Submitted by: Shahram Najam Syed (snsyed) **Homework # 5 (16-720)**
Dated: 1st December, 2023

Q1.1:

Ans: To prove that the softmax function is translation invariant, we can simply start by defining the softmax function and then show that for any constant translational shift "c", $softmax(x) = softmax(x + c)$.

The softmax function for a vector x is defined as follows:

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

For each index i in the vector x.

Now, lets consider the vector translated by a constant "c", i.e. vector $x + c$. The softmax of this translated vector is:

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

Which simplifies to:

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

Since e^c is a constant, it can be factored out in the denominator (from the summation).

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

Now, e^c cancels out in both the numerator and the denominator:

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

Which is the original softmax function:

$$softmax(x_i + c) = softmax\ (x_i)$$

The choice of $c = -\max_i(x_i) \leq x_i$ in practice is motivated by numerical stability. The largest value x after translation becomes 0, i.e. $x_i - \max_i(x_i) \leq 0$ for all i. This prevents large exponentials when computing e^{x_i}, which could lead to numerical issues (e.g. overflow).

In contrast, if $c = 0$, then e^{x_i} for large values of x_i can result in very large numbers, potentially causing computational instability. By translating x such that the maximum value is 0, we ensure the arguments to the exponential function are non-positive, leading to more manageable and stable computations in the softmax function.

Q1.2:

Ans: As presented in the write up, the softmax function can indeed by broken down into a three-step process, which are as follows:

a. Exponentiation: For a vector $x \in R^d$, compute $s_i = e^{x_i}$ for each element x_i. This step converts each element of x into a positive number (since the exponential function e^x is always positive).
b. Summation: Compute the sum of the exponentiated values using: $S = \sum_{i=1}^{d} s_i$.
c. Normalization: Finally, compute the softmax of each element as $softmax\ (x_i) = \frac{1}{s} s_i$. This step normalizes each s_i by the sum S, ensuring that the output values are between 0 and 1.

- **As x ∈ R d , what are the properties of softmax(x), namely what is the range of each element? What is the sum over all elements?**

Properties of softmax:

For a vector $x \in R^d$ passed through the softmax function, the properties of the resulting vector $softmax\ (x_i)$ are as follows:

- Each element of $softmax\ (x)$ is in the range (0,1). This is because each element $softmax\ (x_i)$ is calculated as a ratio of positive number (since $e^{x_i} > 0$) for any real number x_i to the sum of positive numbers. As a ratio of a part to a whole, each element is thus a non-negative fraction less than or equal to 1.
- The sum of all elements of $softmax\ (x)$ is exactly 1. This property arises because the softmax function normalizes the vector x such that the elements represent a probability distribution. Mathematically,

$$\sum_{i=1}^{d} \boldsymbol{softmax}(\boldsymbol{x_i}) = \mathbf{1}$$

Which aligns with the definition of a probability distribution where the total probability must sum to 1.

- **One could say that "softmax takes an arbitrary real valued vector x and turns it into a _____".**

One could say that, "softmax takes an arbitrary real valued vector x and turns it into a probability distribution". This statement succinctly captures the essence of the softmax function, i.e. it converts any real-value vector, regardless of its elements values or the vector's dimension, into a vector of the same dimension where each element lies between 0 and 1, and the sum of all these elements equals 1. The transformed vector can then be interpreted probabilistically, with each element representing the probability of a corresponding outcome or class.

- **Can you see the role of each step in the multi-step process now? Explain them.**

i. Exponentiation:
   a. Purpose: This step involves taking the exponential of each element in the vector x. The exponential function e^{x_i} ensure that every output is positive, regardless of whether the input x_i is positive, negative, or zero.
   b. Roles: By converting to positive values, it prepares the data for creating a probability distribution. Additionally, exponentiation amplified those differences between the elements of x. Larger values in x becomes significantly larger in the output, while smaller values (especially negative) become closer to zero (which is important from the context of classification to accentuate the most likely class).

ii. Summation:
   a. Purpose: This step involves summing all the exponentiated values to get a total sum S.
   b. Role: The summation provides a normalization factor. It aggregates the weights of all exponentiated inputs, creating a base against which each exponentiated value can be proportionally compared. This total sum acts as a denominator in the next step, ensuring the final values are appropriately scaled.

iii. Normalization:
   a. Purpose: In this step, each exponentiated value s_i is divided by the total sum S to produce the final softmax output.
   b. Role: Normalization is critical for transforming the vector into a probability distribution. By dividing each s_i by S, each element in the softmax output is guaranteed to be between 0 and 1, and the sum of all these elements equals 1. This step effectively scales down the exponentiated values proportionally, ensuring that they collectively form a valid probability distribution.

Q1.3.:

Ans: In this question we are asked to prove that a multi-layer neural network without non-linear activation function is equivalent to a linear regression. In such a case, each layer computes a linear transformation of its input. For layer i, this transformation is given by $x_{i+1} = W_i x_i + b_i$, where W_i and b_i are the weight matrix and bias vector for the i^{th} layer, respectively, and x_i is the input to the i_{th} layer.

Since in a multi-layer network, the output of one layer becomes the input to the next so, for a network with n layers, the output y can be expressed as a series of nested linear transformations:

$$y = W_n(W_{n-1}\,(\dots(W_2(W_1 x_1 + b_1) + b_2)\dots) + b_{n-1}) + b_n$$

Since matrix multiplication and addition are associative, we can simplify the above expression by collapsing the nested terms, and this combining all the weights and biases into a single weight matrix W' and a single bias vector b'. Thus we can rewrite y as:

$$y = W'x + b'$$

Which is a linear equation indicating a linear regression model computing the output as a linear combination of the input features, with a set of weights and a bias term.

Q1.4:

Ans: The sigmoid activation function is given by:

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

Derivating both sides w.r.t. x:

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

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

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

Which is the gradient/derivative of the sigmoid function w.r.t. x.

Since,

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

i.e.

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

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

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

So,

$$\frac{d}{dx}\sigma(x) = \frac{\frac{1-\sigma(x)}{\sigma(x)}}{\left(\frac{1}{\sigma(x)}\right)^2}$$

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

This derivation shows that the gradient of the sigmoid function can be written as a function of $\sigma(x)$ itself, without directly accessing x.

Q1.5:

Ans: Since we are given:

$$y = Wx + b$$

Where J is some loss function.

Since,

$$\frac{\partial J}{\partial W} = \frac{\partial J}{\partial y}\frac{\partial y}{\partial W}$$

Since we are given that,

$$\frac{\partial J}{\partial y} = \delta \in R^{k\times 1}$$

And,

$$\frac{\partial y}{\partial W} = x^T$$

Hence,

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

Now, for the derivative w.r.t. x:

$$\frac{\partial J}{\partial x} = \frac{\partial J}{\partial y}\frac{\partial y}{\partial x}$$

Since we are given that,

$$\frac{\partial J}{\partial y} = \delta \in R^{k\times 1}$$

And,

$$\frac{\partial y}{\partial x} = W^T$$

Therefore,

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

Now, finally for the derivative w.r.t. b:

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

Since we are given that,

$$\frac{\partial J}{\partial y} = \delta \in R^{k \times 1}$$

And,

$$\frac{\partial y}{\partial b} = 1$$

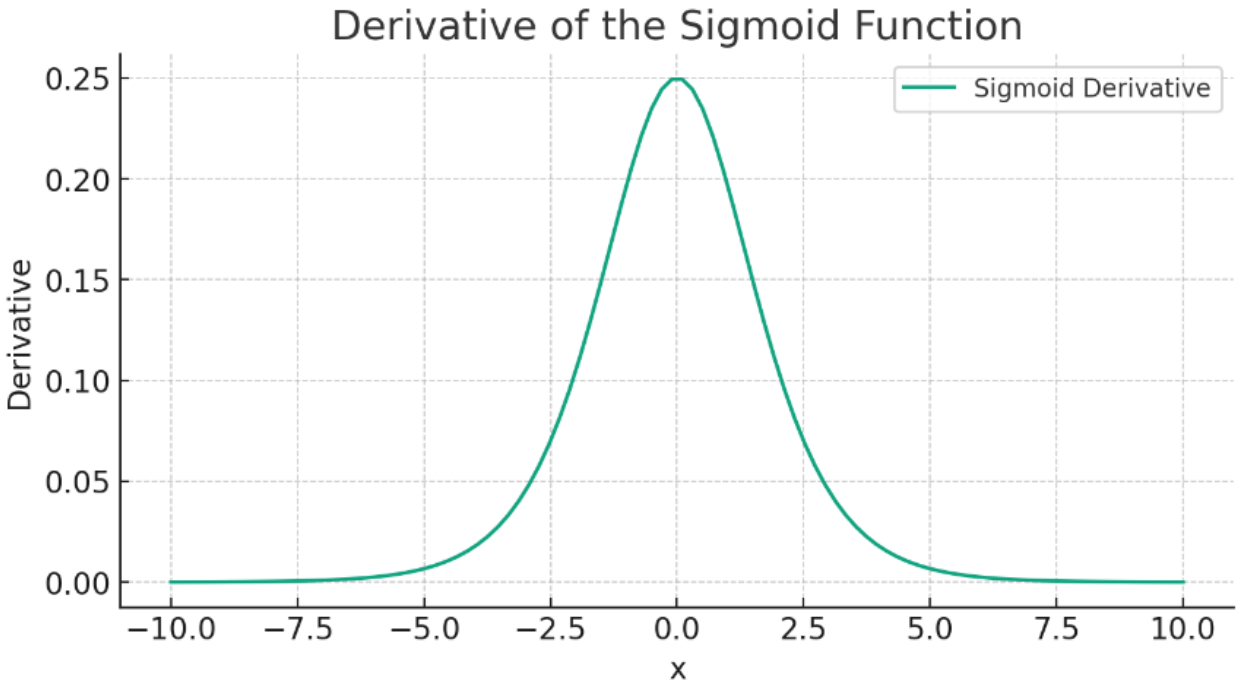
Hence,

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

Q1.6:

Ans:

1. The sigmoid activation function $\sigma(x) = \frac{1}{1+e^{-x}}$ has a derivative ranging between 0 and 0.25. In deep neural networks, this small derivative gets multiplied in succession layer after layer during backpropagation. Since these products are less than 1, they progressively get smaller, leading to very small gradients in earlier layers. This phenomenon is known as the vanishing gradient problem, and it makes the network hard to train effectively particularly for deeper architecture.



2. The tanh function (tangential hyperbolic), defined as:

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

Has an output range of (-1,1). In contrast sigmoid's output range is (0,1), which means it never outputs negative values. The zero-centered nature of tanh can lead to faster convergence during training because when inputs to a neuron are zero-centered, the gradients are less likely to get stuck near regions where gradients are near zero.

3. The derivative of $\tanh(x)$ function is $1 - \tanh^2(x)$, which has a range of (0,1), which implies that the gradients are generally larger tan those of the sigmoid function. Hence, $\tanh(x)$ partly mitigates the vanishing gradient problem, especially comparison to sigmoid, as it provides stronger gradients for a wider range of input values.
4. Since we are given:

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

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

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

Since,

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

$$\tanh(x) = \frac{1 - \dfrac{1 - \sigma(2x)}{\sigma(2x)}}{1 + \dfrac{1 - \sigma(2x)}{\sigma(2x)}}$$

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

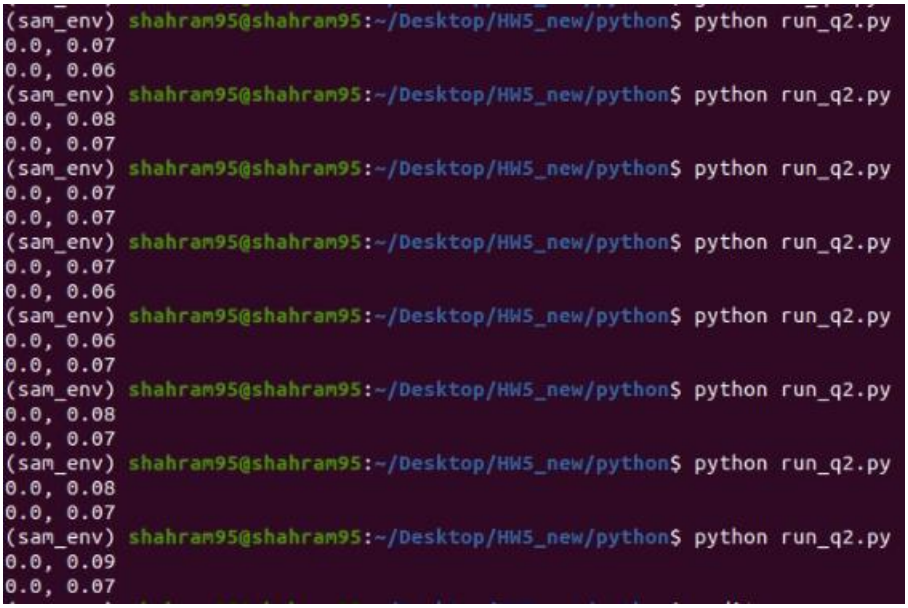
Q2.1.1.:

Ans: Initializing a neural network with all zeros is not a good idea for several reasons:

a. In a neural network, each neuron should learn to recognize different features or patterns in the input data, but if all weight are initialized to zero, every neuron in a layer will produce the same output during the forward pass. This means that during backpropagation each neuron in the layer will receive the same gradient update which would result in all the neurons learning the same features during training resulting in a highly inefficient model and thus being antithetical to the concept of having multiple neurons in a layer.
b. The learning of a neural network occurs as it adjusts its weights in response to the error from the cost function observed at the output which is usually done using gradient descent. But when weights are initialized to zero, the gradient calculated for the weight update during backpropagation will also be zero (since the derivative of the error with respect to the weights will be zero) which would result in the network not learning anything.
c. With zero-initialized weights and biases the output of every neuron will be zero-initialized. Even after several iterations of training the network will likely continue to output uniform values, which is a problematic tasks like classification where distinct outputs are required to differentiate between classes.

2.1.2.

Ans: Coded in python, submitted in the zipped file submission. As shown below, the output from then run_q2.py script is consistently bounded between 0, [0.05 to 0.12]:



2.1.3.:

Ans: Following are the reason why we initialize weights with random number, and as to why we scale the initialization depending on layer size:

- Random initialization ensures different neurons learn diverse features preventing uniformity in learning across neurons.
- Properly scaled random weights prevent gradients from becoming too small (vanishing) or too large (exploding), crucial for stable training in deep networks.
- Scaling initialization based on layer size (as done in Xavier initialization) keeps activation variances stable across layers, ensuring consistent learning dynamics.
- Random and scaled initialization promotes faster convergence and balanced learning across network layers, particularly important in deep architectures.

Q2.2.1:

Ans: Coded in python, submitted in the zipped file submission. As shown below, after running the script run_q2.py following is the result we get:

```
(sam_env) shahram95@shahram95:~/Desktop/HW5_new/python$ python run_q2.py
0.0, 0.09
0.0, 0.07
/home/shahram95/Desktop/HW5_new/python/nn.py:32: RuntimeWarning: overflow encountered in exp
  res = 1/(1+np.exp(-x))
should be zero and one   0.0 1.0
(40, 25)
```

Q2.2.2:

Ans: Coded in python, submitted in the zipped file submission. Executing the run_q2.py we get:

```
(sam_env) shahram95@shahram95:~/Desktop/HW5_new/python$ python run_q2.py
0.0, 0.07
0.0, 0.06
/home/shahram95/Desktop/HW5_new/python/nn.py:32: RuntimeWarning: overflow encountered in exp
  res = 1/(1+np.exp(-x))
should be zero and one   0.0 1.0
(40, 25)
0.07853725353087682 0.9999999999999998 1.0000000000000002 (40, 4)
```

Q2.2.3:

Ans: Coded in python, submitted in the zipped file submission. Executing the run_q2.py code we get:

```
(sam_env) shahram95@shahram95:~/Desktop/HW5_new/python$ python run_q2.py
0.0, 0.08
0.0, 0.07
/home/shahram95/Desktop/HW5_new/python/nn.py:32: RuntimeWarning: overflow encountered in exp
  res = 1/(1+np.exp(-x))
should be zero and one   0.0 1.0
(40, 25)
0.09384733378872284 0.9999999999999998 1.0000000000000002 (40, 4)
55.358174399303465, 0.28
Wlayer1 (2, 25) (2, 25)
Woutput (25, 4) (25, 4)
blayer1 (25,) (25,)
boutput (4,) (4,)
(sam_env) shahram95@shahram95:~/Desktop/HW5_new/python$
```

Q2.3.1.

Ans: Coded in python, submitted in the zipped file submission. Executing run_q2.py we get:

Q2.4:

Ans: Coded in python, submitted in the zipped file submission.

```
(sam_env) shahram95@shahram95:~/Desktop/HW5_new/python$ python run_q2.py
0.0, 0.06
0.0, 0.07
/home/shahram95/Desktop/HW5_new/python/nn.py:32: RuntimeWarning: overflow encountered in exp
  res = 1/(1+np.exp(-x))
should be zero and one   0.0 1.0
(40, 25)
0.14843515921396216 0.9999999999999998 1.0000000000000002 (40, 4)
55.8000892422008, 0.28
Wlayer1 (2, 25) (2, 25)
Woutput (25, 4) (25, 4)
blayer1 (25,) (25,)
boutput (4,) (4,)
[5, 5, 5, 5, 5, 5, 5, 5]
itr: 00          loss: 54.59     acc : 0.28
itr: 100         loss: 42.50     acc : 0.75
itr: 200         loss: 35.39     acc : 0.78
itr: 300         loss: 30.93     acc : 0.78
itr: 400         loss: 28.02     acc : 0.80
(sam_env) shahram95@shahram95:~/Desktop/HW5_new/python$
```

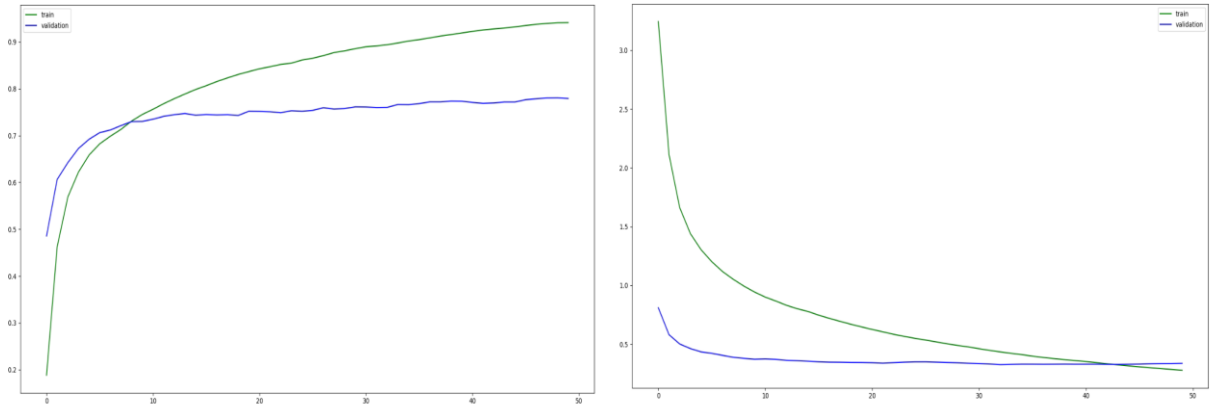
Q2.5:

Ans: Coded in python, submitted in the zipped file submission. Executing the run_q2.py code we get:

```
(sam_env) shahram95@shahram95:~/Desktop/HW5_new/python$ python run_q2.py
0.0, 0.08
0.0, 0.07
/home/shahram95/Desktop/HW5_new/python/nn.py:32: RuntimeWarning: overflow encountered in exp
  res = 1/(1+np.exp(-x))
should be zero and one   0.0 1.0
(40, 25)
0.05009253539239516 0.9999999999999998 1.0000000000000002 (40, 4)
55.83333022113523, 0.25
Wlayer1 (2, 25) (2, 25)
Woutput (25, 4) (25, 4)
blayer1 (25,) (25,)
boutput (4,) (4,)
[5, 5, 5, 5, 5, 5, 5, 5]
itr: 00          loss: 66.53     acc : 0.25
itr: 100         loss: 43.56     acc : 0.65
itr: 200         loss: 37.36     acc : 0.78
itr: 300         loss: 33.44     acc : 0.78
itr: 400         loss: 31.08     acc : 0.78
grad_Woutput 4.52e-06
grad_boutput 2.77e-07
grad_Wlayer1 1.20e-06
grad_blayer1 1.13e-06
total 7.12e-06
```

Q3.1:

Ans: Using a single hiddenlayer with 64 hidden units, and training the network for 50 epochs, batch size of 32, and learning rate = 0.009, we get:

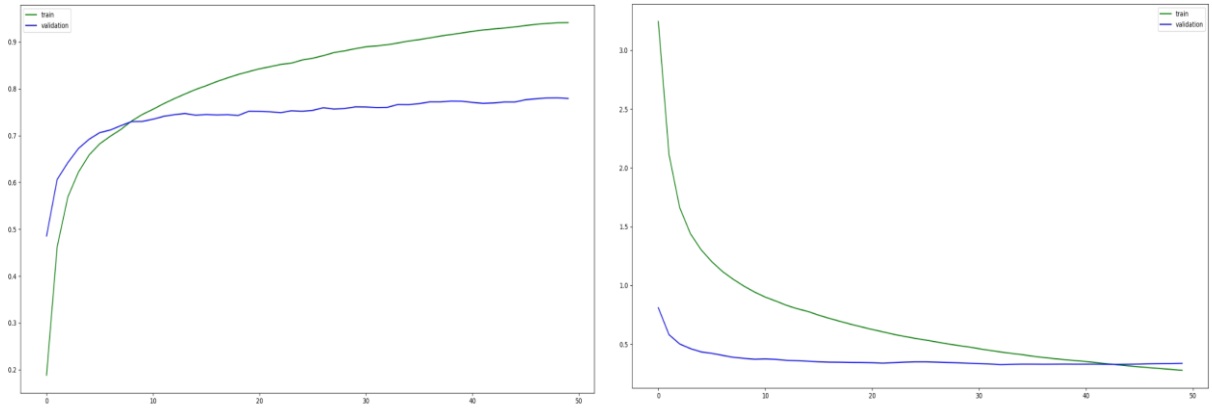


And following is the validation accuracy coming out to ~78.25%.

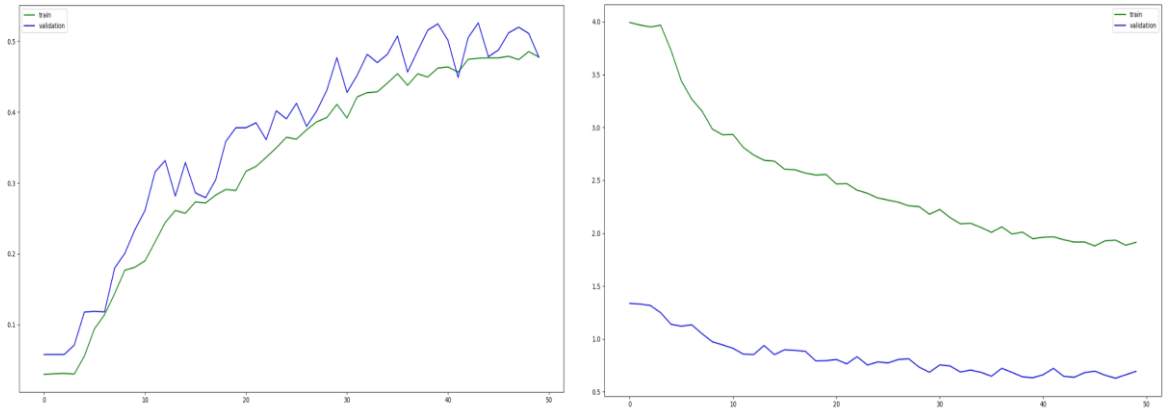
```
(sam_env) shahram95@shahram95:~/Desktop/HW5_new/python$ python run_q3.py
itr: 00         loss: 3.25      acc : 0.19
itr: 02         loss: 1.66      acc : 0.56
itr: 04         loss: 1.30      acc : 0.65
itr: 06         loss: 1.12      acc : 0.70
itr: 08         loss: 0.99      acc : 0.74
itr: 10         loss: 0.90      acc : 0.76
itr: 12         loss: 0.83      acc : 0.78
itr: 14         loss: 0.78      acc : 0.79
itr: 16         loss: 0.72      acc : 0.81
itr: 18         loss: 0.67      acc : 0.82
itr: 20         loss: 0.62      acc : 0.84
itr: 22         loss: 0.58      acc : 0.85
itr: 24         loss: 0.55      acc : 0.86
itr: 26         loss: 0.52      acc : 0.87
itr: 28         loss: 0.49      acc : 0.88
itr: 30         loss: 0.46      acc : 0.88
itr: 32         loss: 0.43      acc : 0.89
itr: 34         loss: 0.41      acc : 0.90
itr: 36         loss: 0.39      acc : 0.90
itr: 38         loss: 0.37      acc : 0.91
itr: 40         loss: 0.35      acc : 0.92
itr: 42         loss: 0.33      acc : 0.92
itr: 44         loss: 0.31      acc : 0.93
itr: 46         loss: 0.30      acc : 0.93
itr: 48         loss: 0.28      acc : 0.94
Validation accuracy:  0.7825
```

Q3.2:

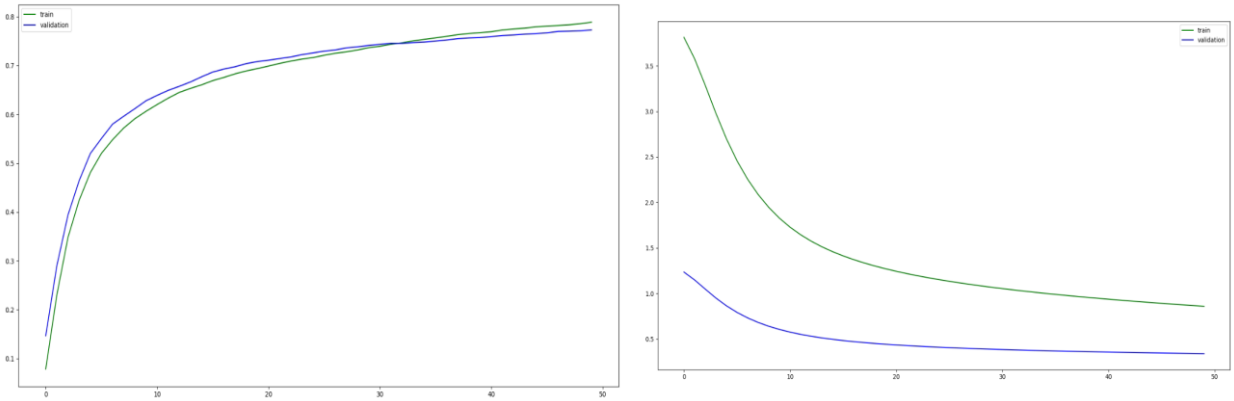
Ans: The best validation accuracy I got was ~78.25% using learning rate 0.009, batch size 32, and 50 epochs. Following are the loss and accuracy plots for the same:



Using a learning rate 10 times the best learning rate i.e. 0.09, following are the loss and accuracy plots for the same:



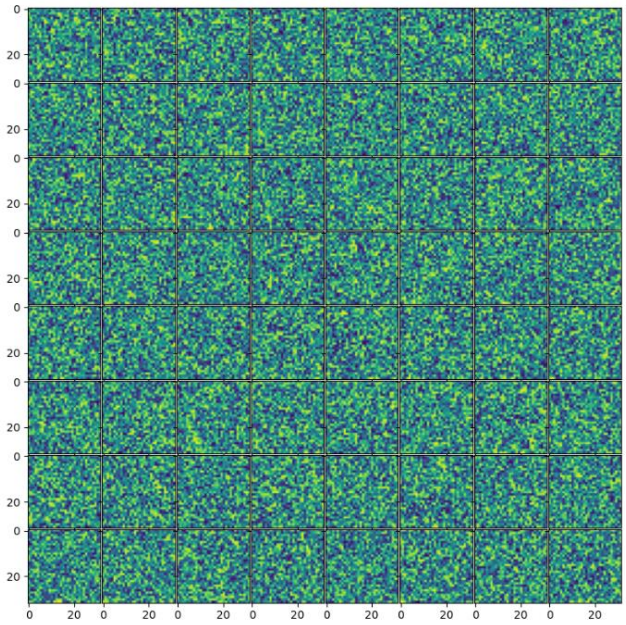
Using a learning rate 1/10th times the best learning rate i.e. 0.009, following are the loss and accuracy plots for the same:



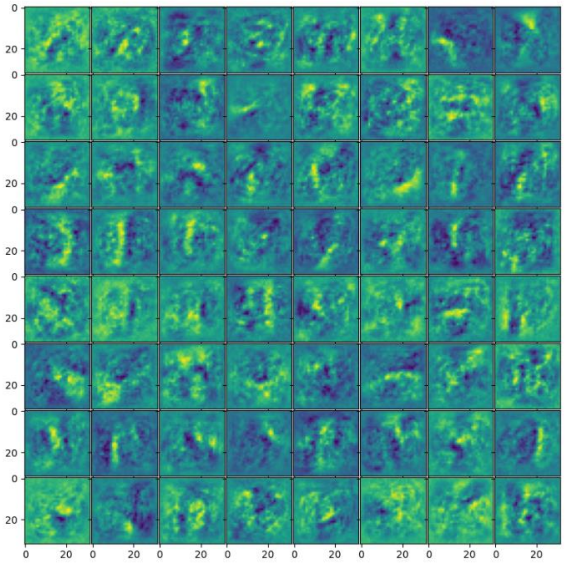
Multiplying the best performing learning rate by 10 leads to rapid initial convergence but results in overshooting the minima, causing instability and divergence in the training process due to excessively large update steps. Conversely, reducing the best performing learning rate by $1/10^{th}$ significantly slows down convergence leading to a smoother convergence, which increases the training time to optimal model convergence. The best network is for learning rate 0.009, and **hence the final accuracy on the test set is ~75%**.

Q3.3:

Ans: Following is the visualization of the initialized network weights:



Following are the first layer weights after training the network:

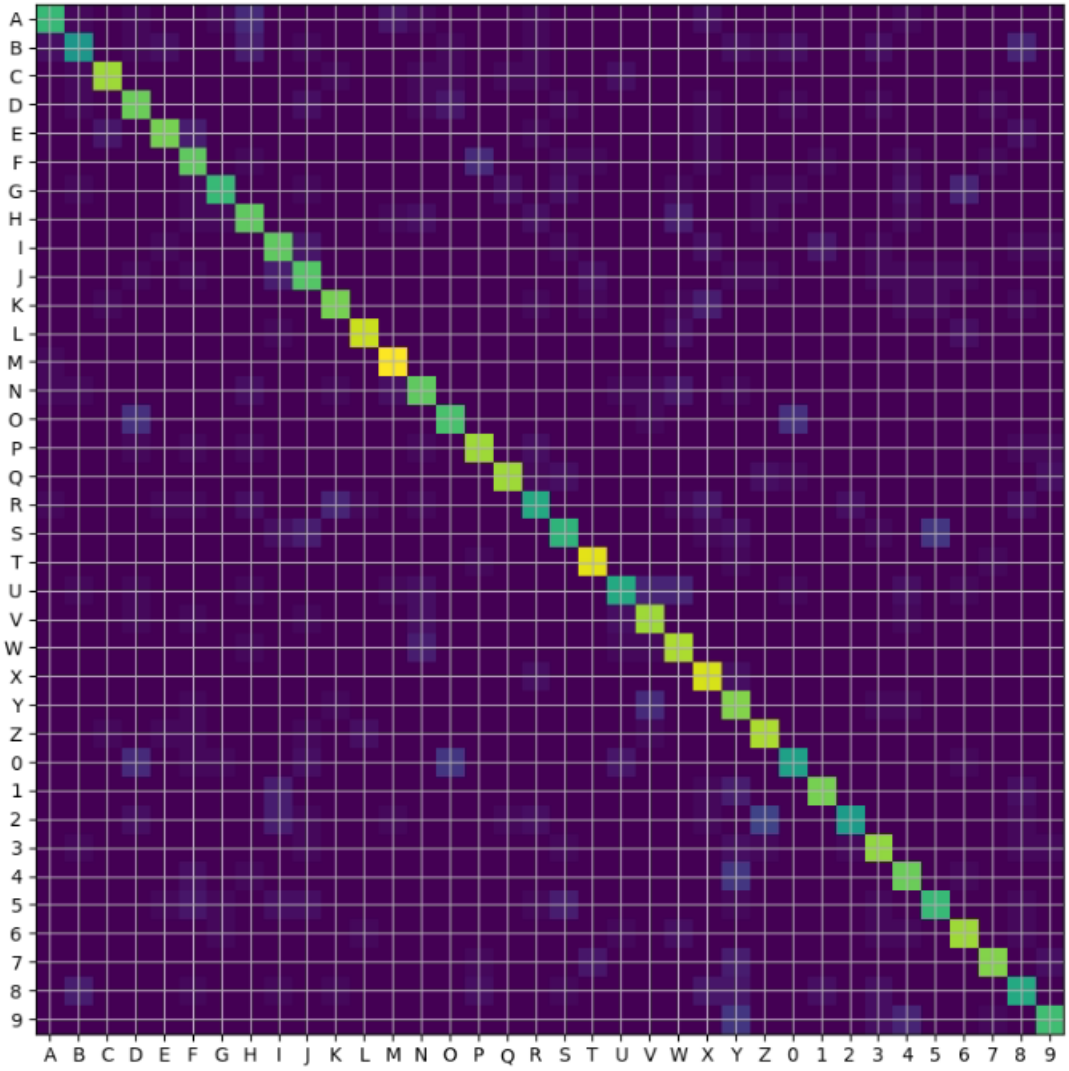


The comparison between the initialized weights and those after training on the NIST dataset reveals a clear transition from randomness to structured patterns. When initialized, the weights display a high-entropy pattern typical of random initialization, which is crucial for breaking symmetry and allowing diverse feature learning across neurons. After training, the weights exhibit distinct localized features resembling strokes and shapes characterstic of handwritten digits and alphabets, indicating that the network has learned to identify and represent salient features of the dataset highlighting the

fast that the network is now capable of discerning and adapt to the data's underlying structure, optimizing the weights to minimize loss and improve prediction accuracy.

Q3.4:

Ans: Following is the confusion matrix of the test data for my best performing model using learning rate 0.009, batch size 32, and 50 epochs.



From the confusion matrix it is quite evident that the most misclassified classes are letter "O" being misclassified as the number "0", letter "Z" misclassified as the number "2", number "5" misclassified as the letter "S", Letter "D" misclassified as the letter "O", and letter "I" misclassified as number "1". Intuitively this does makes sense since these classes sometimes are nearly impossible to discern even for us humans.

Q4.1:

Ans: The outlined method for text parsing from an image makes several key assumptions that can lead to inaccuracies in character detection:

a. It assumes that each characters in the image can be isolated as a separate connected component. This assumption fails when characters are not well-separated (e.g. in cursive or handwritten text where letters may be joined) or when the image resolution causes breaks within characters, leading to multiple components per character.
b. The method presumes uniformity in character size, font, and stroke width, matching those of the training set, and it assumes that there is no significant noise, such as smudges or artifacts that could be misinterpreted as characters. It also assumes that there is no overlapping text, such as in densely packed or layered writing, which can cause multiple characters to be detected as a single component.

Following are the two case images where I would expect the detection to fail:

Figure: Non-uniform strokes with gaps/voids

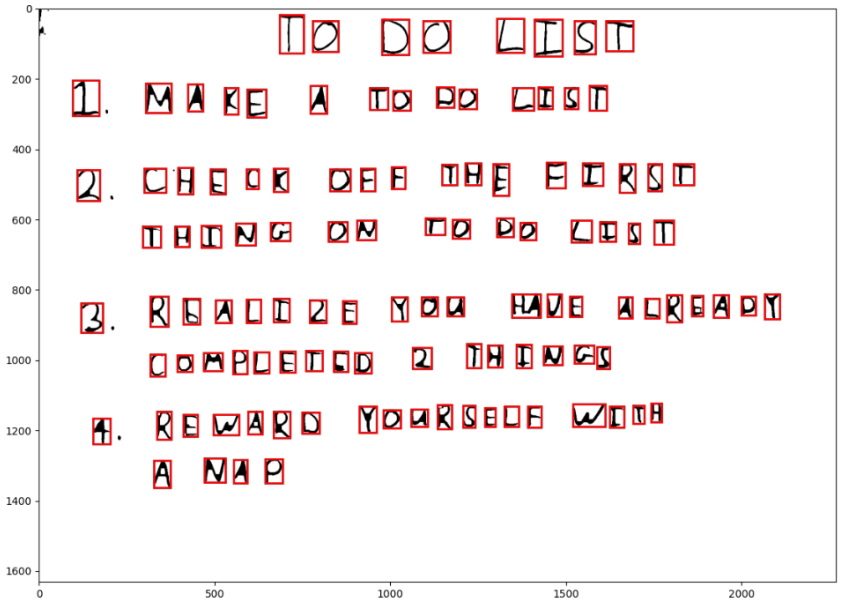
Figure: Connected letters/alphabets

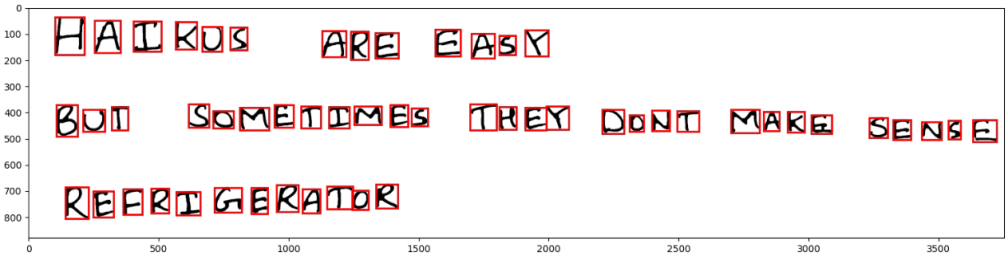
Q4.2:

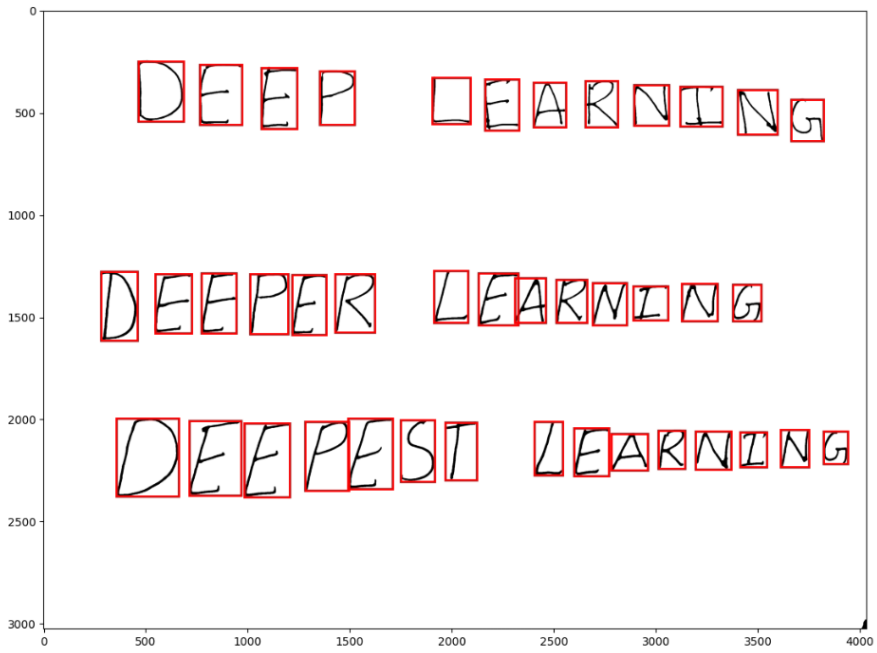
Ans: Implemented as python code and added in the zip folder submission.

Q4.3:

Ans: Following are the results of the findLetters(..) function (implemented in q4.py):






DEEP LEARNING
DEEPER LEARNING
DEEPEST LEARNING


A B C D E F G
H I J K L M N
O P Q R S T U
V W X Y Z
1 2 3 4 5 6 7 8 9 0

Q4.4:

Ans: Following are the results of the classified letters based on the detected alphabets (in the previous question):

```
01_list.jpg
TODQLIST
1MAKEATODQLIST
2CHECKQFFTH44F1RST
THINGQNTQDOLIST
3REALI7E4OUHAVEALREADY
E0MPLET6DZTHINGS
4RFWARDYURSELFWITB
ANAP
```

Accuracy: ~89%

```
03_haiku.jpg
HAIKUSARGGASY
VWTSQNETUNEGTHEXDDWTNAJESENGE
RBFRIGBRATQR
```

Accuracy: ~75%

```
04_deep.jpg
DEEPLE2RNING
DEEPERLEARNING
DEEPESTLEARNING
```

Accuracy: ~98%

```
02_letters.jpg
ABCDGF6
HIIKLMN
OPQRSTU
VWXYZ
1Z3G56789D
```

Accuracy: ~89%

Q5.1.1:

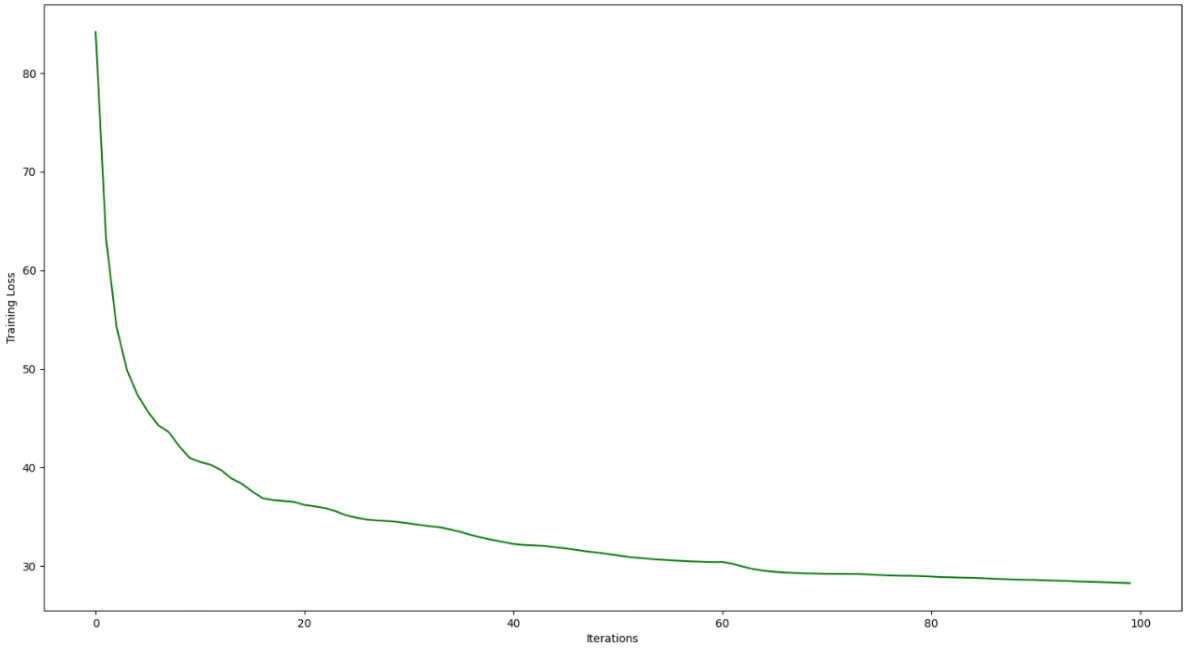
Ans: Implemented as python code and added in the zip folder submission.

5.1.2:

Ans: Implemented as python code and added in the zip folder submission.

5.2:

Ans: Following is the training loss graph for the default parameters of batch size 32 and learning rate of 3e-5:

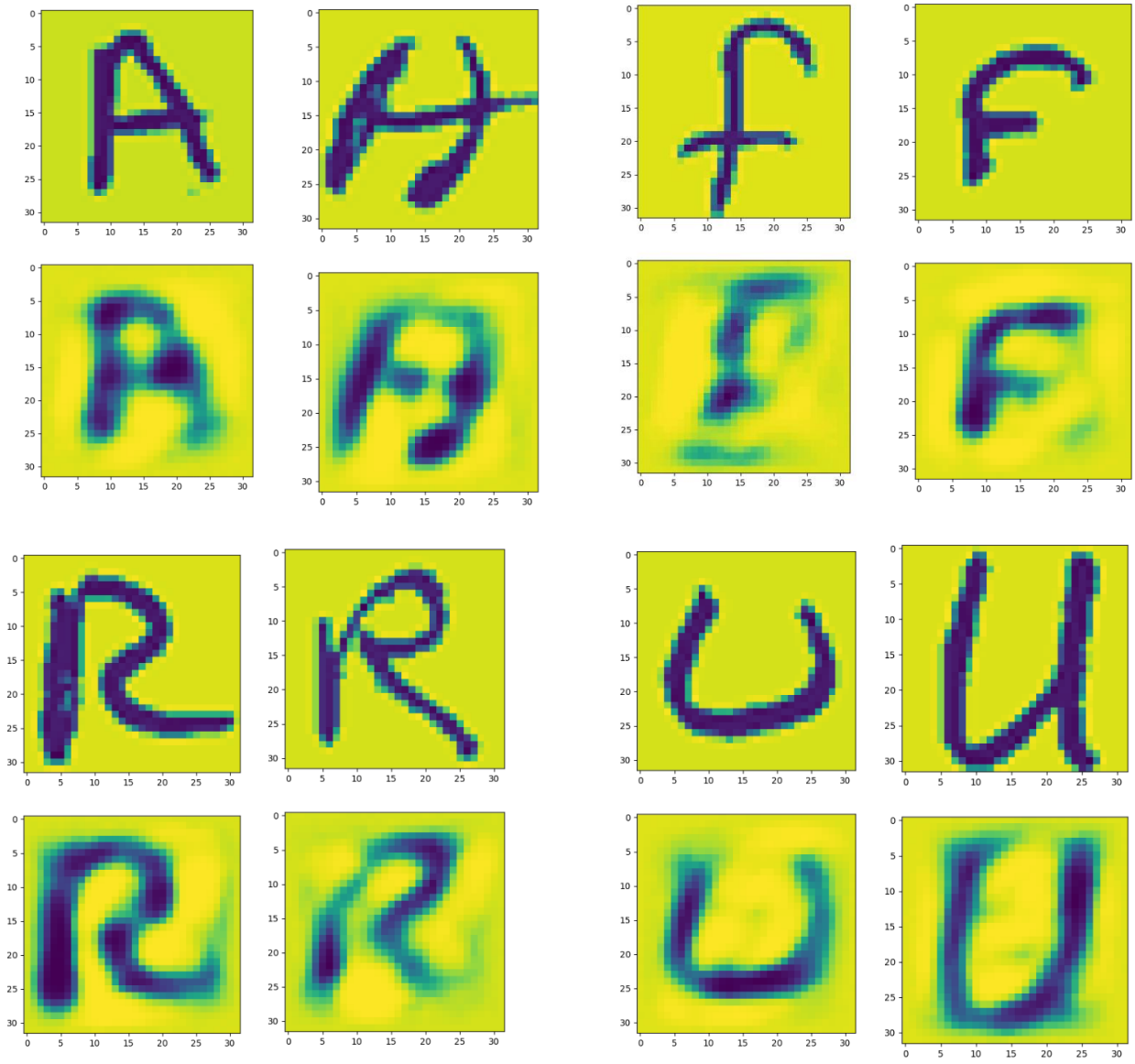


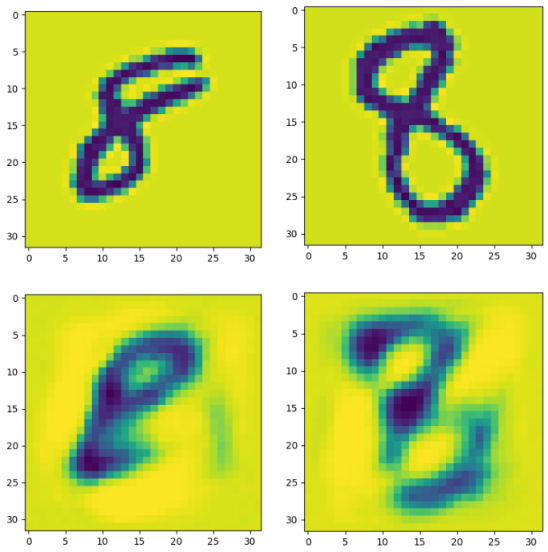
As observed above:

a. The training loss rapidly decreases at the start, which indicates that the model's initial weights are adjusting significantly in response to the gradient optimization, which signifies that the model moves from a high-entropy state towards a more organized state that captures the features of the input data.
b. As the training progresses, the slope of the curve flattens, indicating diminishing returns on loss reduction with each iteration. This is consistent with approaching a minima in the cost function graph.
c. The curve plateaus as it approaches 100 iterations, which suggests that it has approached an asymptote, beyond which there might be minimal improvements in loss reduction.
d. The smooth and monotonically decreasing nature of the curve suggests that the optimizer is not trapped in the local minima (if it exists).

Q5.3.1:

Ans:





From the above results we can comment:

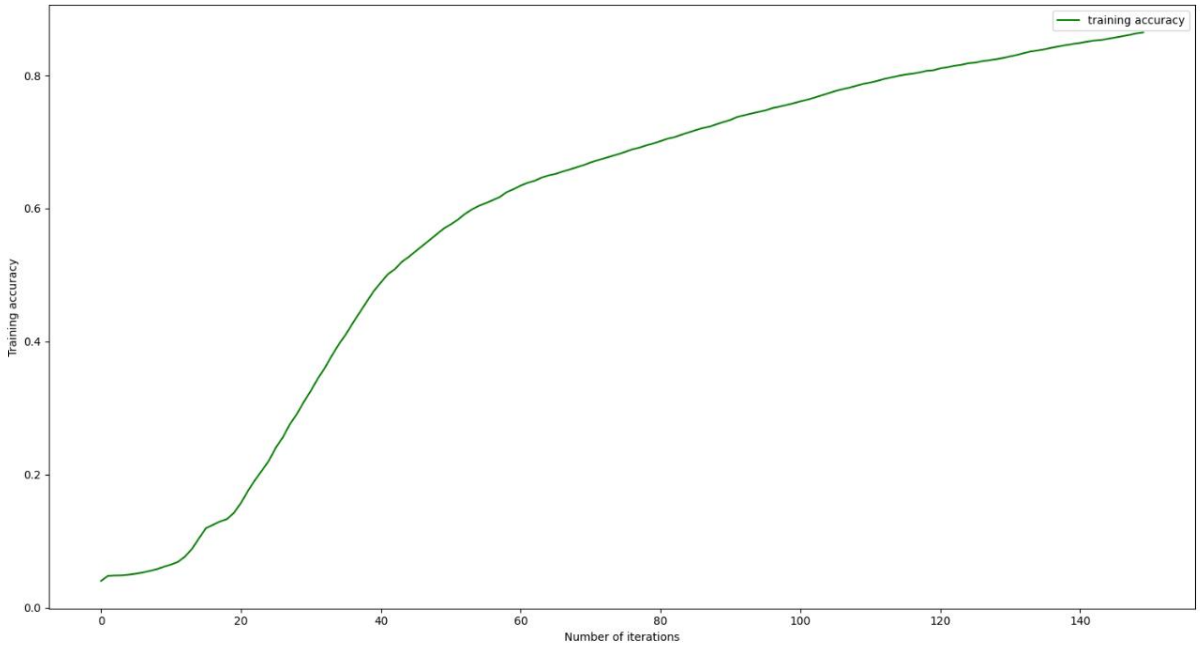
a. The blurred reconstructed images due loss of sharpness, clarity, softened or smeared edges might be due to the autoencoder not learning enough features during the training process, but just enough to reconstruct a rough outline of the alphabets and numbers in the form of low-fidelity images.
b. The autoencoder has managed to capture only the most basic structural aspects of the digits and alphabets and can be particularly seen struggling whenever there are discontinuities in handwritten alphabets and numbers.

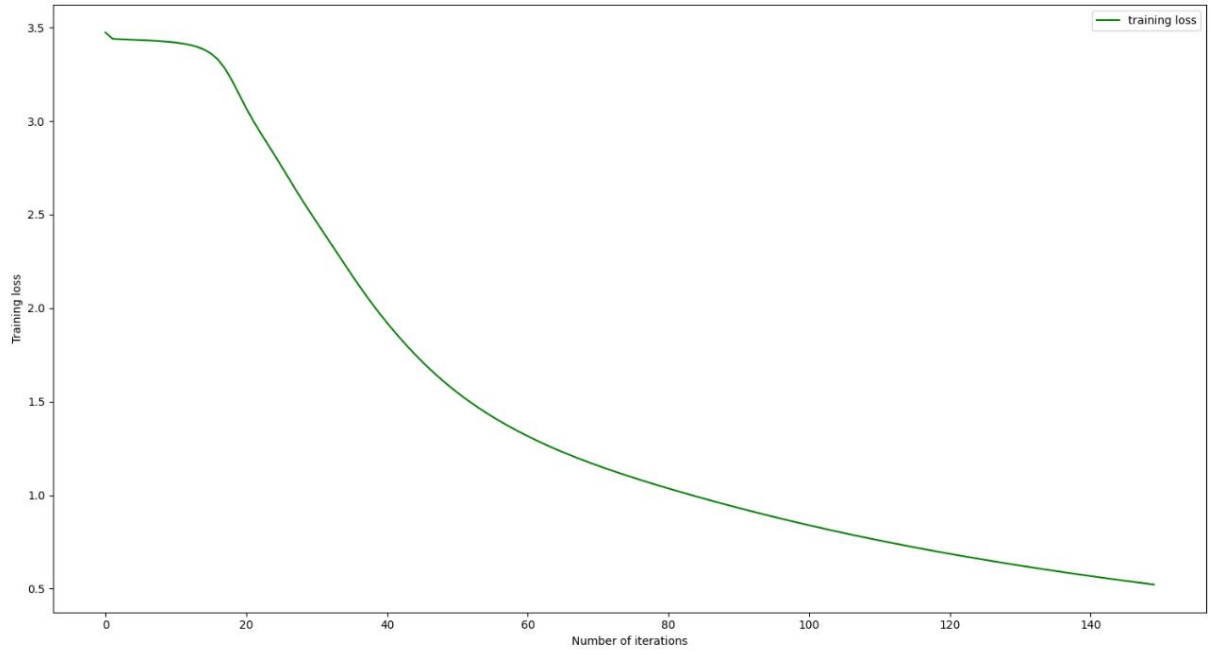
Q5.3.2:

Ans: With the default hyperparameters (of batch size 32 and learning rate 3e-5) the average PSNR we get from the autoencoder across all images in the validation set is ~13.98.

Q6.1.1:

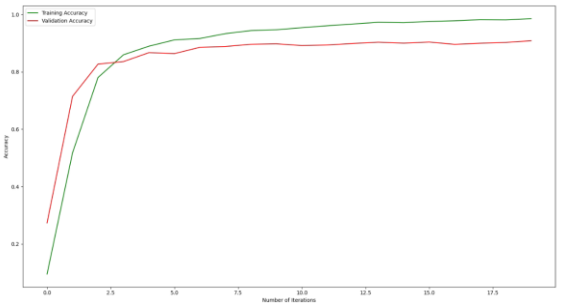
Ans: Following is the plot of training accuracy and training loss with increasing iterations for training a FCN on the NIST36 dataset using PyTorch:

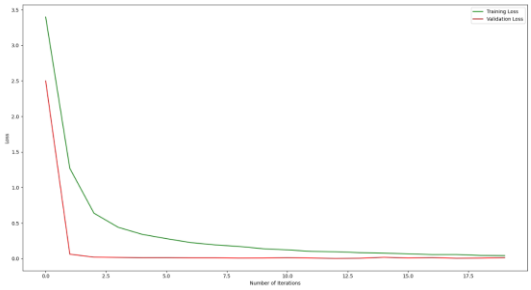




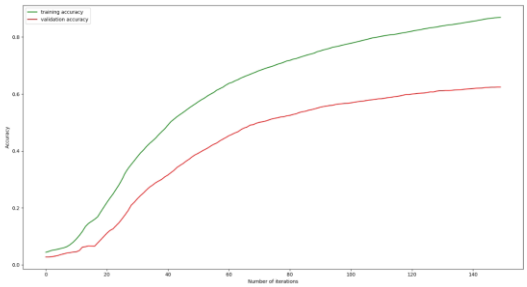
Q6.1.2:

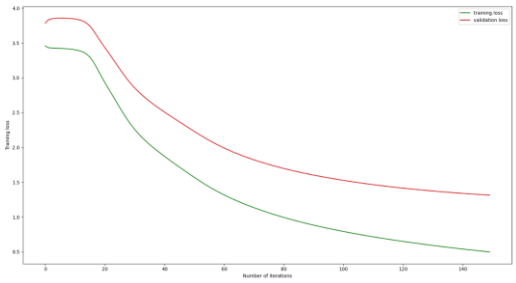
Ans: Following is the training and validation accuracy, and training and validation loss curve for a customized CNN trained on NIST36 dataset:





And following is the training and validation accuracy, and training and validation loss curve for a FCN trained on NIST36 dataset:



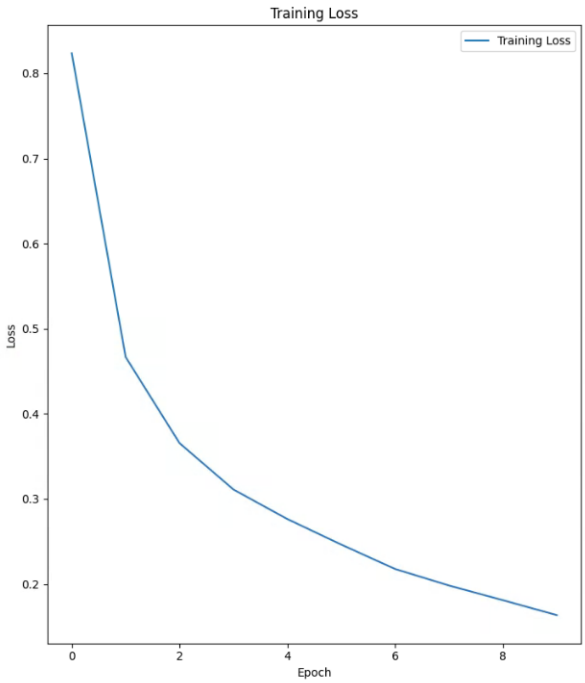


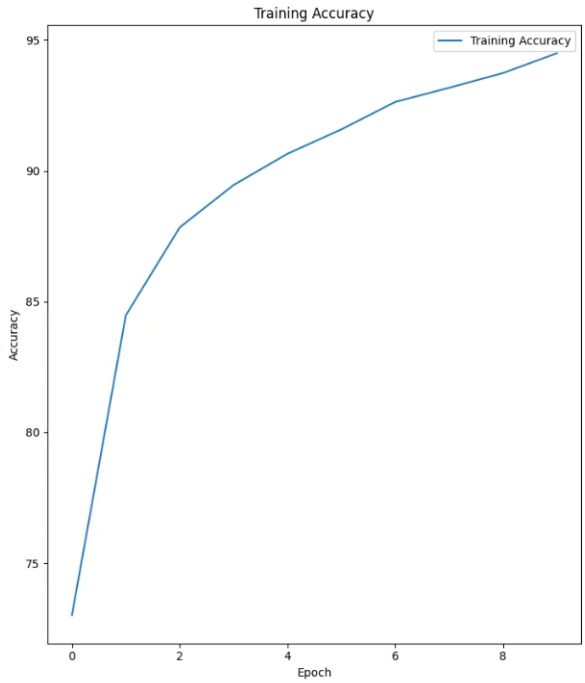
It is evident that the performance of a CNN for this classification task for the particular dataset outperforms the FCN. This conclusion is arrived on using:

a. Faster convergence for CNN
b. Lower validation and training loss for CNN
c. Higher training accuracy achieved within the same number of epochs

Q6.1.3:

Ans: Following are the training loss and accuracy curves for a CNN (MobilNetV2 architecture) for 10 epochs on the CIFAR10 dataset:



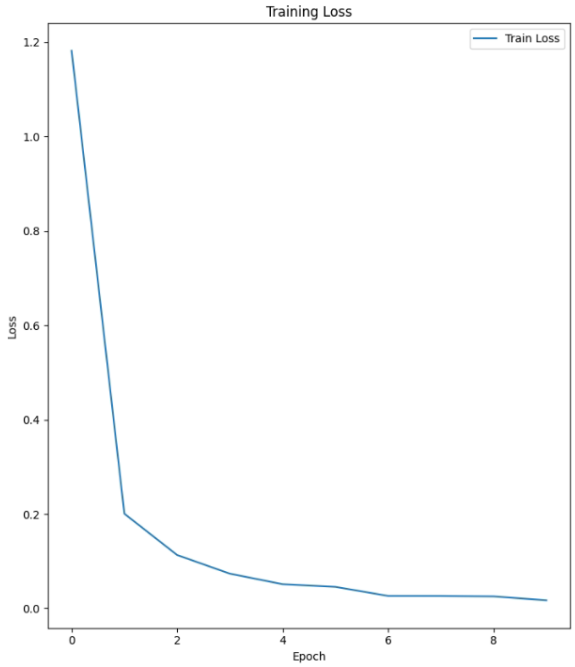


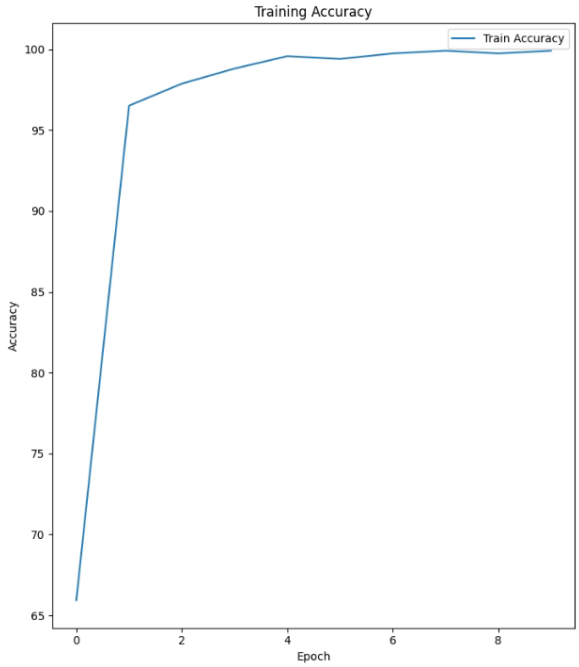
Following is the accuracy of the trained model on the test dataset.

```
Test Loss: 0.0581, Test Accuracy: 93.28%
(sam_env) shahram95@shahram95:~/Desktop/HW5_new/python$
```

Q6.1.4:

Ans: Following are the curves for training loss and training accuracy for the SUN database provided in HW1:





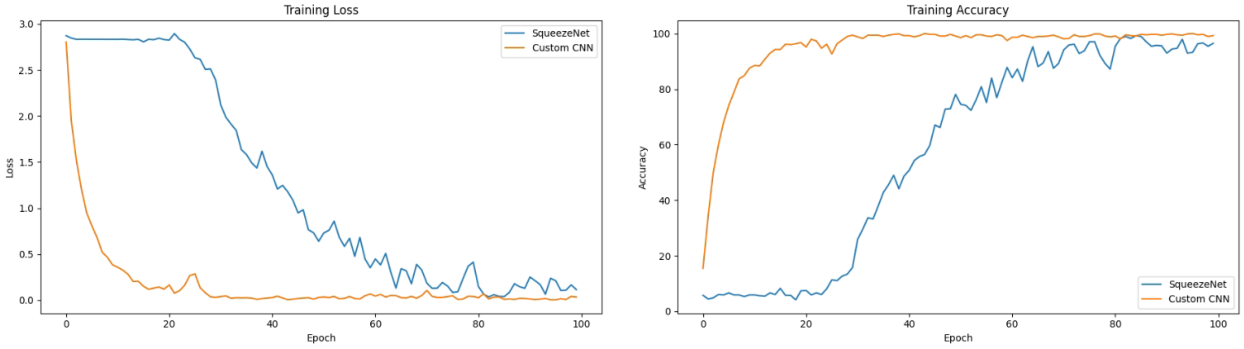
Following is the accuracy and loss we get on the test set of the SUN database (for HW1)

```
(sam_env) shahram95@shahram95:~/Desktop/HW5_new/python$ python run_q6_1_4.py
/home/shahram95/anaconda3/envs/sam_env/lib/python3.9/site-packages/torchvision/models/_utils.py:208: UserWarning: The parameter 'pretrained' is deprecated since 0.13 and may be removed in the future, ple
ase use 'weights' instead.
  warnings.warn(
/home/shahram95/anaconda3/envs/sam_env/lib/python3.9/site-packages/torchvision/models/_utils.py:223: UserWarning: Arguments other than a weight enum or `None` for 'weights' are deprecated since 0.13 and
may be removed in the future. The current behavior is equivalent to passing `weights=MobileNet_V2_Weights.IMAGENET1K_V1`. You can also use `weights=MobileNet_V2_Weights.DEFAULT` to get the most up-to-dat
e weights.
  warnings.warn(msg)
Epoch [1/10], Loss: 1.1818, Accuracy: 65.93%
Epoch [2/10], Loss: 0.2007, Accuracy: 96.52%
Epoch [3/10], Loss: 0.1132, Accuracy: 97.88%
Epoch [4/10], Loss: 0.0737, Accuracy: 98.81%
Epoch [5/10], Loss: 0.0514, Accuracy: 99.58%
Epoch [6/10], Loss: 0.0457, Accuracy: 99.41%
Epoch [7/10], Loss: 0.0265, Accuracy: 99.75%
Epoch [8/10], Loss: 0.0264, Accuracy: 99.92%
Epoch [9/10], Loss: 0.0255, Accuracy: 99.75%
Epoch [10/10], Loss: 0.0173, Accuracy: 99.92%
Test Loss: 0.0767, Test Accuracy: 98.00%
```

In HW1, my best performing model (with the data augmentation to increase the training images) using BoW to classify the scenes came out to be 80.5%. But with very little effort and using an off-the-shelf low-end backbone architecture of MobilNetV2 I am able to achieve 98.00% accuracy seamlessly.

Q6.2.1:

Ans: Following is the juxtaposition of the training loss and accuracy curves for SqueezeNet and Custom CNN trained on flower17 dataset:



Following are the accuracy achieved for both the models after training (on the test set):

```
Comparison:
SqueezeNet Accuracy: 66.17647058823529%
Custom CNN Accuracy: 50.88235294117647%
(sam_env) shahram95@shahram95:~/Desktop/HW5_new/python$
```

It is evident that finetuning the model using pretrained weights of SqueezeNet generalized better to the dataset as compared to the CustomCNN which was trained from scratch.