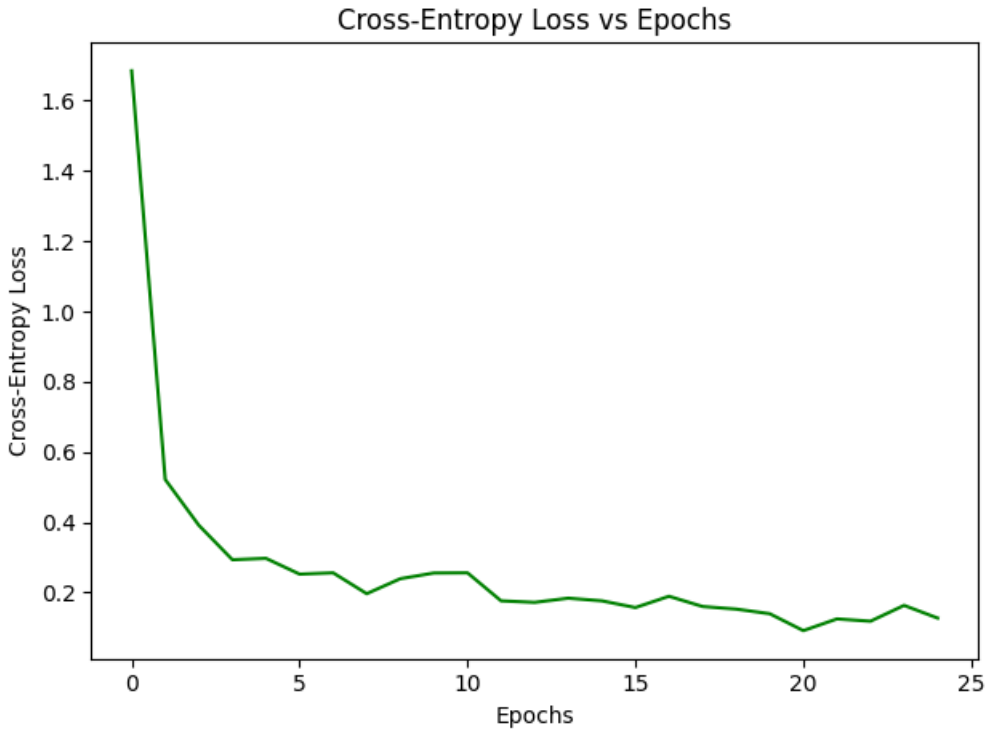


Figure 21: Cross Entropy Loss vs Epochs on SqueezeNet fine-tuning on flowers 17 dataset

This model is training by fine tuning the pretrained SqueezeNet model. The classifier layer is modified to have output size of 17 classes since 17 types of flowers are present in flowers 17 dataset. Only the parameters of this classifier layer are updated with fine-tunning for which results are shown in the plots above. We can see that within first 5 epochs, the accuracy reaches 90%. And then the accuracy and loss stabilizes around 15 epochs which denotes that the fine-tuned model converges.