Machine Learning

Assignment 3

Sahil Goyal

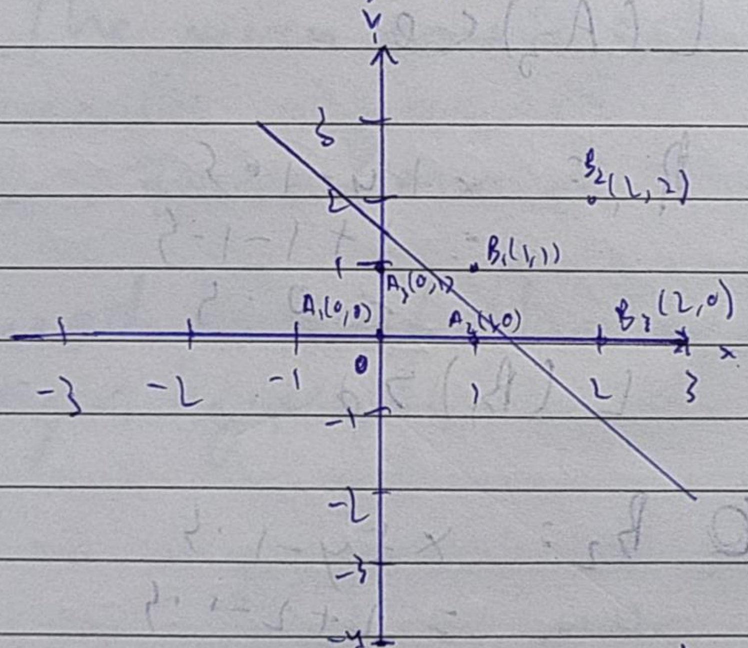
2020326

Section A:

Machine Learning
Assignment 3
Section A

1.

1. Let $A_1 = (0,0)$
   $A_2 = (1,0)$
   $A_3 = (0,1)$
   $B_1 = (1,1)$
   $B_2 = (2,2)$
   $B_3 = (2,0)$



We can see that the line $x + y - 1.5 = 0$ linearly separates the points of the given classes A and B

~~We can find~~

We can prove this by substituting values in line L

$A_1: x+y-1.5$
$= -1.5$
$\therefore L(A_1) < 0$
$A_2: x+y-1.5$
$= 1-1.5$
$= -0.5$
$L(A_2) < 0$
$A_3: x+y-1.5$
$= 1-1-5$
$= -0.5$
$L(A_3) < 0$

$B_1: x+y-1.5$
$= 1+1-1.5$
$= 0.5$
$L(B_1) > 0$

$B_2: x+y-1.5$
$= 2+2-1.5$
$= 2.5$
$L(B_2) > 0$
$B_3: x+y-1.5$
$= 2-1.5$
$= 0.5$
$L(B_3) > 0$

∴ All points of class A are negative while all points of class B are positive

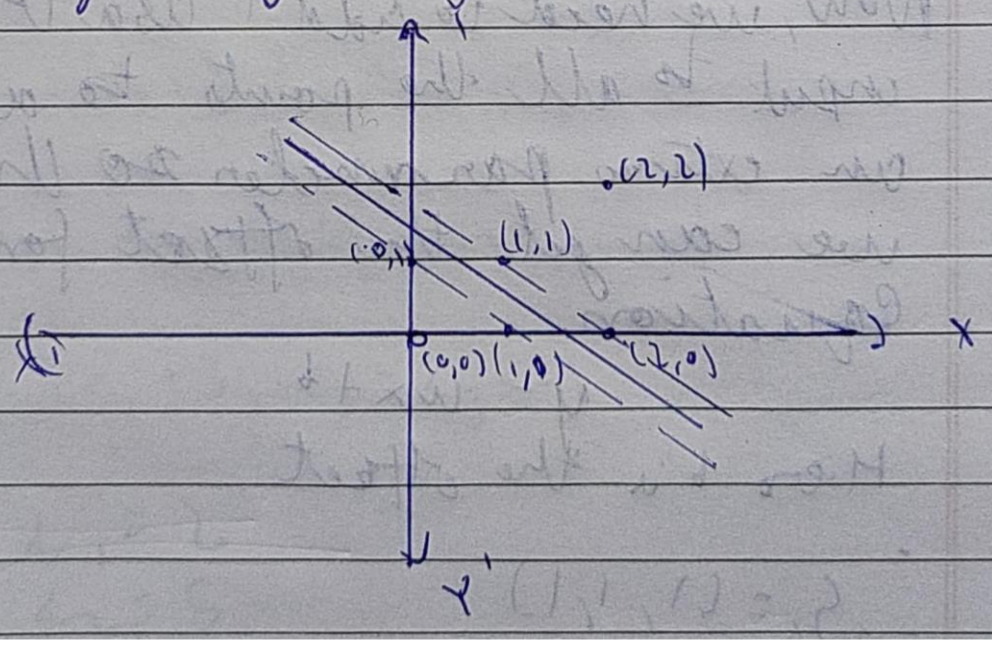
∴ Line $x+y-1.5$ linearly separates the points of class A and B, or points of class A and B lie on opposite sides of line

∴ ~~Classes~~

∴ The given points are linearly separable.

∴ Thus Proved

2. We need to find the maximum margin hyperplane



We can see from graph that our support vectors would be (1,0) for A class and (2,0) and (1,1) for B class

$\therefore S_1 = (1, 1)$
$S_2 = (2, 0)$
$S_3 = (1, 0)$

$\therefore S_1, S_2, S_3$ are our support vectors

~~From my plot, we have calculated that $L(B_1) > 0$, $L(B_3) > 0$ and $L(A_2) < 0$~~

Now, we need to add 1 as a bias input to all the points to add an extra parameter so that we can get the offset for the equation

$$y = wx + b$$

Here b is the offset.

$\therefore S_1 = (1, 1, 1)$
$S_2 = (2, 0, 1)$
$S_3 = (1, 0, 1)$

From Lagrangian method,

$$w = \sum_{i=1}^{n} \alpha_i s_i$$

$\therefore$ From Lagrangian method conditions

$$\sum_{i=1}^{n} \alpha_i s_i s_j = \text{sign of class}$$

$\therefore \alpha_1 s_1 \cdot s_1 + \alpha_2 s_1 \cdot s_2 + \alpha_3 s_1 \cdot s_3 = 1$

$\therefore \alpha_1 (3) + \alpha_2 (3) + \alpha_3 (2) = 1$

$\therefore 3\alpha_1 + 3\alpha_2 + 2\alpha_3 = 1$

$\alpha_1 s_2 s_1 + \alpha_2 s_2 s_2 + \alpha_3 s_2 s_3 = 1$

$\therefore 3\alpha_1 + 5\alpha_2 + 3\alpha_3 = 1$

$\alpha_1 s_3 s_1 + \alpha_2 s_3 s_2 + \alpha_3 s_3 s_3 = -1$

$\therefore 2\alpha_1 + 3\alpha_2 + 2\alpha_3 = -1$

$\therefore \alpha_1 = 2$

$\alpha_2 = 5$

$\alpha_3 = -10$

$w = \sum_{i=1} \alpha_i s_i$

$$w = 2\begin{bmatrix}1\\1\\1\end{bmatrix} + 5\begin{bmatrix}2\\0\\1\end{bmatrix} - 10\begin{bmatrix}1\\0\\1\end{bmatrix}$$

$$w = \begin{bmatrix}2\\2\\-3\end{bmatrix}$$

$\therefore$ Equation of hyperplane is

$w^T x + b$

$= \begin{bmatrix}2 & 2\end{bmatrix}\begin{bmatrix}x_1\\x_2\end{bmatrix} - 3$

$= 2x_1 + 2x_2 - 3$

$\therefore$ $2x_1 + 2x_2 - 3 = 0$ is equation for required hyperplane.

$\therefore$ Distance of points from plane is:

A_1 : $(0,0)$ $\frac{|-3|}{\sqrt{8}} = \frac{3}{2\sqrt{2}}$

$A_2(1,0) = \frac{|2-3|}{\sqrt{8}} = \frac{1}{2\sqrt{2}}$

$A_3(0,1) = \frac{|2-3|}{\sqrt{8}} = \frac{1}{2\sqrt{2}}$

$B_1(1,1) = \frac{|2+2-3|}{\sqrt{8}} = \frac{1}{2\sqrt{2}}$

$B_2(2,2) = \frac{|4+4-3|}{\sqrt{8}} = \frac{5}{2\sqrt{2}}$

$B_3(2,0) = \frac{|4-3|}{\sqrt{8}} = \frac{1}{2\sqrt{2}}$

$\therefore$ The nearest points and support vectors are

$(1,0), (0,1), (1,1) (2,0)$

3. In the previous part, we found that there are 4 support vectors (1,0), (0,1), (1,1), (2,0)
2 support vectors on each side of hyperplane.

For the given points, the optimal margin will increase if we remove the support vectors (1,0)(1,1). The optimal margin won't change if ~~when~~ we remove (2,0)(0,1)

Optimal margin $= \frac{2}{\|w\|}$

$\therefore$ If the hyperplane changes, $\|w\|$ will change and and $2/\|w\|$ will also change, but if hyperplane doesn't change, $\|w\|$ will remain the same and $2/\|w\|$ will also remain the same
For this data, the hyperplane will change upon removing (1,0),(1,1) and it will remain the same upon removing (2,0),(0,1)
$\therefore$ Optimal margin will also change according to hyperplane

4. In general, for any dataset the optimal margin will change according to the support vectors present on the two sides of the plane.

There are multiple cases:

Case 1:

1 side has a single support vector and other side has multiple support vectors.

In this case, if we remove support vector from side with a single support vector, then new nearest point would be calculated which would change the optimal margin.

But if we remove support vector from side with multiple support vectors, there will be no change to the optimal margin as there are multiple support vectors

Case 2:
Both sides only have a single support vector.
In this case, if we remove support vector from either side, a new ~~supp~~ nearest point will be calculated which would change the optimal margin

Case 3:
Both sides have multiple support vectors
In this case, if we remove a support vector from either side it wont change anything as multiple support vectors are present on both sides. Thus, there will be no change to the optimal margin.

Section B:

1.

Explanation of Code:

- After we get the data using mnist.load_data(), we check the data for any null values. We find that there aren't any null values. Then, we plot some of the input images and plot pie charts of the training and testing labels to find the distribution of the data. We find the shape of the data. The input data is of the form of a 28 X 28 matrix, and the output data is a number denoting what type of item it is.
- We reshape the input data to make it simpler to train our model on the data. Now, we have converted the data into an array of length 784. The output data consists of numbers from 0 to 9, so we will also reshape it.
- We will now implement our NeuralNetwork class. While declaring the model, we need to pass parameters like number of layers, size of layers, learning rate, activation function, weight initialisation function, number of epochs and batch size. We also intialise the weights and biases for each layer for the neural network. This is done according to the weight initialisation function which was passed as a parameter.
- Then, we implement the fit() function. Inside the fit function, we use some helper functions like forward(), backward().
- The forward() function is for forward propagation, and it calculates the output of one layer and sends it to the next layer, and this continues till the last layer. This function also calls the predict_proba() function, that utilises the softmax function to calculate the probabilities of each of the ten classes and returns a vector, from which the class with the maximum probability is taken as the output of the final layer.
- The backward() function is for back propagation, and it is responsible for the changes in weights and biases that are made after an iteration. It calculates the gradients of the last layer, and then moves backwards till the first layer.
- In the fit() function, we train the model on the training set. We use forward() to make predictions, and after comparing them with the actual outputs, we use backward() and update the weights and the biases. We also store the training and validation loss during each epoch so that we can plot it later on. We train the model using mini batch gradient descent, by dividing the dataset into multiple batches and then update the weights and biases by training on each batch. For each epoch, we train the model on all the batches.
- We have also implemented the predict() function. It calls forward() function, and gives us the predictions of the input data, which is the result of the last layer calculated by forward() function.
- We have implemented the score() function. It calculates the accuracy of our model which is the division of the number of correct predictions made by our model by the total number of instances in the input data.

2.

Explanation of Code:

- We have implemented the various activation functions and their gradients. These functions were sigmoid, tanh, relu, leaky relu, linear, and softmax.

Code:

```
def sigmoid(self,X):
    return 1/(1+np.exp(-X))

def sigmoid_gradient(self,X):
    return self.sigmoid(X)*(1-self.sigmoid(X))

def tanh(self,X):
    return np.tanh(X)

def tanh_gradient(self,X):
    return 1-np.square(np.tanh(X))

def relu(self,X):
    return np.maximum(0,X)

def relu_gradient(self,X):
    X[X<=0] = 0
    X[X>0] = 1
    return X

def leaky_relu(self,X):
    return np.maximum(0.01*X,X)
```

```python
def leaky_relu_gradient(self,X):
    X[X<=0] = 0.01
    X[X>0] = 1
    return X

def linear(self,X):
    return X

def linear_gradient(self,X):
    return np.ones(X.shape)

def softmax(self,X):
    return np.exp(X-np.max(X))/np.sum(np.exp(X-np.max(X)))

def softmax_gradient(self,X):
    return X*(1-X)
```

3.

Explanation of Code:

- We have implemented the various weight initialisation functions zero_init, random_init, and normal_init (normal(0,1)).

Code:

```python
def zero_init(self, shape):
    return np.zeros(shape)

def random_init(self,shape):
    return np.random.rand(shape[0],shape[1])

def normal_init(self,shape):
    return np.random.normal(0,1,shape)
```

4.

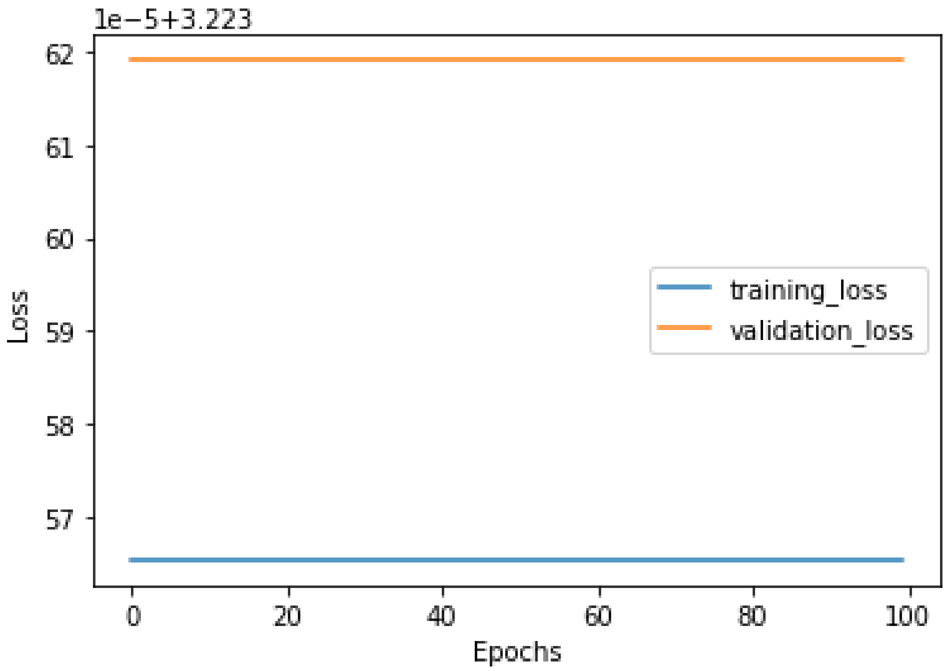
Explanation of Code:

- We have now created multiple models and trained them, and reported the score we are getting using the score() function we created, and also plotting the training and validation loss.

Results and Graphs:

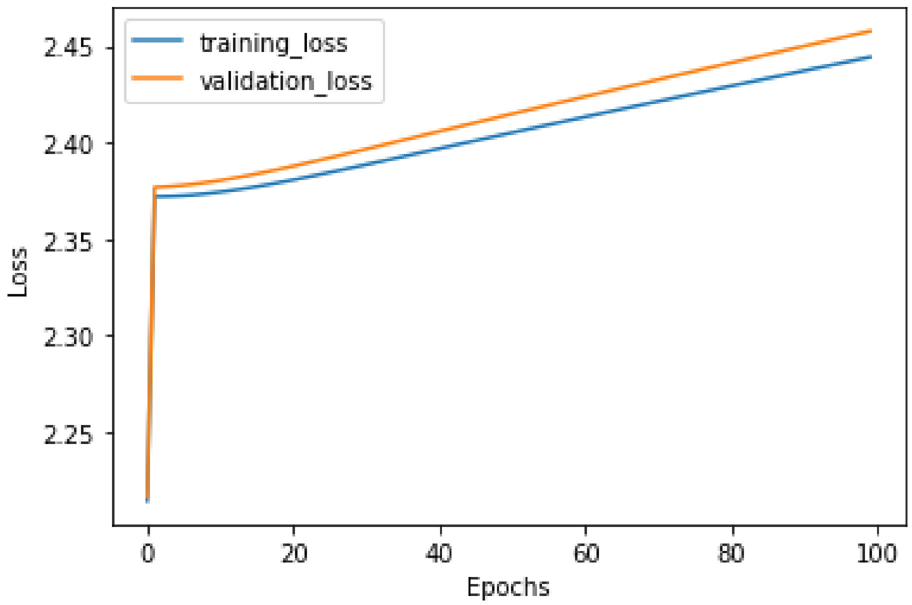
Model 1:

```
0.09685496794871795
```



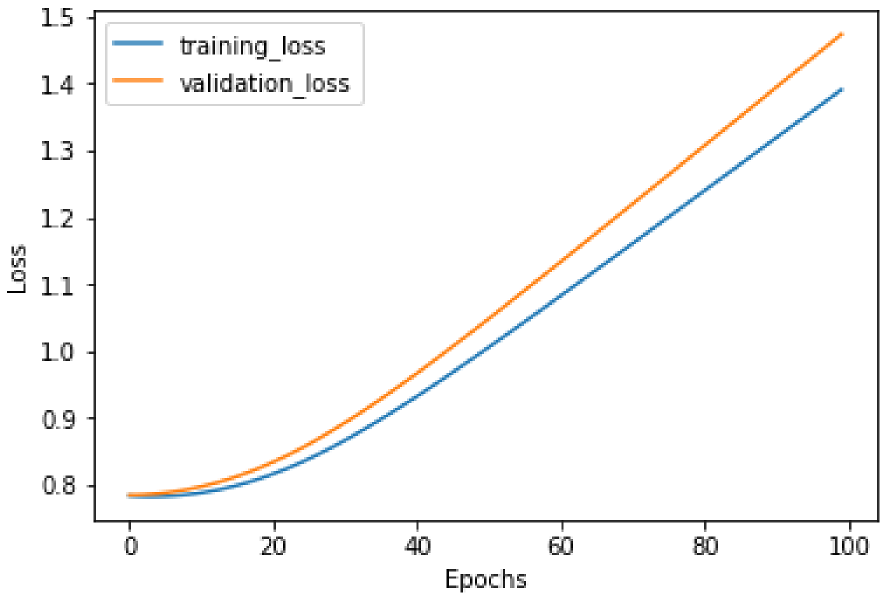
Model 2:

```
0.10857371794871795
```



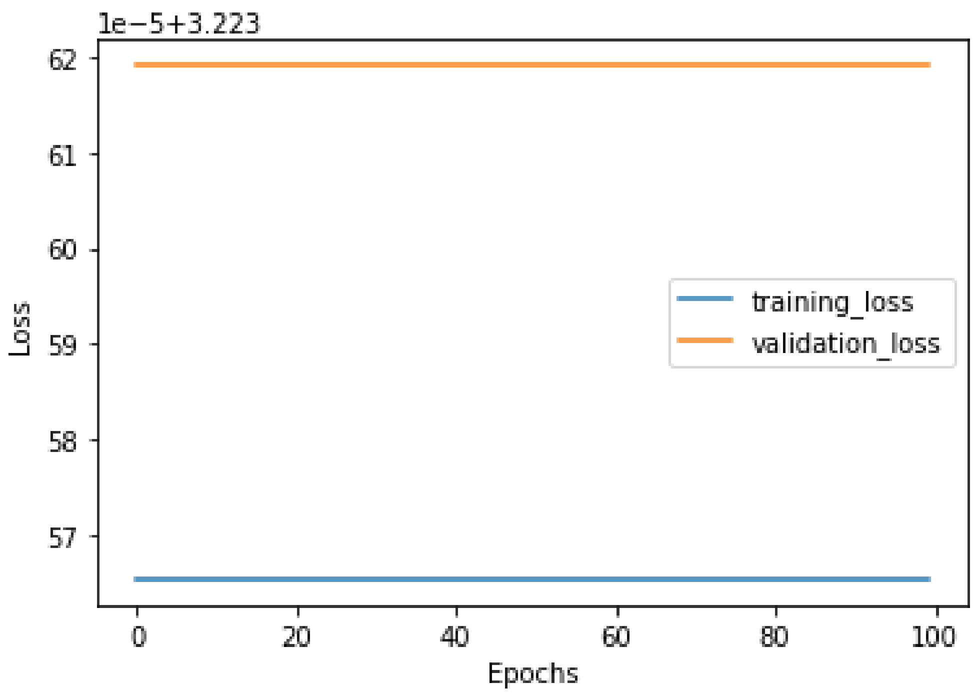
Model 3:

0.10997596153846154



Model 4:

0.09685496794871795



Explanation:

- In model 1, we have used number of layers 6, activation function relu, layers size [784,256,128,64,32,10], learning rate 0.001, weight initialisation function normal, and batch size as 128. Then, we fit the data to train our model and find the score of our data. We get a score of 0.09685496794871795 or 9.685496794871795 %. Thus, we can tell that our model doesn't converge as our accuracy is low. This is because our model is stuck on a local minima and cannot get out of it because of which the accuracy is very low.

- In model 2, we change the learning rate to 0.005, the activation function to sigmoid, the weight initialisation function to random, and the batch size to 64. After that, we repeat the same process as before. We get a score of 0.10857371794871795. We can see that the accuracy is very low again. This is because our model is stuck on a local minima and cannot get out of it again.
- In model 3, we change the learning rate to 0.01, the activation function to tanh, and the weight initialisation function to zero, and the batch size to 256. We get a score of 0.10997596153846154. We can see that the accuracy is very low again. This is because our model is stuck on a local minimum and cannot get out of it again.
- In model 4, we gave changed the learning rate to 0.05, the activation function to leaky relu, the weight initialisation function to random, and the batch size to 128. We get a score of 0.09685496794871795. We can see that the accuracy is very low again. This is because our model is stuck on a local minima and cannot get out of it again.
- Thus, we can see that even after training multiple models with different parameters, we weren't able to get a better accuracy than 10.997596153846154% with tanh activation function, and weight initialisation function as zero, and batch size of 256. This is only slightly better than what we would get if we used a random classifier (10%). This is because every model trained by us was stuck on a local minima and couldn't leave it. This is because Neural network models have very complex functions. They don't have a single local minima but multiple local minimas, because of which it is very easy to get stuck on any local minima and to not be able to reach the global minima.

Section C:

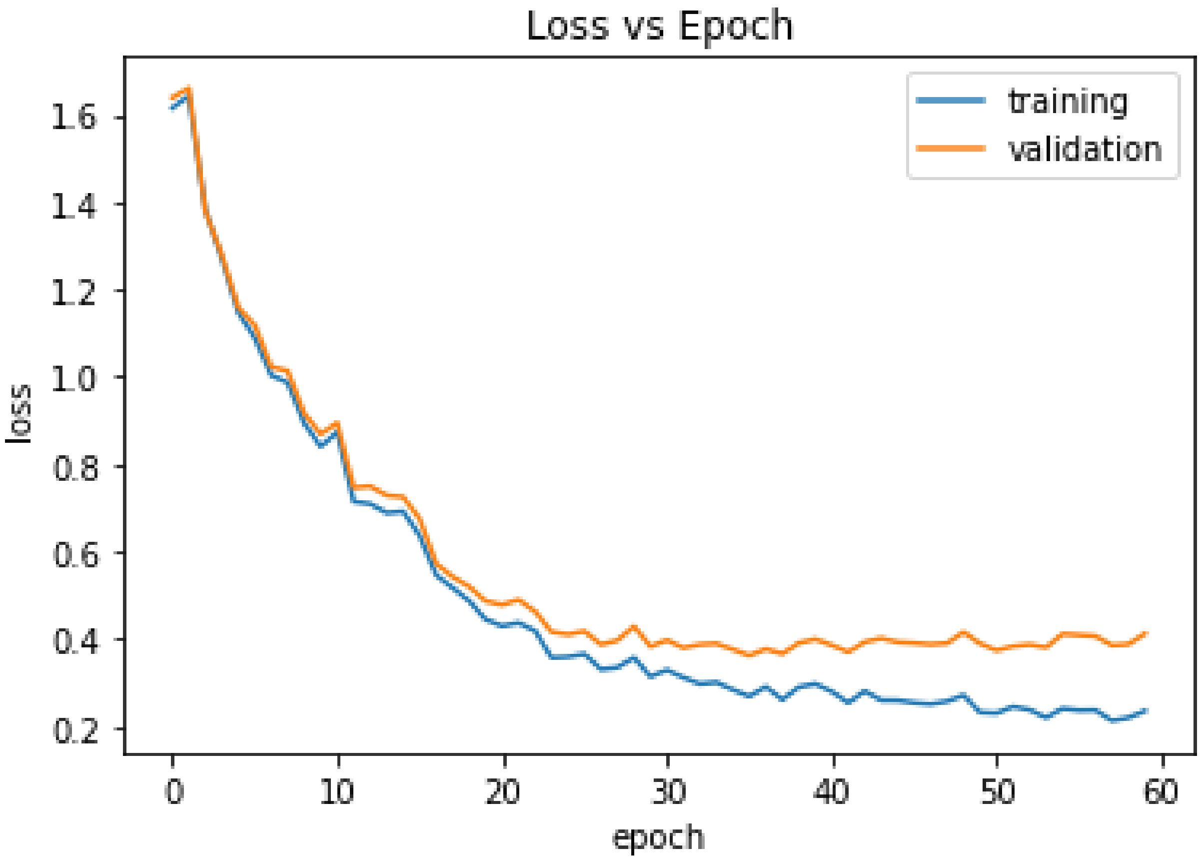
1.

Explanation of Code:

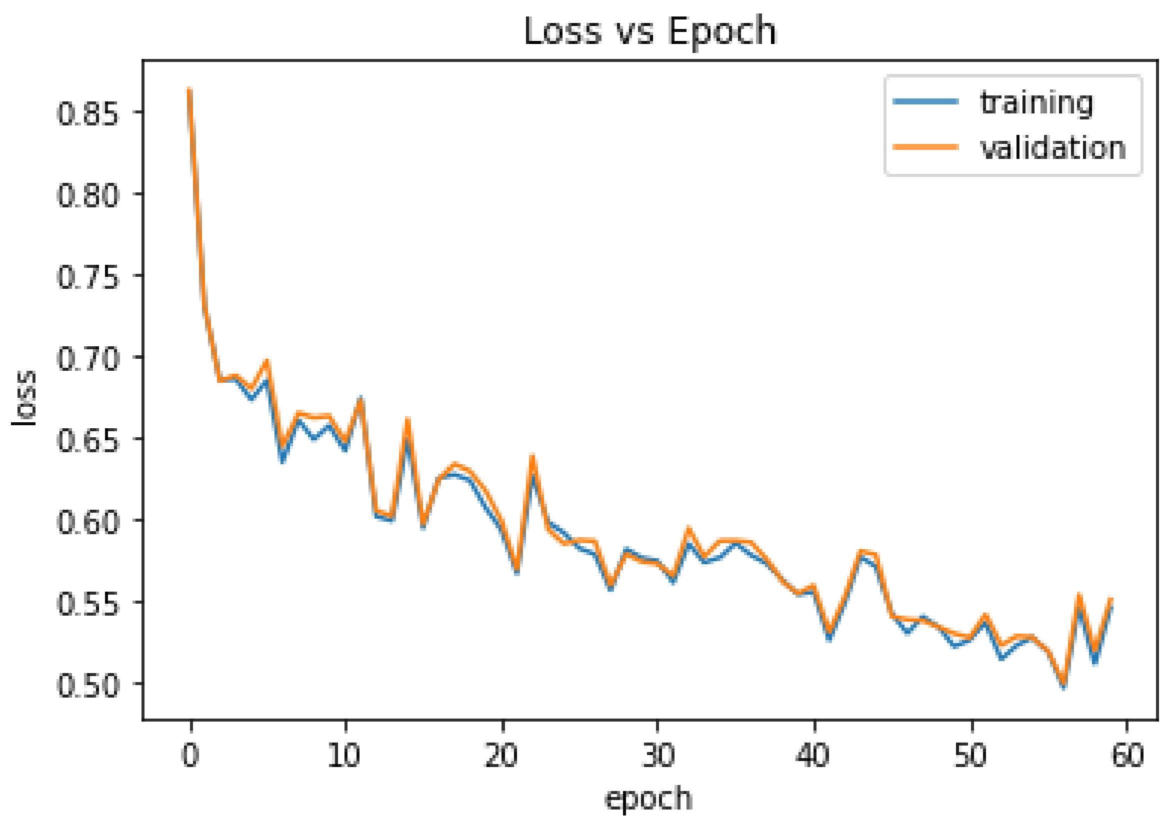
- After we get the data using fashion_mnist.load_data(), we have plotted some of the input images, and then we have plotted a pie chart that shows us that the types are evenly distributed in both the training and the testing data. We have also checked for any null values in the data, and haven't found any.
- We split the training data into a training and validation set. After that, we reshape the input data for the training, testing and validation sets. Earlier, each entry in the input data was a matrix of dimensions 28 X 28, but now they are an array of length 784. We have done this because the sklearn library cannot train the model on matrix inputs.
- After that, we start training our models. First, we train the data on the ReLu activation function. We set the hidden layers to (256, 32) as told. We have taken the number of epochs as 60 because the model starts to slightly overfit just before that. We can see this, because the training loss keeps decreasing, and the validation loss starts to increase. We have taken a batch size of 256, and we start training our data.
- At each epoch, we add the current value of training and validation loss to a list, and then when the model is trained, we plot the training and validation loss curves against epochs in a single plot.
- We also print the score the model gets on the training and validation data.
- We repeat the same process with the sigmoid, tanh, and linear functions.

Graphs:

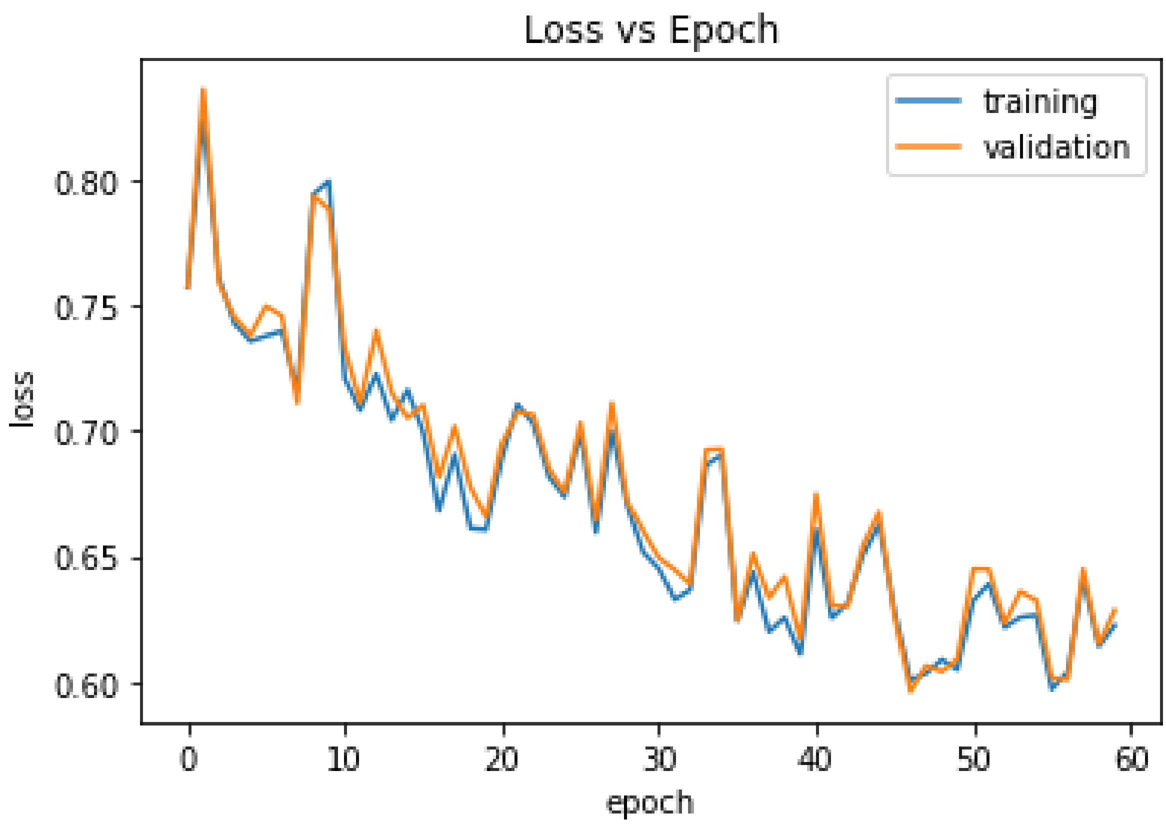
Relu activation function:



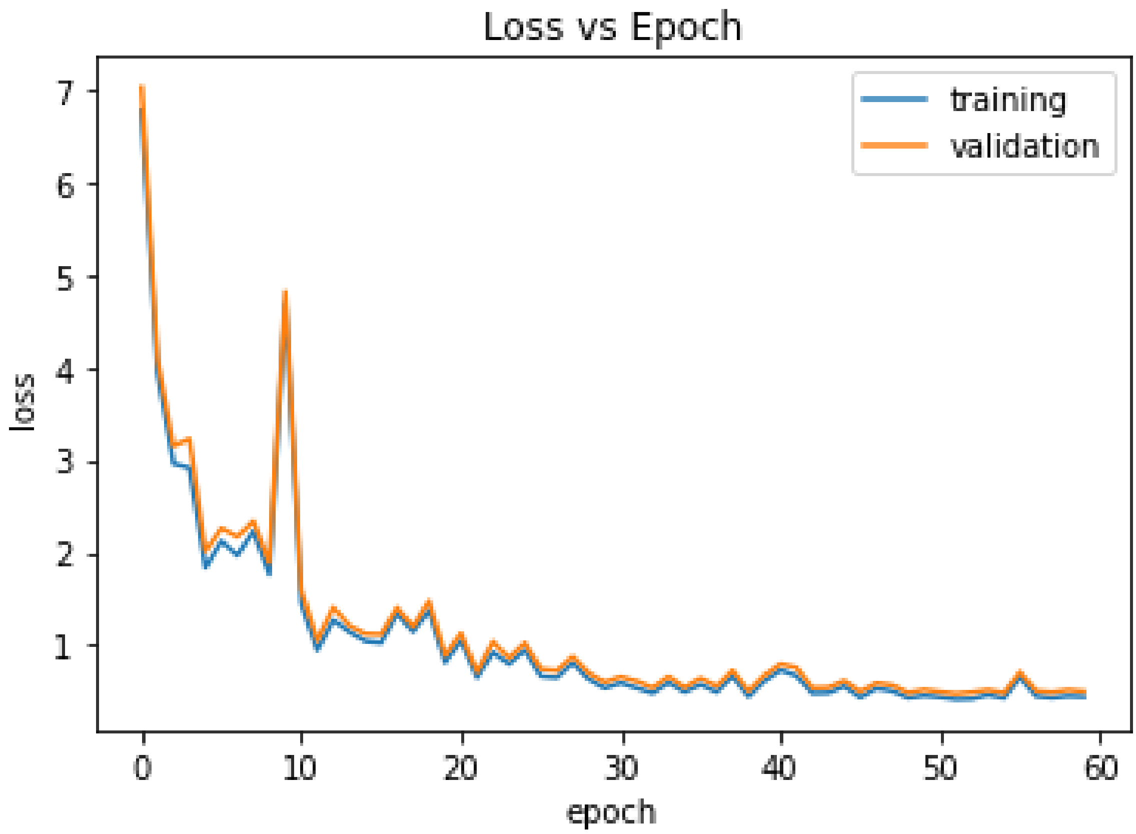
Logistic (sigmoid) activation function:



Tanh activation function:



Identity (linear) activation function:



Scores on validation and training set:

Relu:

```
0.8752222222222222
0.9126862745098039
```

Logistic:

```
0.803
0.8017450980392157
```

Tanh:

```
0.7618888888888888
0.7661568627450981
```

Identity:

```
0.839
0.8499803921568627
```

Results:

- We can see that the loss is the lowest and the score is the highest using the relu activation function.

Analysis and Comparison:

- We can see that relu and identity activation function has a stable validation and training loss curve without much fluctuation whereas logistic and tanh activation functions have validation and training loss curves with a lot of fluctuation.
- This can be because we are using mini batch gradient descent, which cannot capture the trend of the data in logistic and tanh activation functions as well as it can in relu and identity activation functions. This might be why the curves are noisy.
- This can be because when the activation functions converge at a local minimum, they have a different curvature, because of which there are differences in the respective loss curves.
- Relu gives us the smallest loss, and the greatest score. Therefore, we will be using the relu activation function for the next parts.

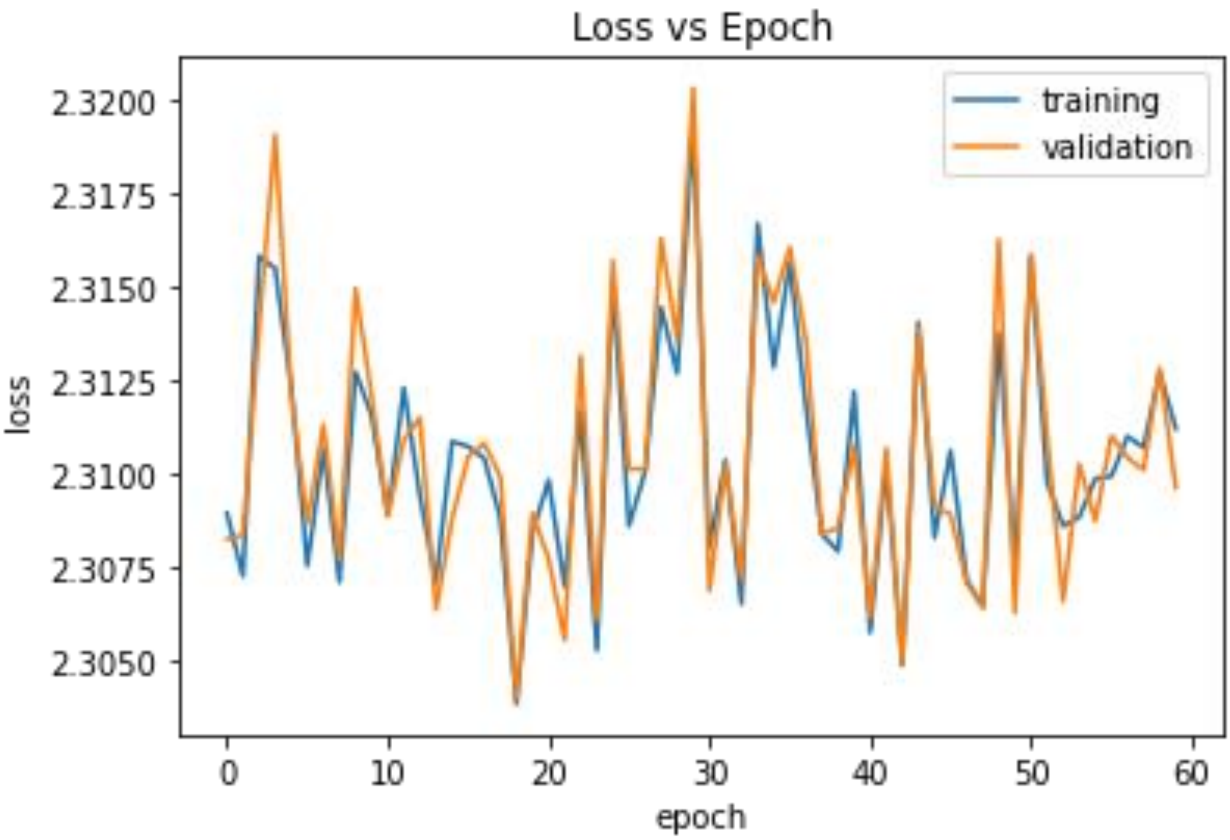
2.

Explanation of Code:

- In the last part, we found that relu activation function is the best activation function, and now using that activation function, we will modify the learning rate and plot the loss curves thrice, with the learning rate being 0.1, 0.01, 0.001.
- Other than the change in the learning rate, the code is the same as used in the previous part.

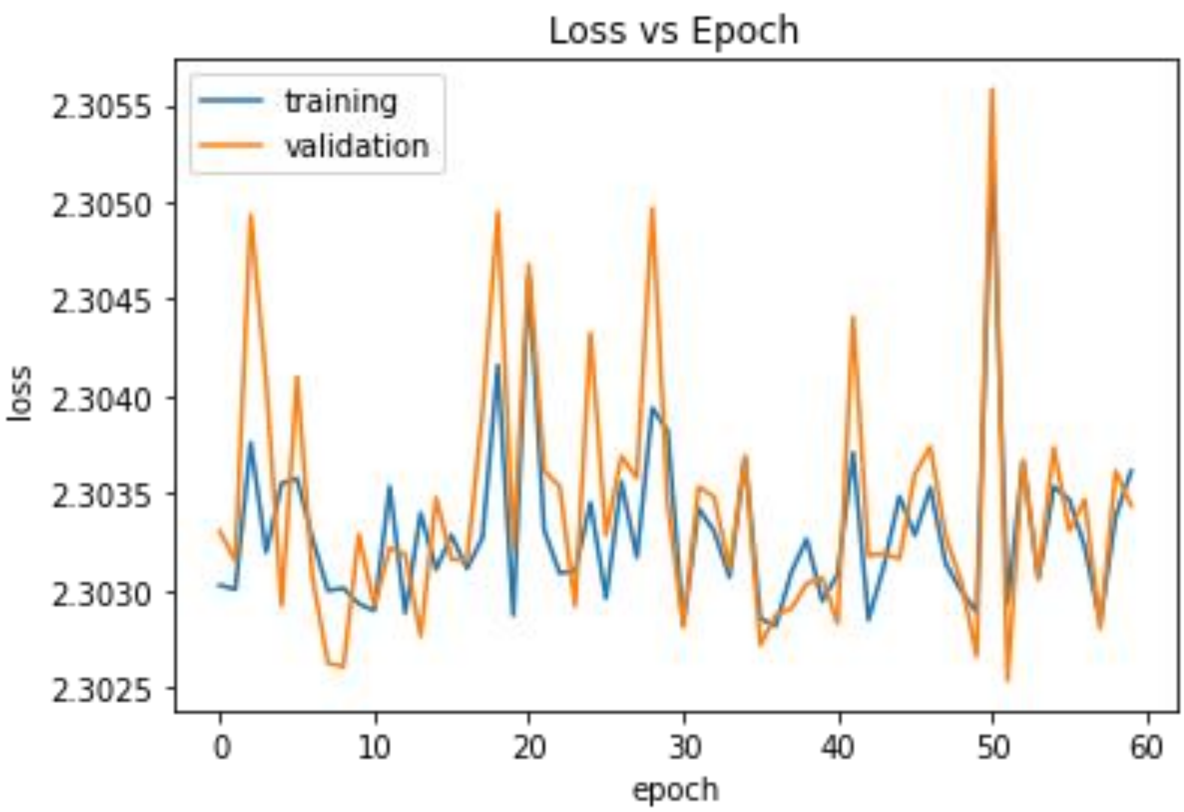
Graphs:

For learning rate 0.1:



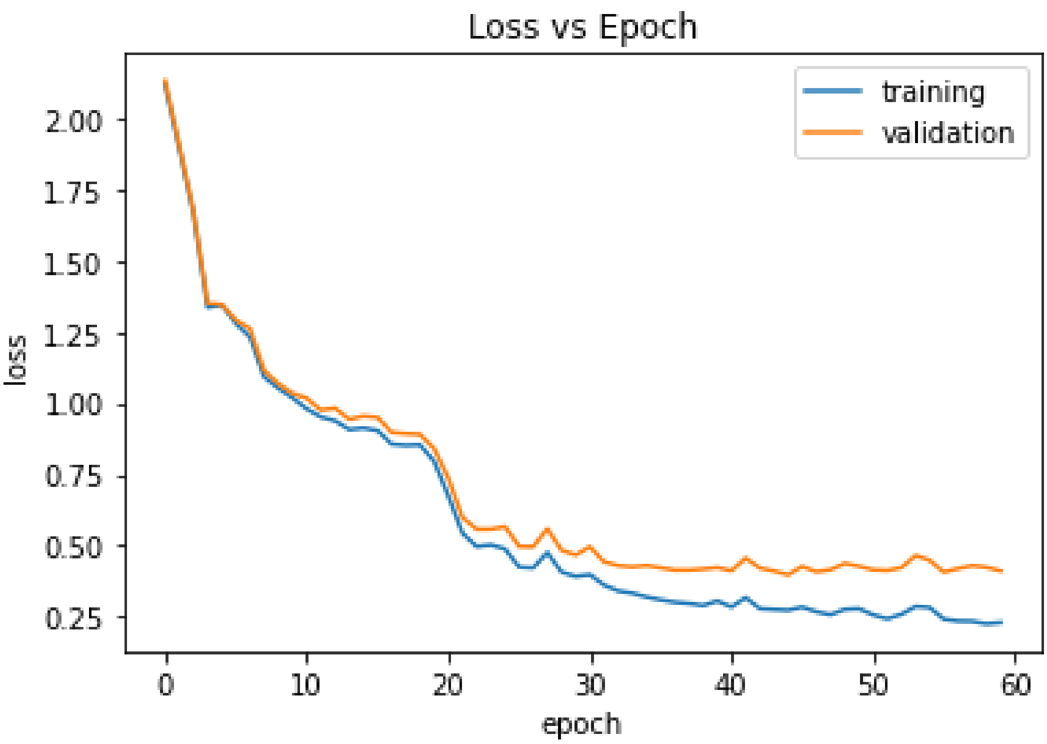

```
0.10077777777777777
0.09986274509803922
```

For learning rate 0.01:




```
0.09766666666666667
0.10041176470588235
```

For learning rate 0.001:



```
0.8826666666666667

0.9184509803921569
```

Results:

- We can see that we get the best score and loss curve with a learning rate of 0.001
- The loss curve for learning rate of 0.001 steadily decreases, whereas the loss curve for 0.1 and 0.01 fluctuates a lot. This is because they reach a local minimum, and because of the curvature, they keep fluctuating, whereas for learning rate 0.001, it has a smaller learning rate, so it gradually and slowly descends down to the local minimum, because of which the loss curve is steady and we get a better score on the validation and training sets.

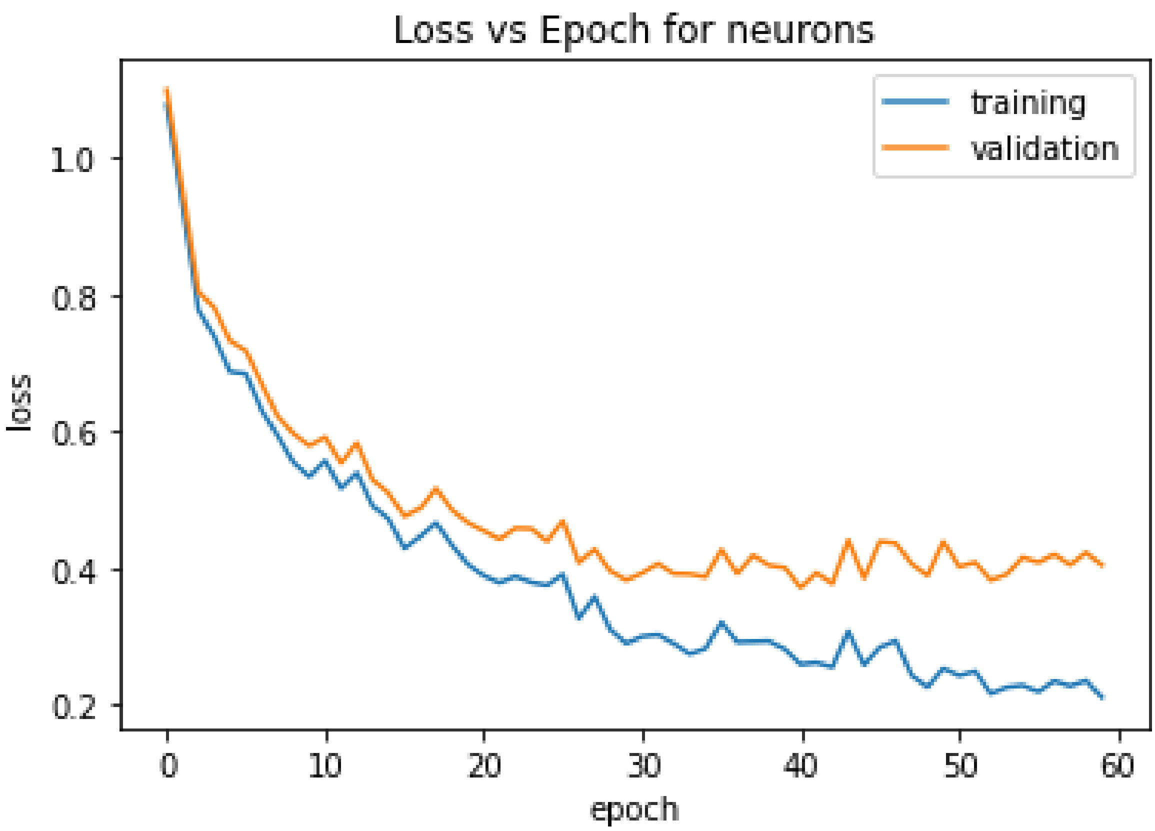
3.

Explanation of Code:

- We have taken the number of neurons as (200,32),(128,16),(64,8),(32,4),(16,2), and we train the models and plot the loss curves and find the score as we did in the previous parts. Other than the change in the number of neurons, the code is the same as used in the previous part

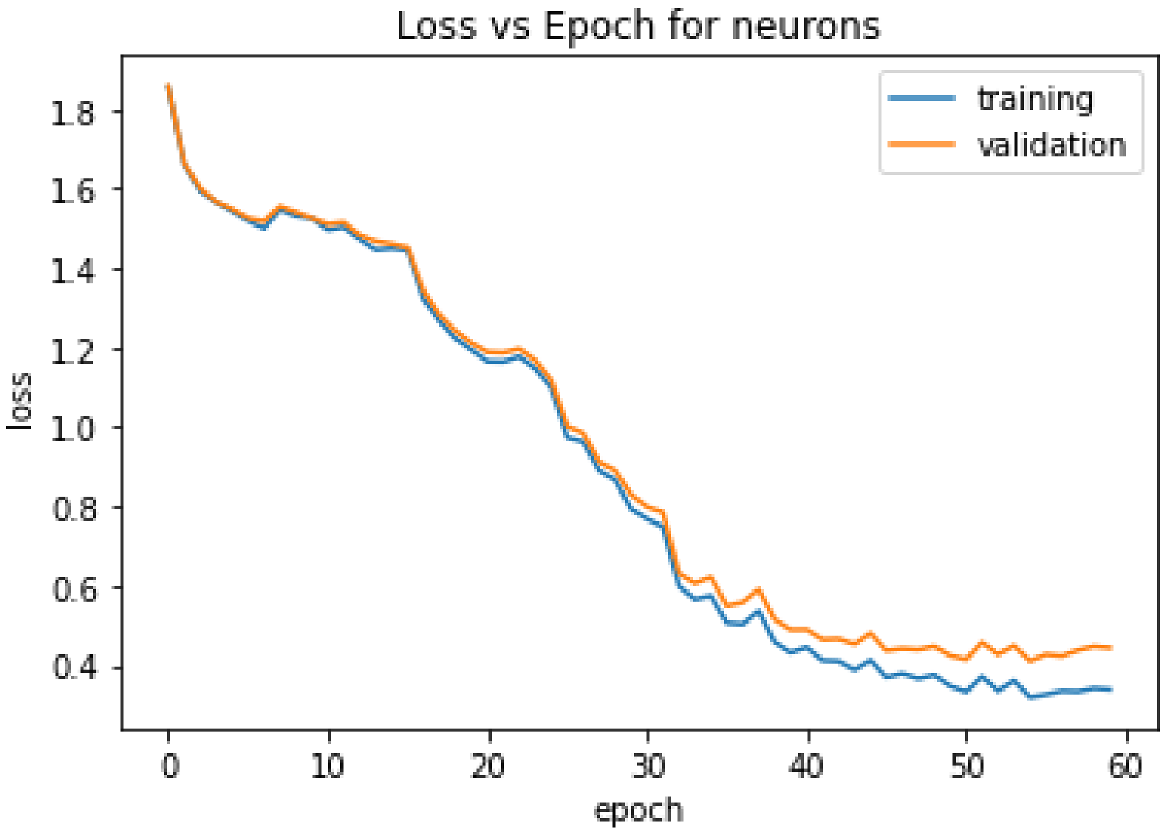
Graphs:

For the number of neurons as (200,32):



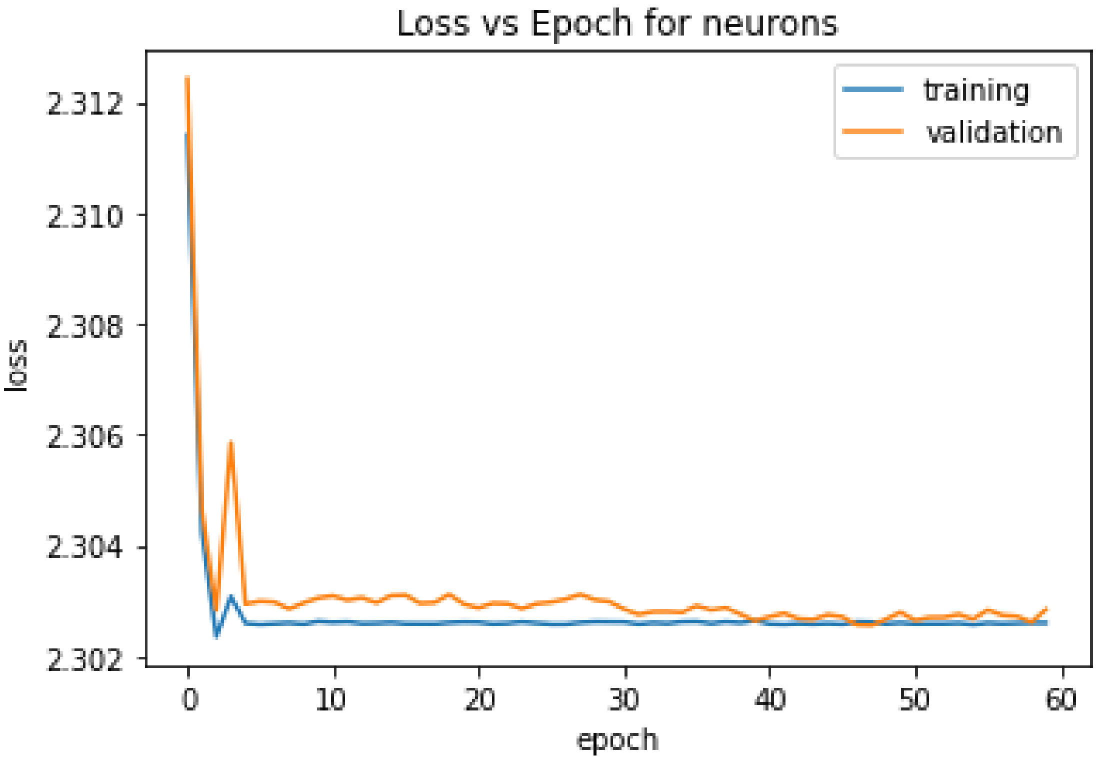
```
0.8827777777777778
0.9258823529411765
```

For the number of neurons as (128,16):



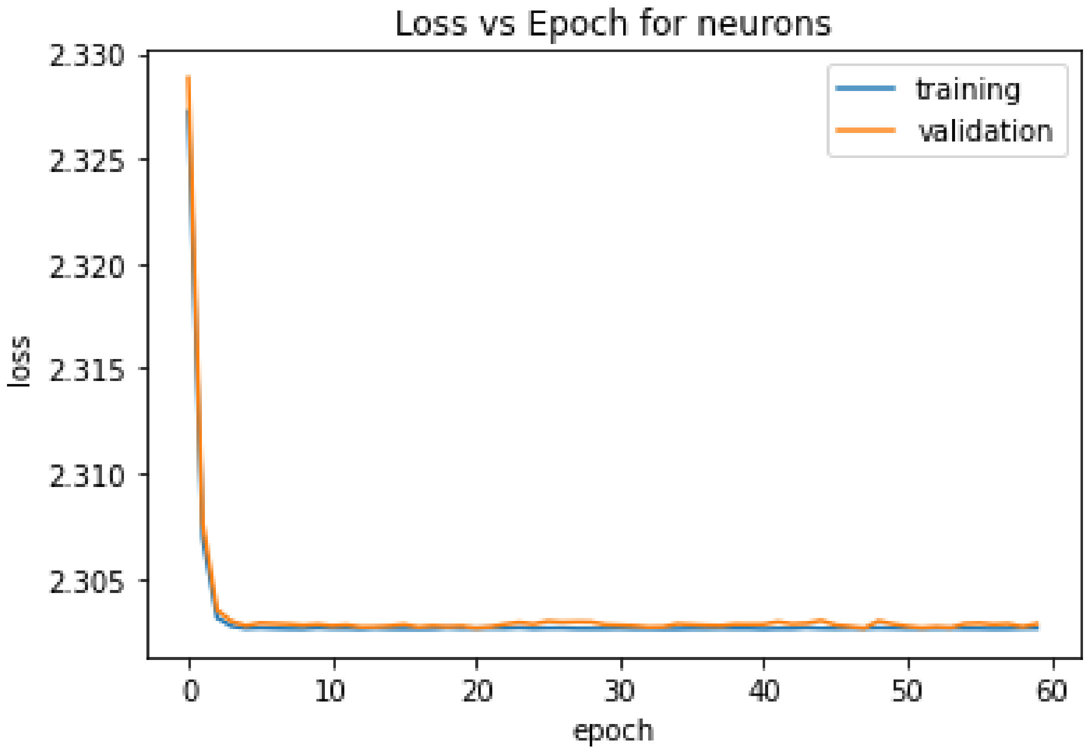

```
0.8517777777777777
0.8694901960784314
```

For the number of neurons as (64,8):



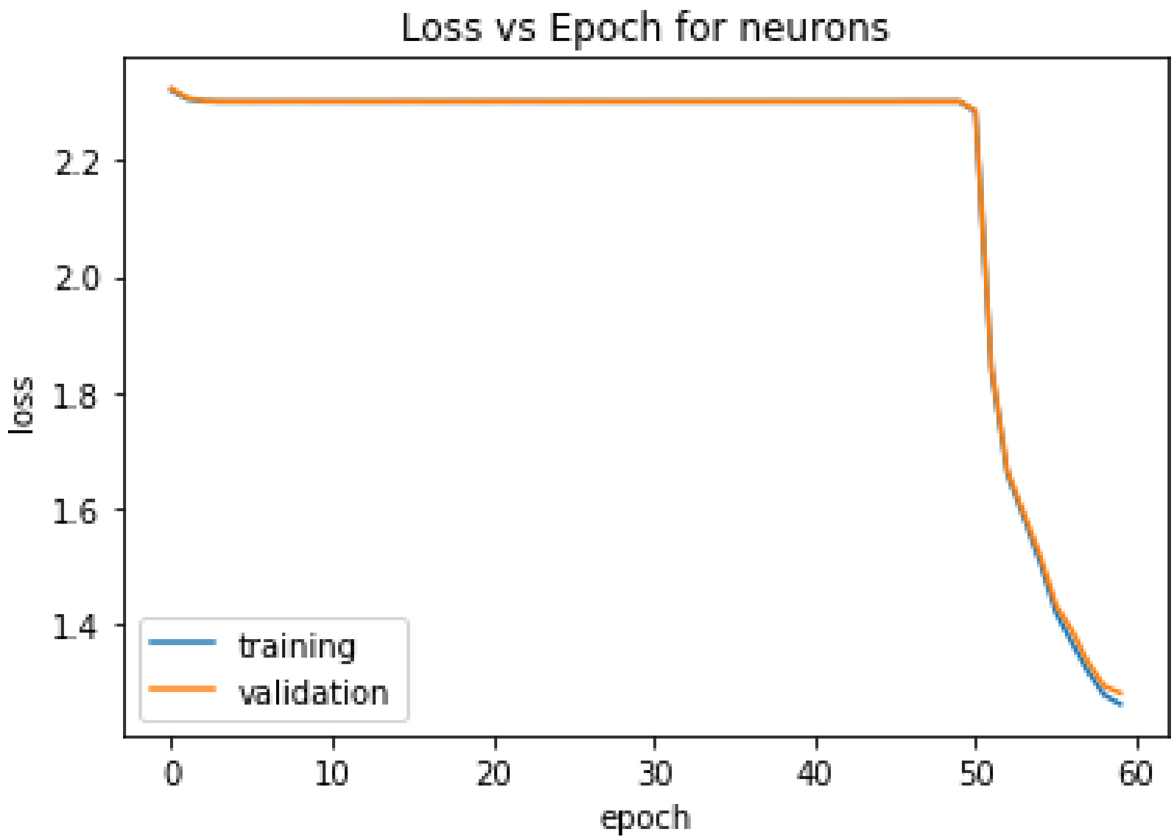

```
0.09566666666666666
0.10076470588235294
```

For the number of neurons as (32,4):



```
0.10077777777777777
0.09986274509803922
```

For the number of neurons as (16,2):




```
0.47033333333333333
0.4792549019607843
```

Results:

- We can see that we get the best loss curve and the best score with the number of neurons set as (200,32).
- We can see that as we decrease the number of neurons the score decreases and the loss increase. This is because as we reduce the number of neurons in the layers, the MLPClassifier model cannot learn as well because it cannot retain as much information due to a reduced number of neurons. Every neuron can learn a feature about the input data. Thus, by reducing the number of neurons, we are reducing the amount of information our model can learn, and the model becomes less flexible.

4.

Explanation of code:

- In this code, we have performed grid search on the appropriate parameters of MLPClassifier.
- We have performed grid search on the tol, alpha and learning_rate_init parameters.
- After that, we have printed the best score, the parameters that gave us this score, and we have also printed the score we got on the training validation and the testing set.

Results:

```
Fitting 3 folds for each of 27 candidates, totalling 81 fits
{'alpha': 0.0001, 'learning_rate_init': 0.0005, 'tol': 0.0001}
0.8583529411764705
MLPClassifier(early_stopping=True, learning_rate_init=0.0005, max_iter=100,
              random_state=0)
0.8922156862745098
0.8672222222222222
0.8562
```

Explanation:

- As we can see, we have found that we get the best score when alpha = 0.0001, learning_rate_init = 0.0005, and tol = 0.0001
- We have gotten the default values for alpha and tol, but we have received a value or learning_rate_init that is smaller than the default value.
- Therefore, a smaller learning rate gives us a better accuracy because otherwise on a larger learning rate, it begins overshooting and the score we would get would lessen as the model would not be able to reach the minimum.
- Alpha denotes the strength of the L2 regularization term. Increasing alpha fixes high variance(overfitting), whereas decreasing alpha would fix high bias(underfitting). Therefore, we can see that the value of alpha is neither too high nor too low, which suggests that our model isn't particularly overfitting or underfitting. Therefore, as the model isn't overfitting or underfitting, and we are getting a good accuracy on the testing set (0.8562), we can see that the value of alpha shouldn't be too large or too small.
- Tol denotes tolerance for the optimization. When the models score doesn't improve by tol for a certain number of iterations, it is assumed that the model has reached convergence and the training stops. It basically tells the model to stop training once we are close enough. We can see that we have received a value of 0.0001 as the best parameter value for tol. This suggests that for this particular dataset, having a high value of tolerance isn't good, as then the model will stop training early, without reaching the best accuracy. Also, we didn't receive the tol as 0.00001 either, because the function might've stopped at two different local minima because of which we got a better value at the first one.