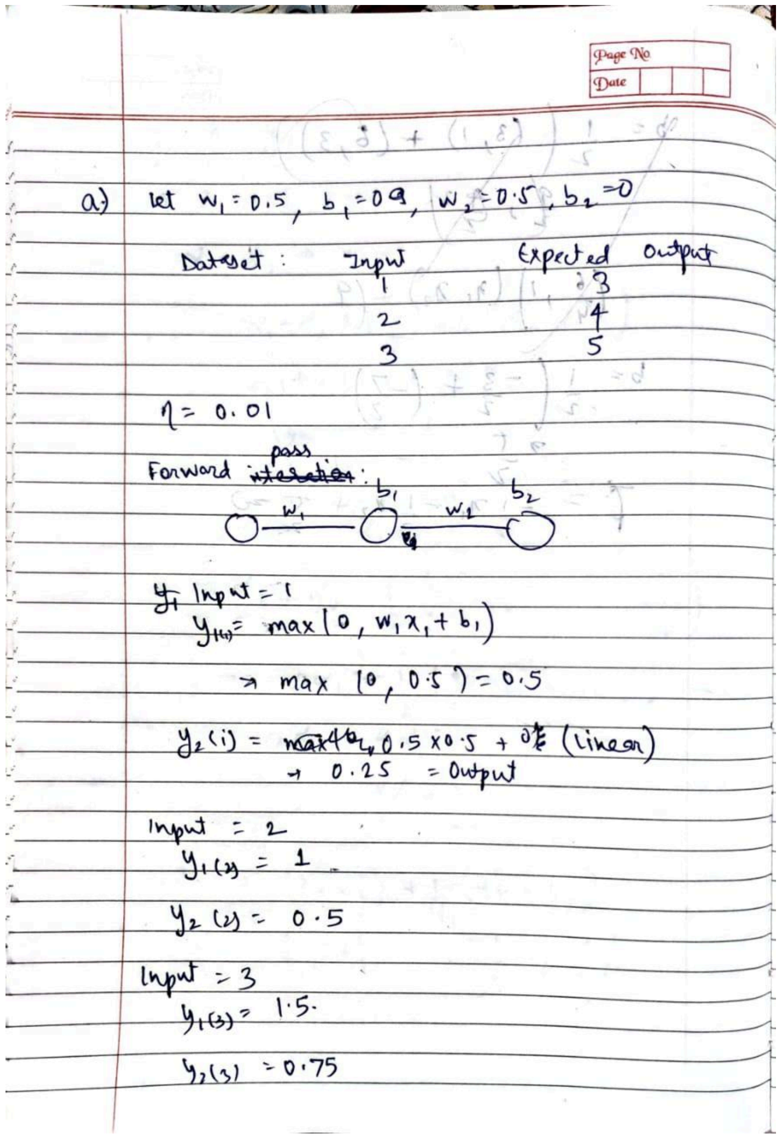
CSE343: Machine Learning
Assignment-3

REPORT

Anushka Srivastava (2022086)

SECTION A

a. Answer 1



Page No

Date

a) let $w_1 = 0.5$, $b_1 = 0$, $w_2 = 0.5$, $b_2 = 0$

Dataset :

	Input	Expected Output
	1	3
	2	4
	3	5

$\eta = 0.01$

Forward ~~iteration~~ pass :

$$\begin{tikzcd} \bigcirc \arrow[r, dash, "w_1"] & \overset{b_1}{\bigcirc} \arrow[r, dash, "w_2"] & \overset{b_2}{\bigcirc} \end{tikzcd}$$

y_1 Input = 1

$y_{1(1)} = \max(0, w_1 x_1 + b_1)$

$\rightarrow \max(0, 0.5) = 0.5$

$y_2(1) =$ ~~max~~ $0.5 \times 0.5 + 0$ (linear)

$\rightarrow 0.25$ = Output

Input = 2

$y_{1(2)} = 1$

$y_2(2) = 0.5$

Input = 3

$y_{1(3)} = 1.5$

$y_{2(3)} = 0.75$

Page No.
Date

Predicted outputs = [0.25, 0.5, 0.75]

loss = MSE = $\frac{1}{3}\sum(y_i - \hat{y}_i)^2 = \phi$

$\Rightarrow \frac{1}{3}\left[(3-0.25)^2 + (\overset{4}{\cancel{0.5}} - 0.5)^2 + (5-0.75)^2\right]$

$\Rightarrow 12.625$

~~$\phi' = \frac{2}{3}\sum(y_i - \hat{y})$~~

Backward pass

~~$\Delta w_2 = \frac{2}{3}\sum(y_i - \hat{y})\cdot$~~

$\Delta w_2 = \eta\,\delta_2 y_i$

$= 0.01 \times \frac{2}{3}\sum(y_i - \hat{y})\cdot y_i$

$\Rightarrow 0.01 \times \frac{2}{3}\left[(3-0.25)\times 0.5 + (4-0.5)\times 1\right.$

$\left.+ (5-0.75)\times 1.5\right)$

$\Rightarrow -0.075$

$w_2 = w_2 - \Delta w_2 = 0.5 + 0.075$

$\Rightarrow$ ~~0.0428~~ 0.575

$\Delta b_2 = 0.01 \times \frac{2}{3}\sum(y_i - \hat{y}) =$ ~~0.10~~ $0.01 \times \frac{2}{3}((3-0.25)$

$+ (4-0.5) + (5-0.75)) = -0.07$

$b_2 = b_2 + \Delta b_2 = +0.07$

$\Delta w_1 = 0.01 \times \frac{2}{3} \Sigma (y_i - \hat{y}_i) \cdot w_2 \cdot 1 \cdot x_i$

$\rightarrow 0.01 \times \frac{2}{3} \left((-2.75) \times 0.5 \times 1 + (-3.5) \times 0.5 \times 2 + (-4.25) \times 0.5 \times 3 \right)$

$\Rightarrow -0.075$

$w_1 = w_1 - \Delta w_1 = 0.575$

$\Delta b_1 = 0.01 \times \frac{2}{3} \Sigma (y_i - \hat{y}_i) \cdot w_2 \cdot 1$

$\rightarrow 0.01 \times \frac{2}{3} \left(0.5 \times (-2.75) + 0.5 \times (-3.5) + 0.5 \times (-4.25) \right)$

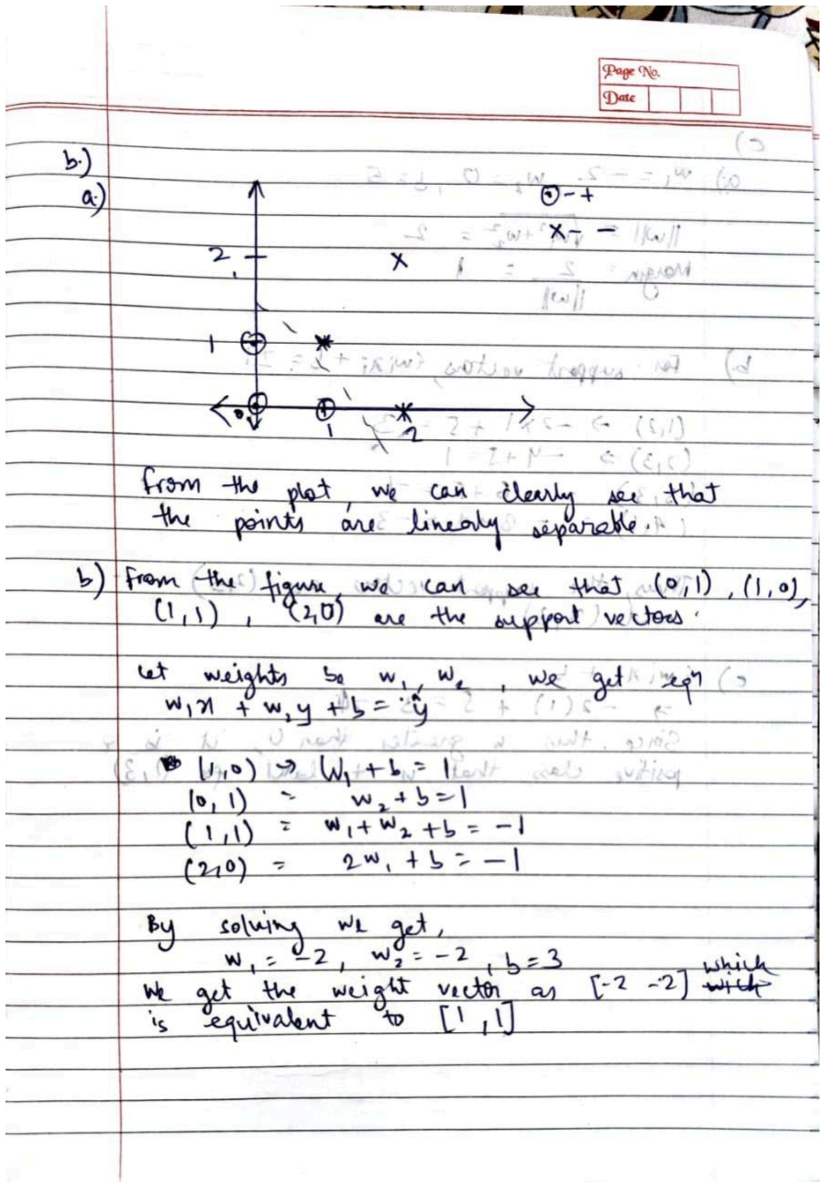
$\rightarrow -0.035$

$b_1 = b_1 - \Delta b_1 = 0.035$

Weights and biases update after 1 iteration

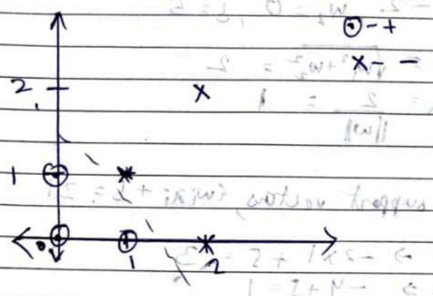
	Initial	Updated
w_1	0.5	0.575
b_1	0	0.035
w_2	0.5	0.575
b_2	0	0.07

b. Answer 2



Page No.
Date

b.)
a.)



from the plot, we can clearly see that the points are linearly separable.

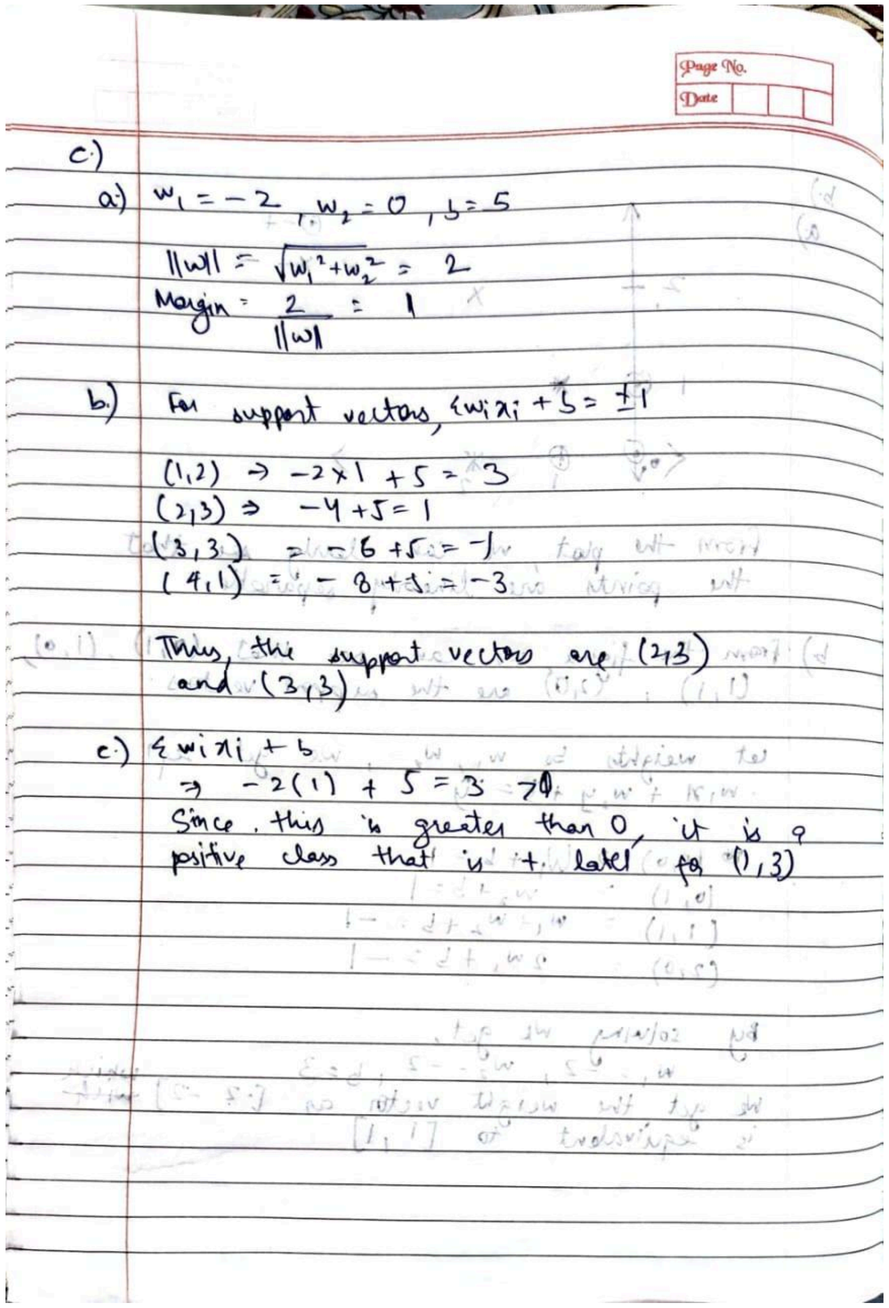
b) From the figure, we can see that (0,1), (1,0), (1,1), (2,0) are the support vectors.

Let weights be w_1, w_2, we get eqⁿ
$w_1 x + w_2 y + b = y$

$(1,0) \rightarrow W_1 + b = 1$
$(0,1) = w_2 + b = 1$
$(1,1) = w_1 + w_2 + b = -1$
$(2,0) = 2w_1 + b = -1$

By solving we get,
$w_1 = -2$, $w_2 = -2$, $b = 3$
We get the weight vector as $[-2\ \ -2]$ ~~with~~ which is equivalent to $[1, 1]$

c. Answer 3



Page No.

Date

c.)

a.) $w_1 = -2$, $w_2 = 0$, $b = 5$

$\|w\| = \sqrt{w_1^2 + w_2^2} = 2$

Margin $= \frac{2}{\|w\|} = 1$

b.) For support vectors, $\Sigma w_i x_i + b = \pm 1$

$(1,2) \rightarrow -2 \times 1 + 5 = 3$

$(2,3) \Rightarrow -4 + 5 = 1$

$(3,3) \Rightarrow -6 + 5 = -1$

$(4,1) \Rightarrow -8 + 5 = -3$

Thus, the support vectors are $(2,3)$ and $(3,3)$

c.) $\Sigma w_i x_i + b$

$\rightarrow -2(1) + 5 = 3 > 0$

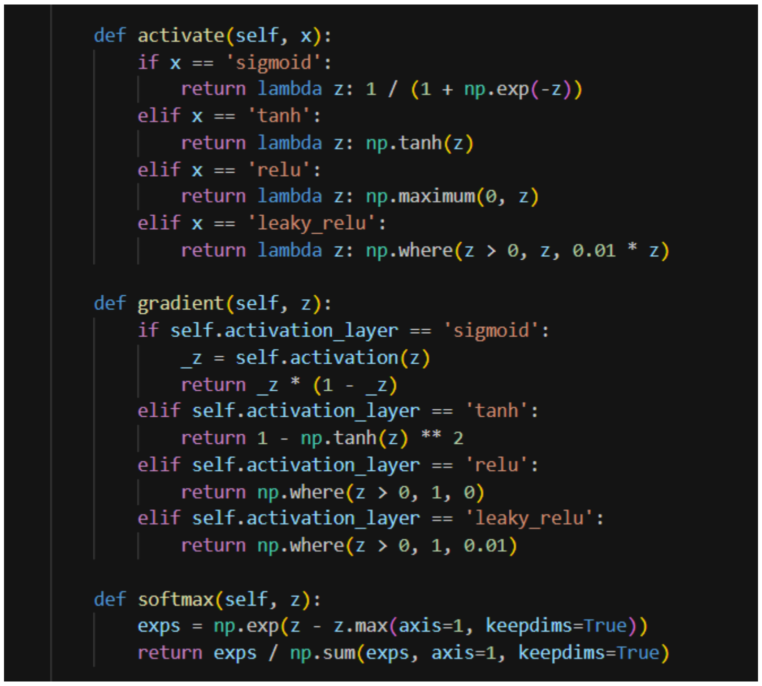
Since, this is greater than 0, it is a positive class that is +1 label for $(1,3)$

SECTION B (BONUS)

1. MNIST Dataset is loaded by creating a custom dataloader using classes. A Neural Network is implemented from scratch by creating a class and defining all the necessary attributes and methods. The dataset consists of handwritten

digits.

2. All the activation functions are implemented as lambda functions within the Neural network class itself, and a gradient function is defined to calculate their respective gradients. Softmax function is defined within the class and is only used in the last layer.



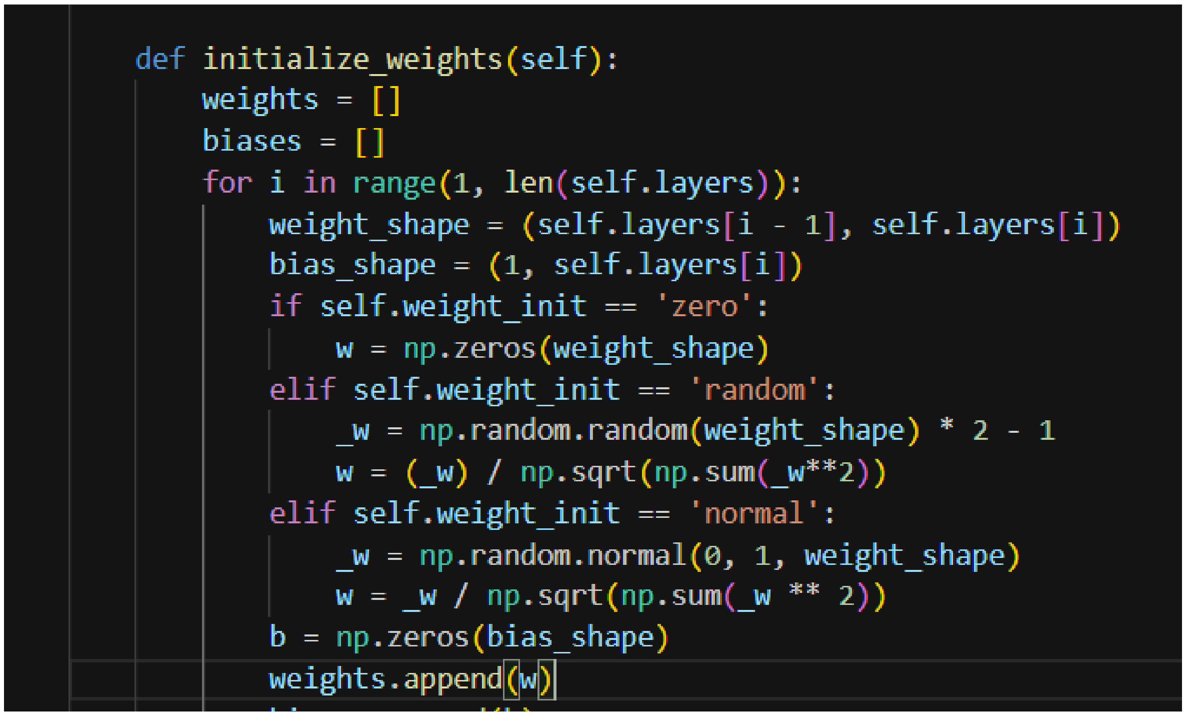
```
def activate(self, x):
    if x == 'sigmoid':
        return lambda z: 1 / (1 + np.exp(-z))
    elif x == 'tanh':
        return lambda z: np.tanh(z)
    elif x == 'relu':
        return lambda z: np.maximum(0, z)
    elif x == 'leaky_relu':
        return lambda z: np.where(z > 0, z, 0.01 * z)

def gradient(self, z):
    if self.activation_layer == 'sigmoid':
        _z = self.activation(z)
        return _z * (1 - _z)
    elif self.activation_layer == 'tanh':
        return 1 - np.tanh(z) ** 2
    elif self.activation_layer == 'relu':
        return np.where(z > 0, 1, 0)
    elif self.activation_layer == 'leaky_relu':
        return np.where(z > 0, 1, 0.01)

def softmax(self, z):
    exps = np.exp(z - z.max(axis=1, keepdims=True))
    return exps / np.sum(exps, axis=1, keepdims=True)
```

3. An initialize_weights() method is defined within the class, which initializes weights according to the passed attribute. The shape of each weight is (previous_layer_neuron, current_layer_neuron), and the shape of bias is (1, num_classes). The Numpy library is used to initialize and store the weights and biases.
   Scaling factors:
   - Zero_init = No scaling
   - Random_init = First range is defined to fit it in [-1, 1], and then it is normalized by dividing it by its norm to control variance.
   - Normal_init = It is assigned a value from a Gaussian distribution with mean 0 and standard deviation 1 and it is normalized by dividing it by its norm to control variance.

   All the biases are initialized with zeroes.

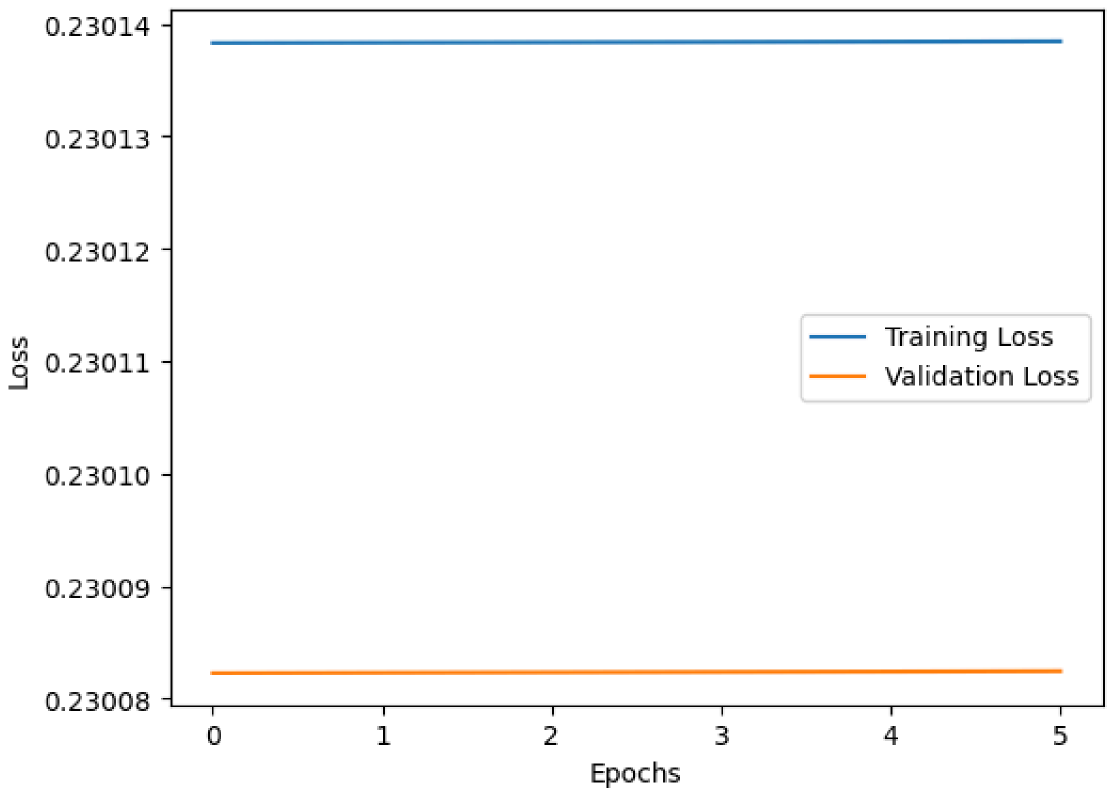


```
def initialize_weights(self):
    weights = []
    biases = []
    for i in range(1, len(self.layers)):
        weight_shape = (self.layers[i - 1], self.layers[i])
        bias_shape = (1, self.layers[i])
        if self.weight_init == 'zero':
            w = np.zeros(weight_shape)
        elif self.weight_init == 'random':
            _w = np.random.random(weight_shape) * 2 - 1
            w = (_w) / np.sqrt(np.sum(_w**2))
        elif self.weight_init == 'normal':
            _w = np.random.normal(0, 1, weight_shape)
            w = _w / np.sqrt(np.sum(_w ** 2))
        b = np.zeros(bias_shape)
        weights.append(w)
```

4. **Preprocessing:**
   All the pixels are divided by 255 to normalize them to bring them to a range between 0 and 1. All the labels are one-hot encoded. The images are flattened from 28 x 28 to 784 to improve training.

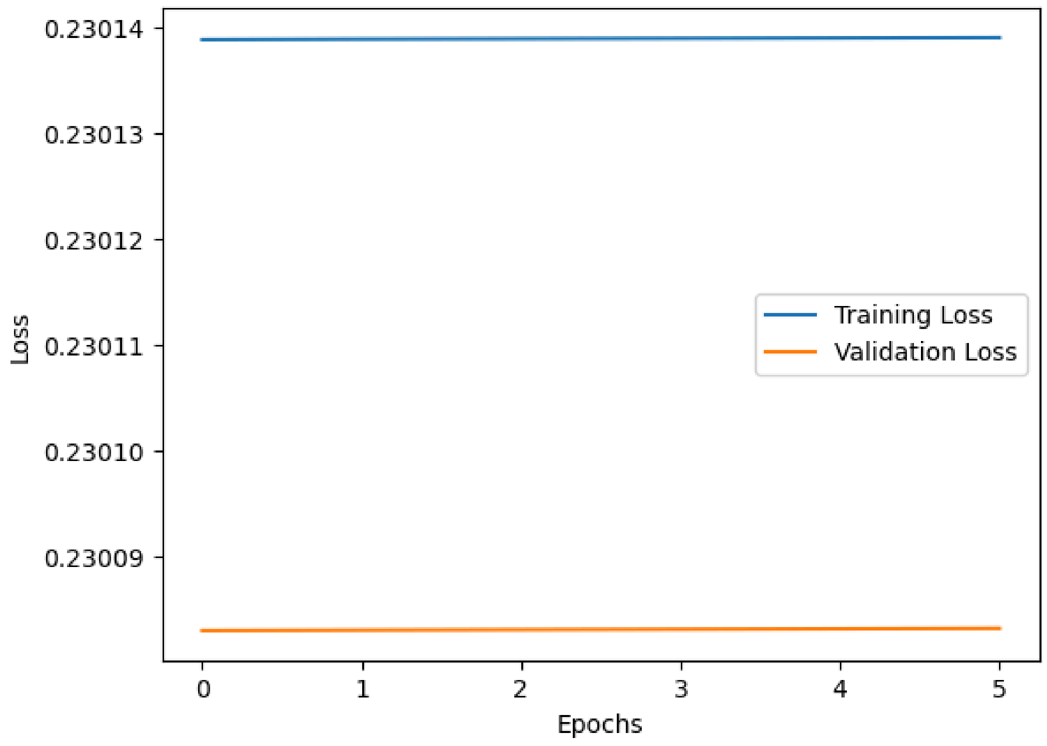
   Model 1:

Activation Layer: Sigmoid, Weight_init = Zero_init



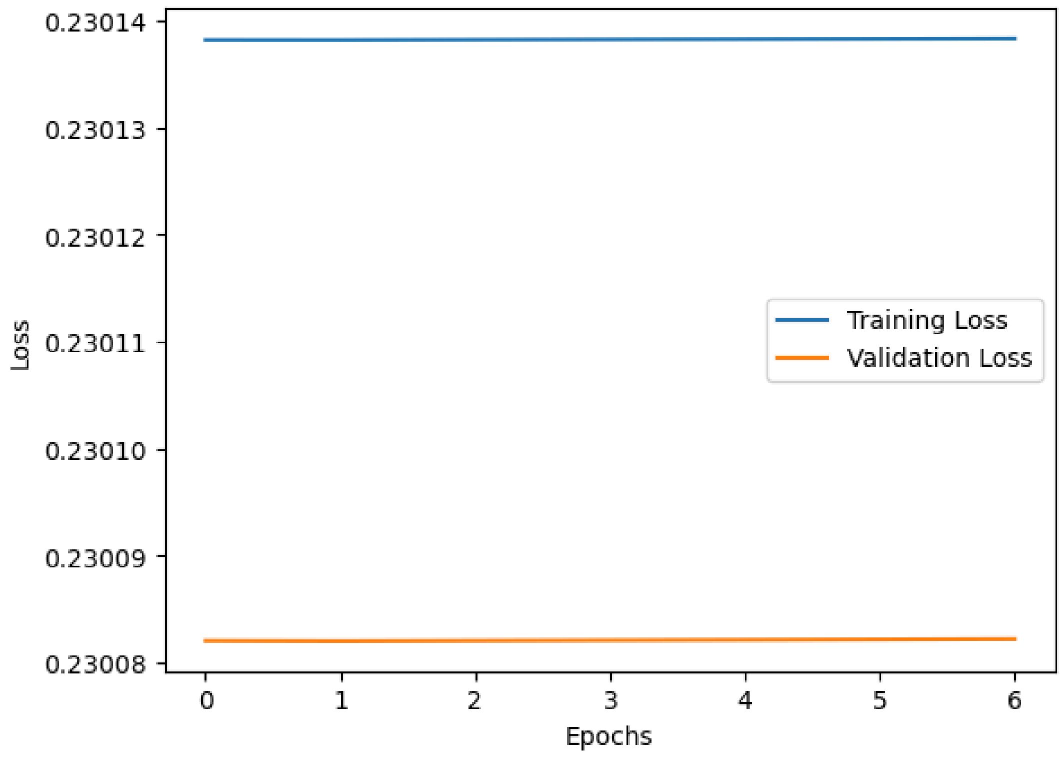
Accuracy: 82.23%
Early stopping at epoch 5.

Model 2:
Activation Layer: Sigmoid, Weight_init = Random_init



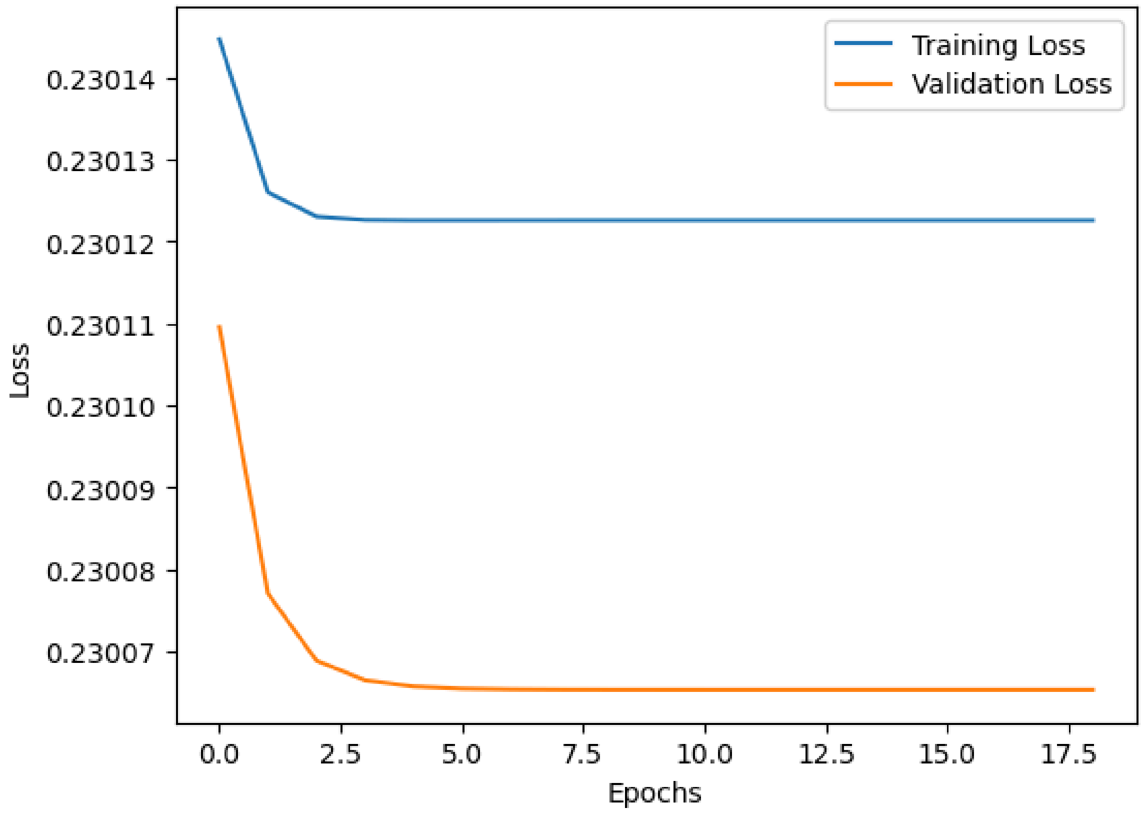
Accuracy: 82.23%
Early stopping at epoch 5.

Model 3:
Activation Layer: Sigmoid, Weight_init = Normal_init



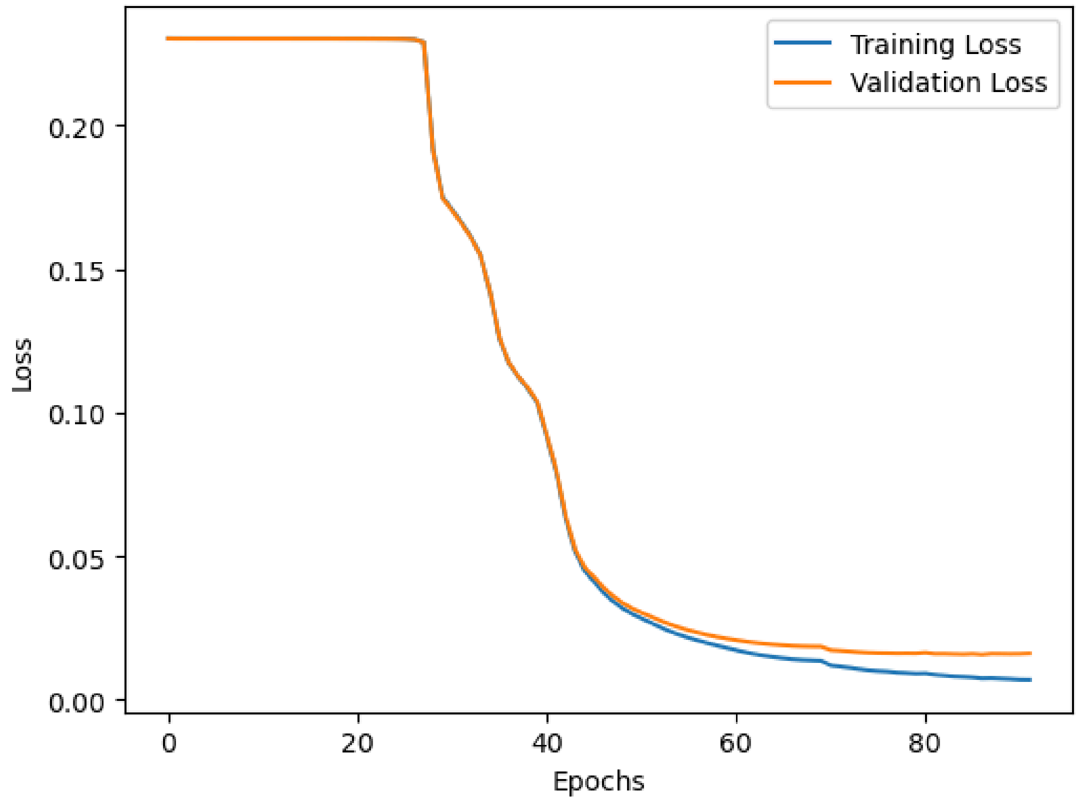
Accuracy: 82.23%
Early stopping at epoch 6.

Model 4:
Activation Layer: Tanh, Weight_init = Zero_init



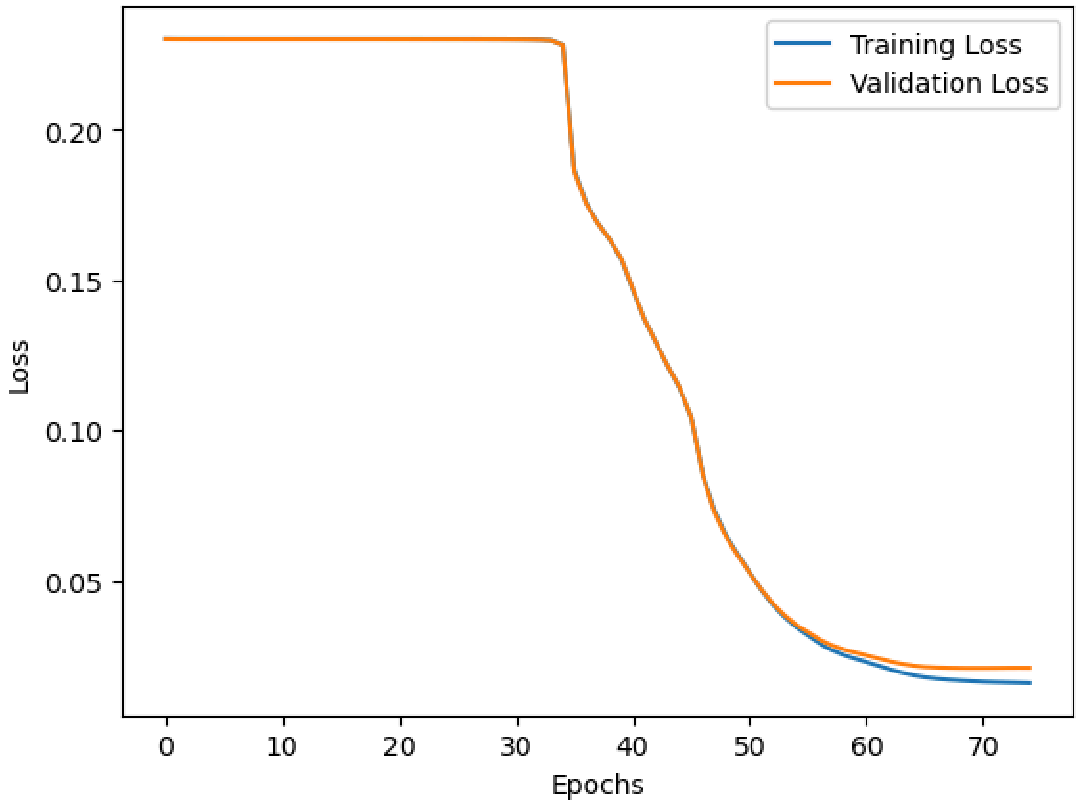
Accuracy: 82.23%
Early stopping at epoch 18.

Model 5:
Activation Layer: Tanh, Weight_init = Random_init



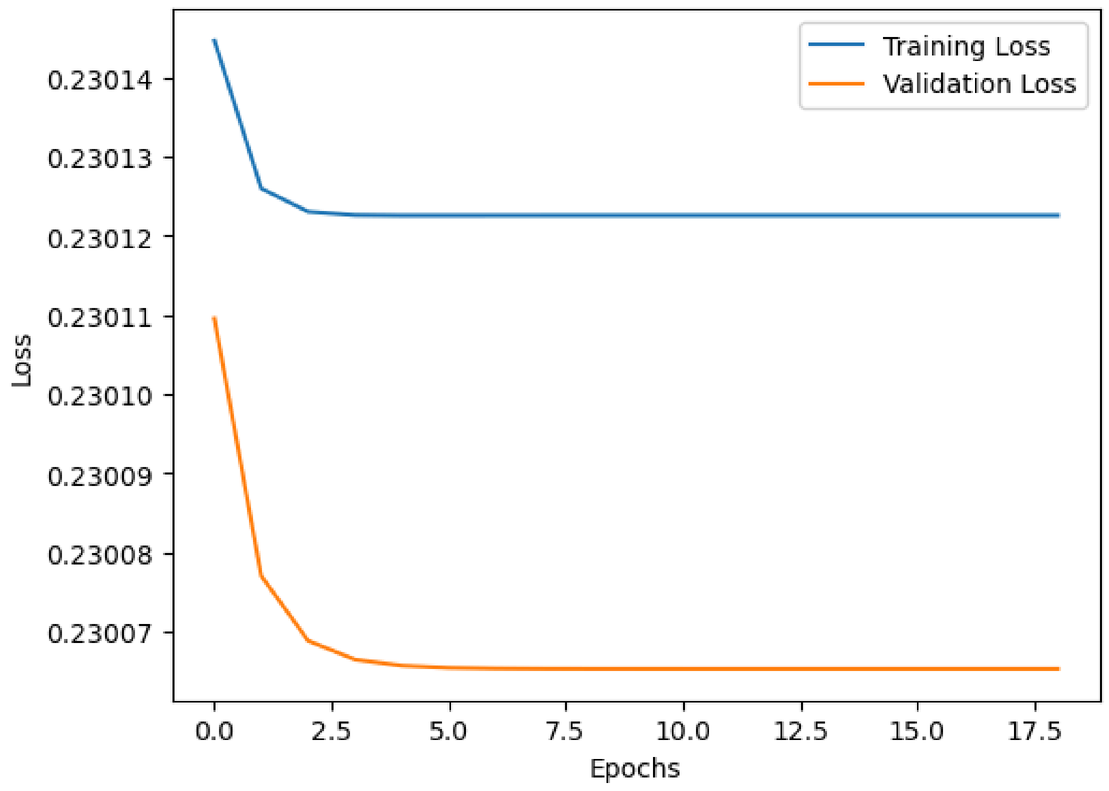
Accuracy: 99.10%
Early stopping at epoch 91.

Model 6:
Activation Layer: Tanh, Weight_init = Normal_init



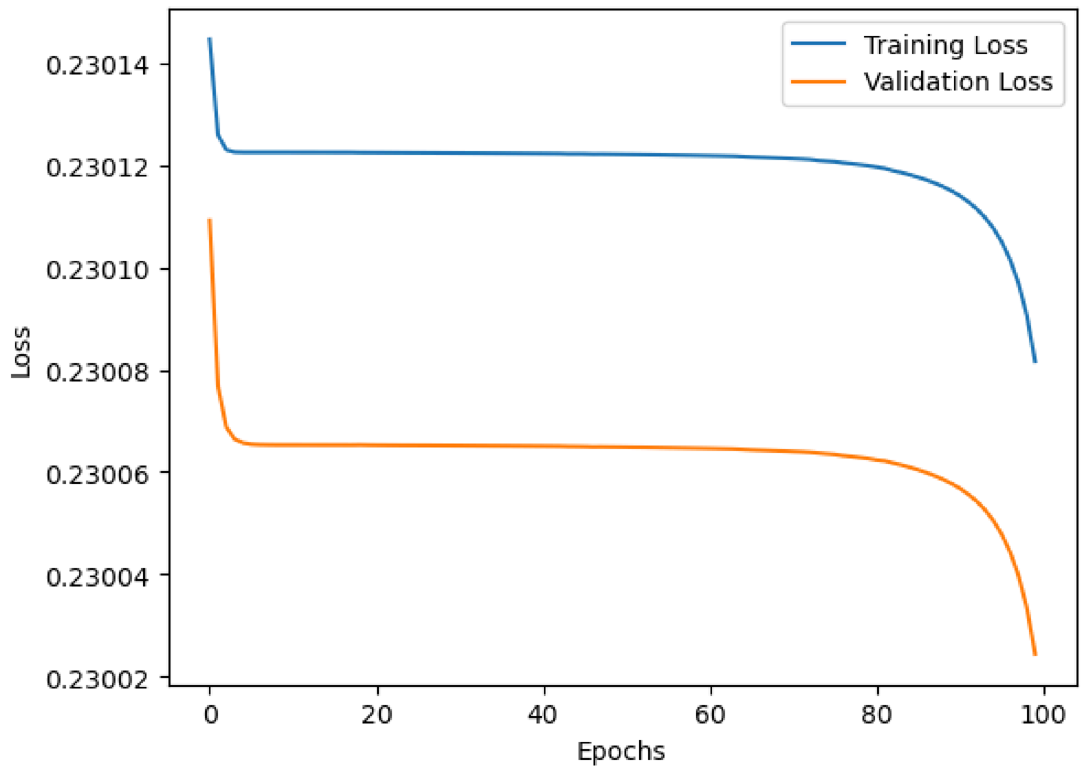
Accuracy: 98.8%
Early stopping at epoch 74.

Model 7:
Activation Layer: Relu, Weight_init = Zero_init



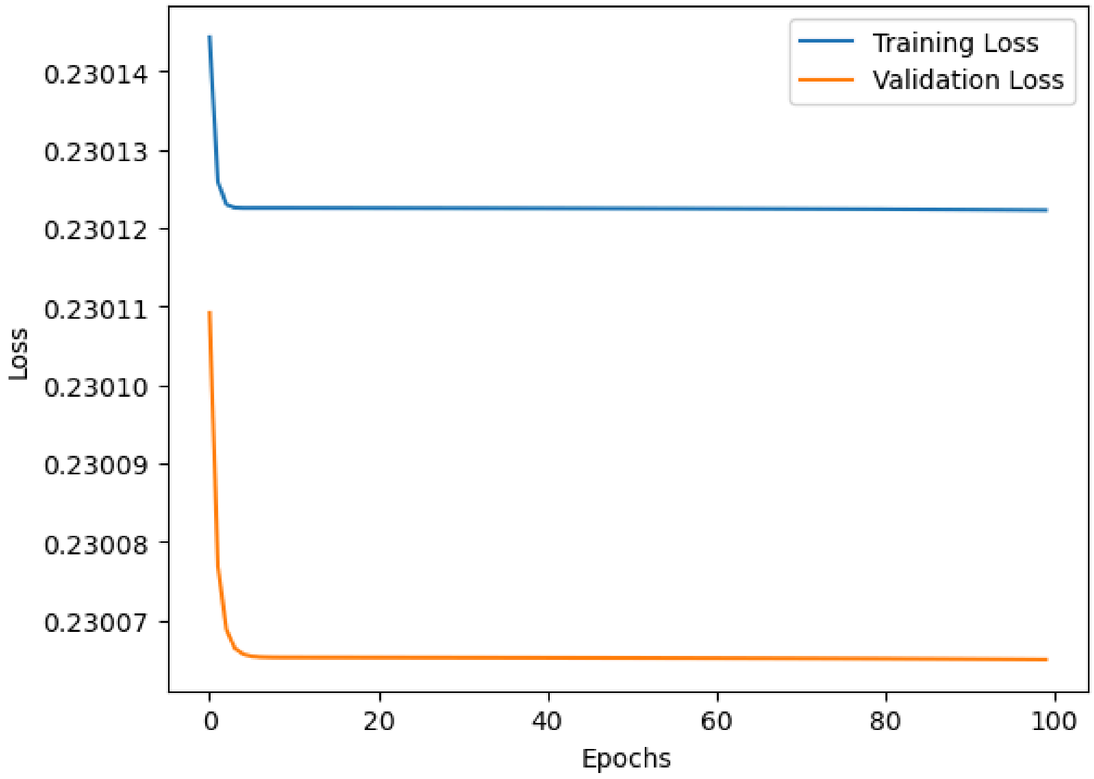
Accuracy: 82.23%
Early stopping at epoch 18.

Model 8:
Activation Layer: Relu, Weight_init = Random_init



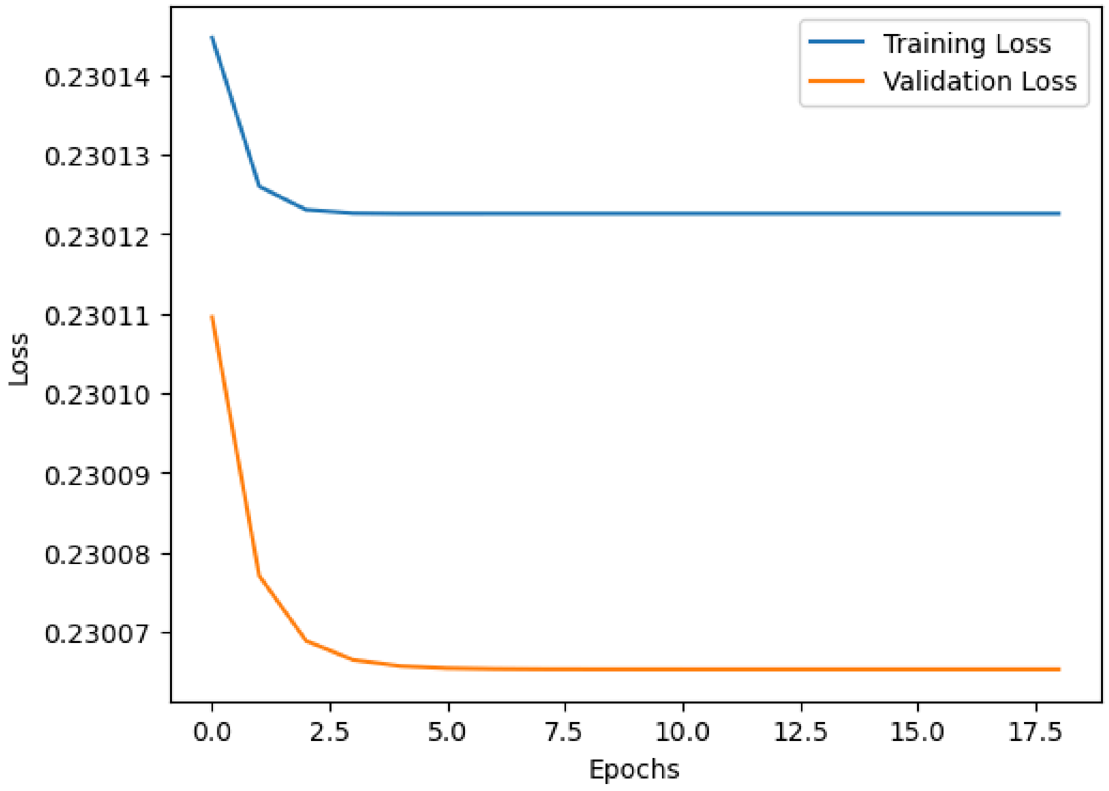
Accuracy: 82.23%
No early stopping.

Model 9:
Activation Layer: Tanh, Weight_init = Normal_init



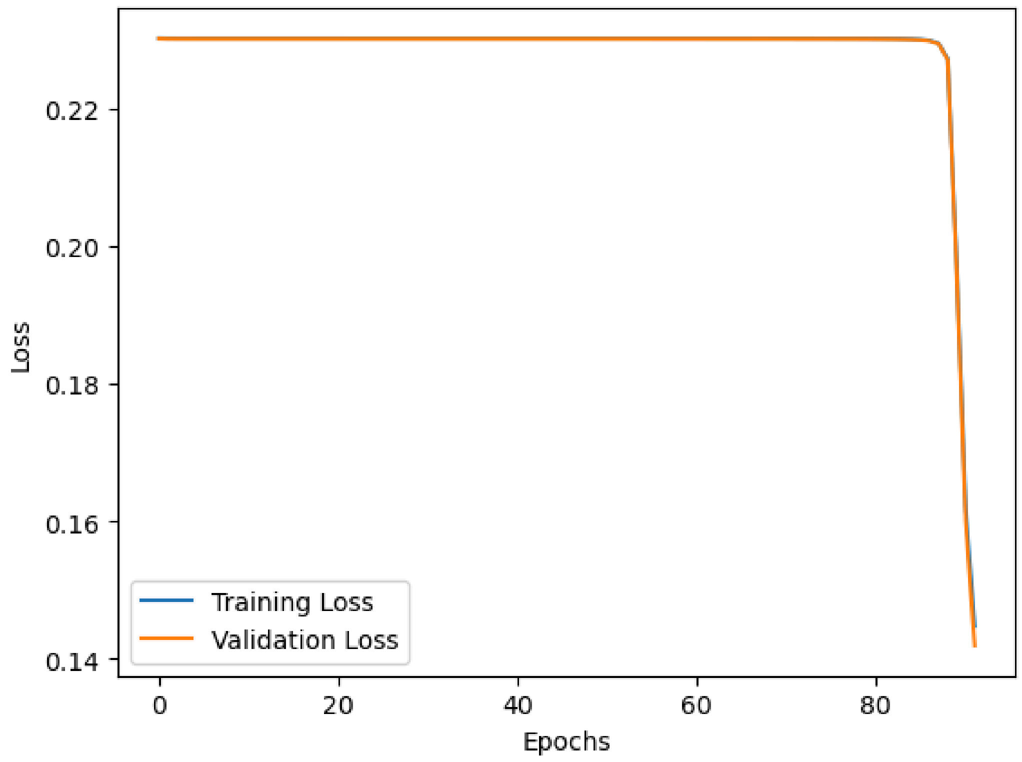
Accuracy: 82.23%
No early stopping.

Model 10:
Activation Layer: Leaky_relu, Weight_init = Zero_init



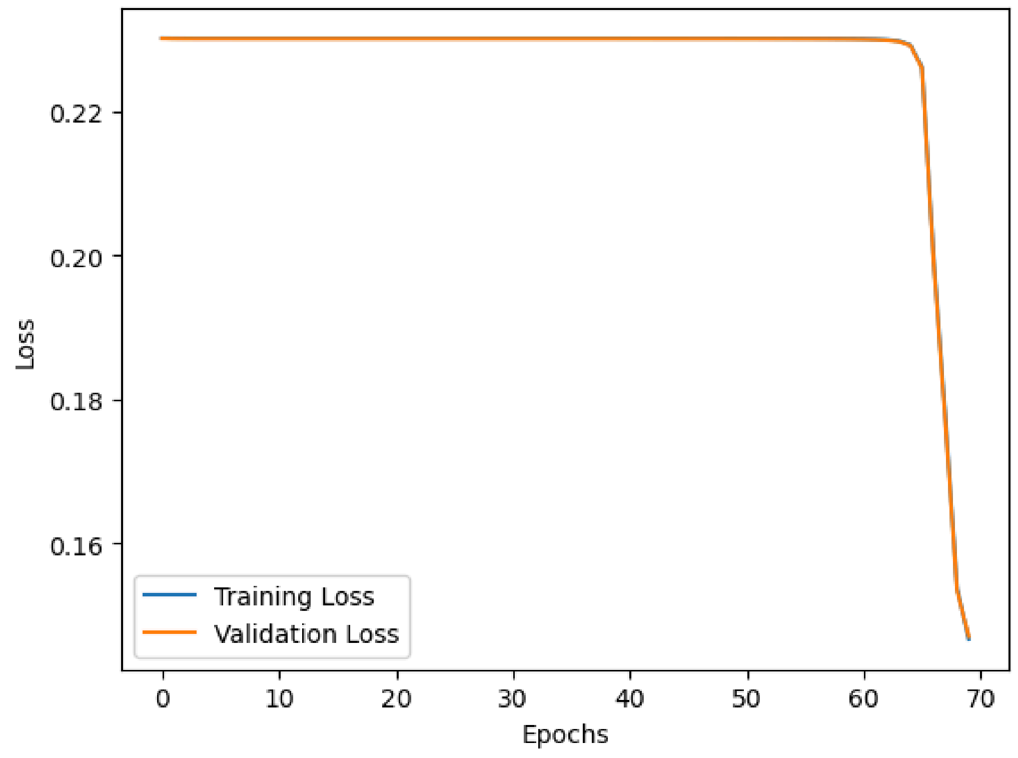
Accuracy: 82.23%
Early stopping at epoch 18.

Model 11:
Activation Layer: Leaky_relu, Weight_init = Random_init



Accuracy: 81.88%
Early stopping at epoch 96.

Model 3:
Activation Layer: Leaky_relu, Weight_init = Normal_init



Accuracy: 81.88%
Early stopping at epoch 74.

Parameters:
N = 6
Hidden layers = 4
Hidden layers sizes = [256, 128, 64, 32]
Input layer size = 784
Output layer size = 10
Batch size = 64
Learning rate = 0.01
Epochs = 100

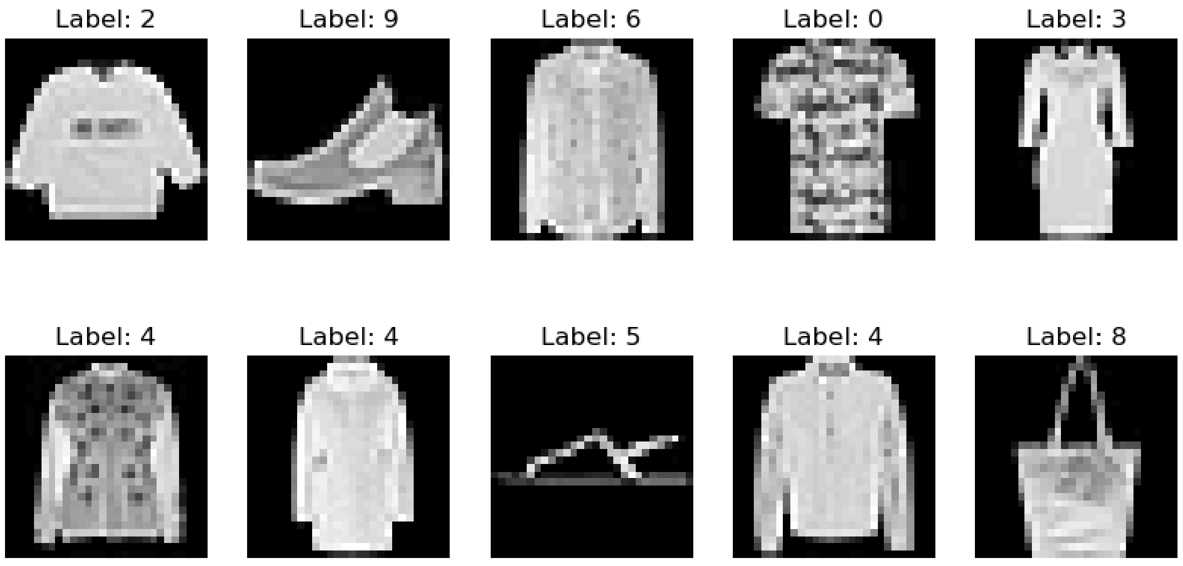
Findings:

- We notice that the model with **tanh** activation layer with **random_init** weight initialization function is the best-performing model with 99.1% accuracy, followed by the model with **tanh** activation layer and **normal_init** weight initialization function with 98.8% accuracy.

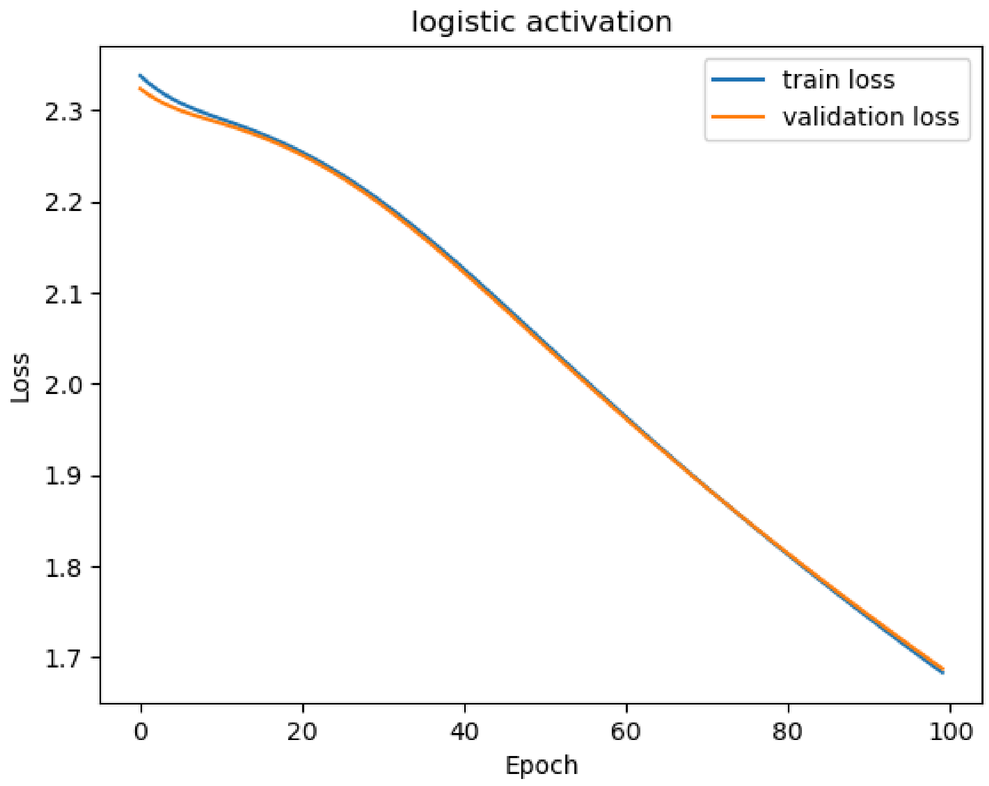
- The other models have comparable performances, with 2 out of 10 models having 81.88% accuracy and the other models having 82.23% accuracy. There could be multiple reasons for the same, like a high learning rate, a low number of epochs, or early stopping to avoid overfitting.
- 10 out of 12 models undergo early stopping, with their training stopping before 100 epochs are completed. This shows an indication of overfitting. The patience level for early stopping is set to 5.
- Among the weight initialization functions, all models have the lowest performance with the zero_init weight initialization function.
- Among the activation layers, leaky_relu has the lowest performance in terms of accuracy, and sigmoid has the lowest performance in terms of error reduction over increasing epochs.

SECTION C

1. **Preprocessing:**
   All the pixels are normalized by dividing by 255 to reduce them to a range of 0 to 1.

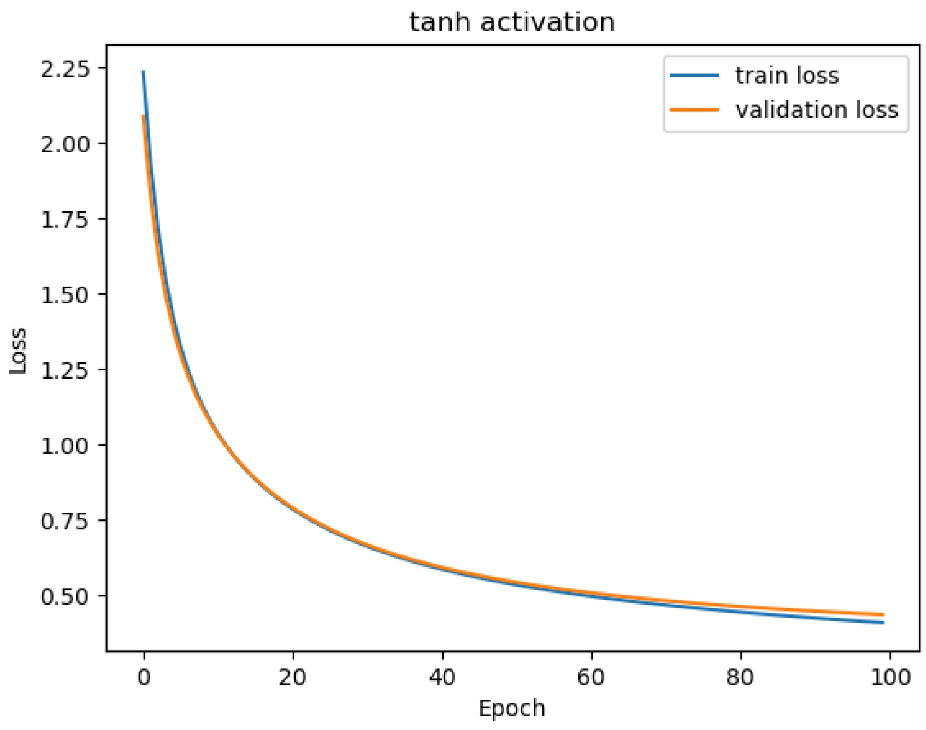


2. Logistic Activation Layer



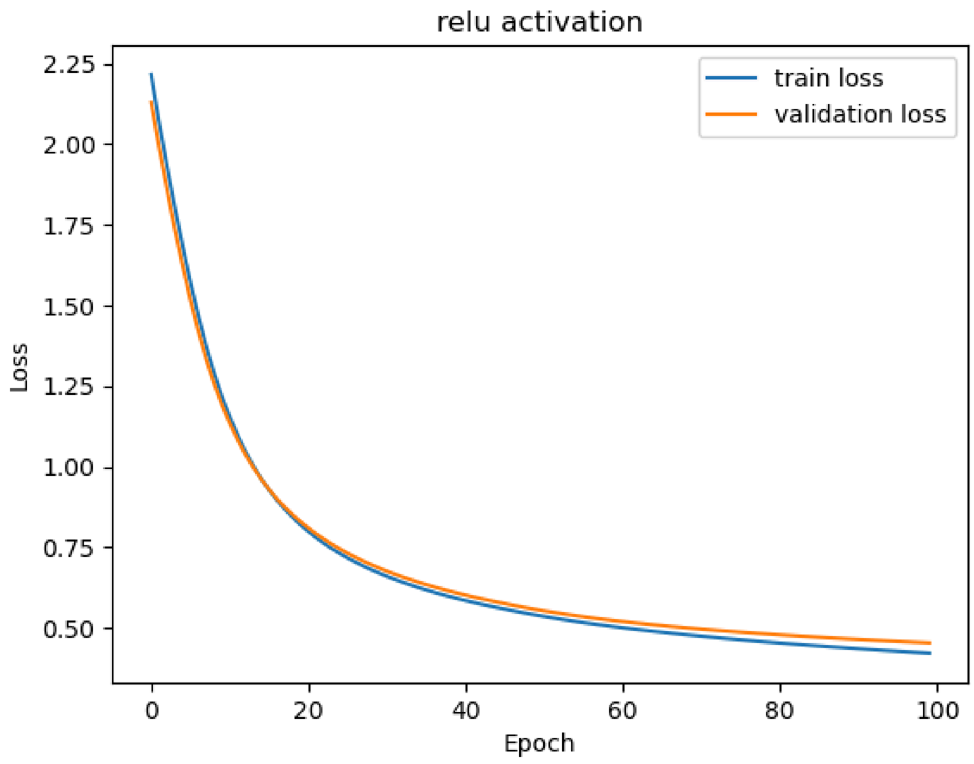
Accuracy: 44%

Tanh Activation Layer



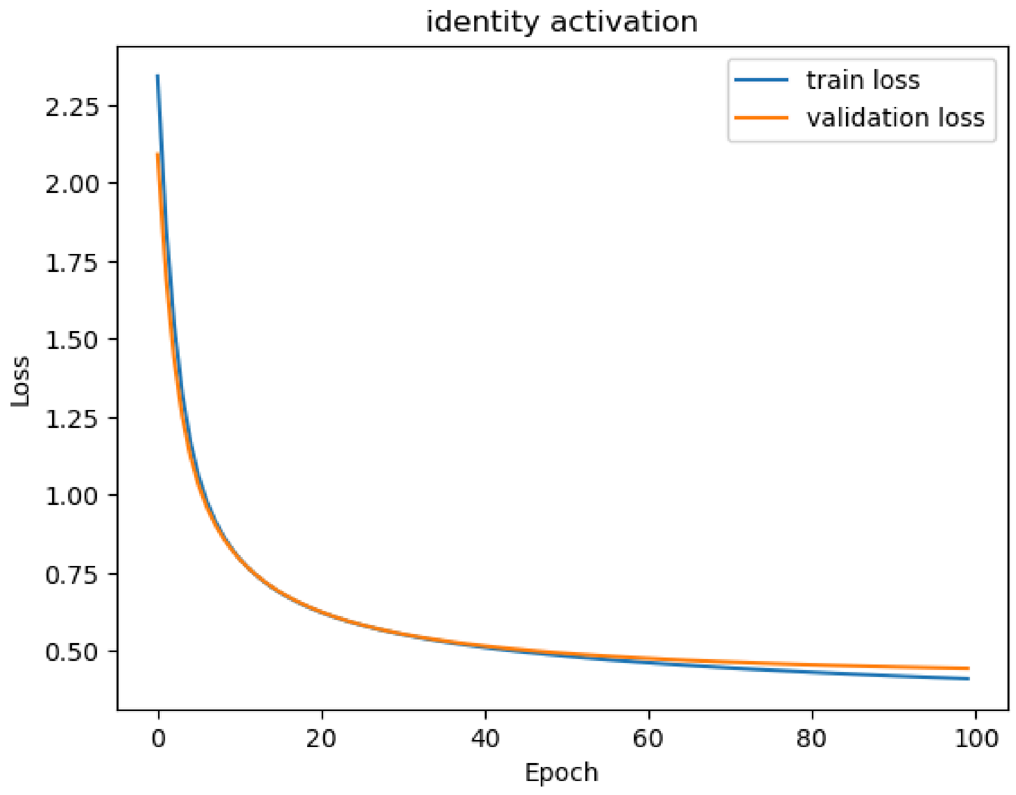
Accuracy: 86.35%

Relu Activation Layer



Accuracy: 85.95%

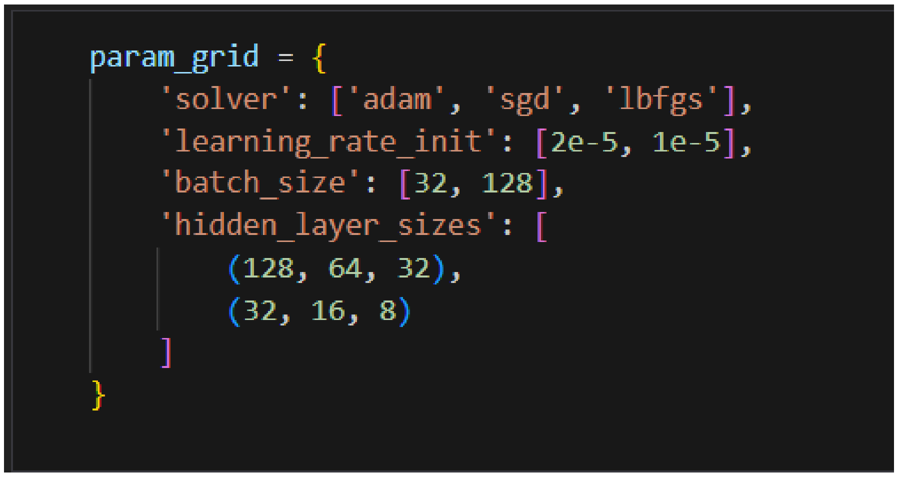
Identity Activation Layer



Accuracy: 86.1%

Tanh gave the best performance out of all the activation layers on the testing dataset, with a testing accuracy of 86.35%.
Sigmoid gave the worst performance out of all the activation layers on the testing dataset, with a testing accuracy of only 44%.

3. Since tanh is our best activation function, we perform GridSearch using the tanh activation function.
   We perform Grid Search with 72 fits shown below:

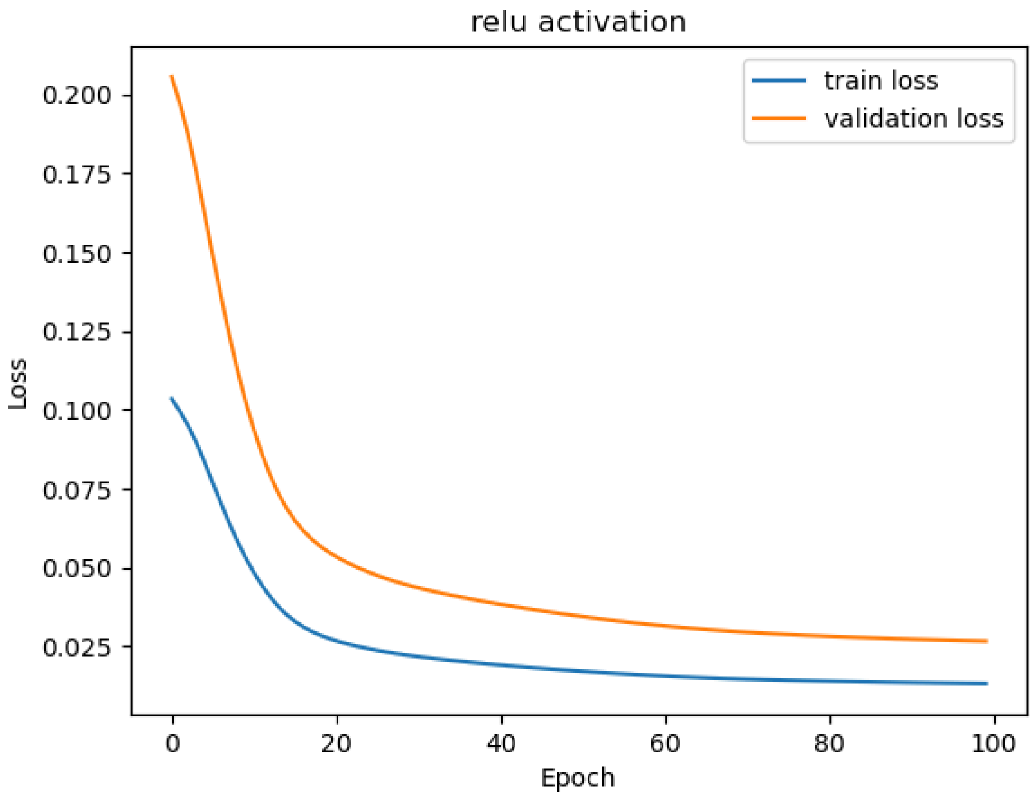


```
param_grid = {
    'solver': ['adam', 'sgd', 'lbfgs'],
    'learning_rate_init': [2e-5, 1e-5],
    'batch_size': [32, 128],
    'hidden_layer_sizes': [
        (128, 64, 32),
        (32, 16, 8)
    ]
}
```

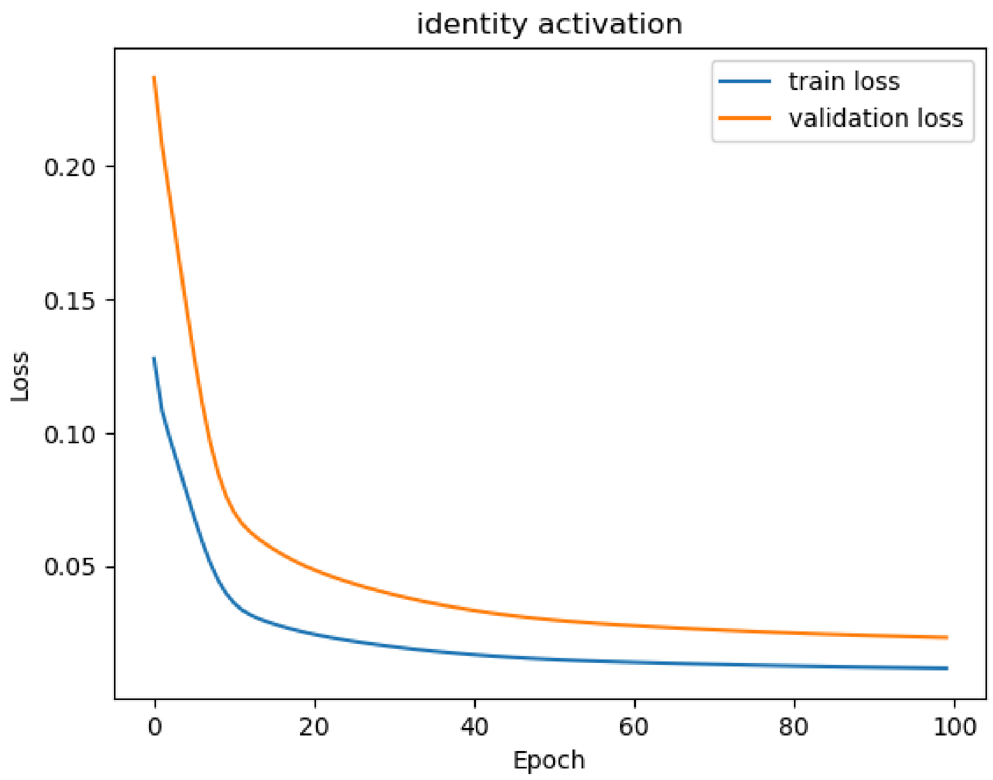
Out of all combinations, we get the best hyperparameters as the batch size of 32, hidden layer sizes of [128, 64, 32], the learning rate of 2e-5, and solver as adam. These hyperparameters give us a cross-validation score of 0.8482.

4. We take the layer size as [128, 64, 32, 64, 128], where we get a = 32. Since we are regenerating the image, we fit the images by comparing our outputs with the image itself.

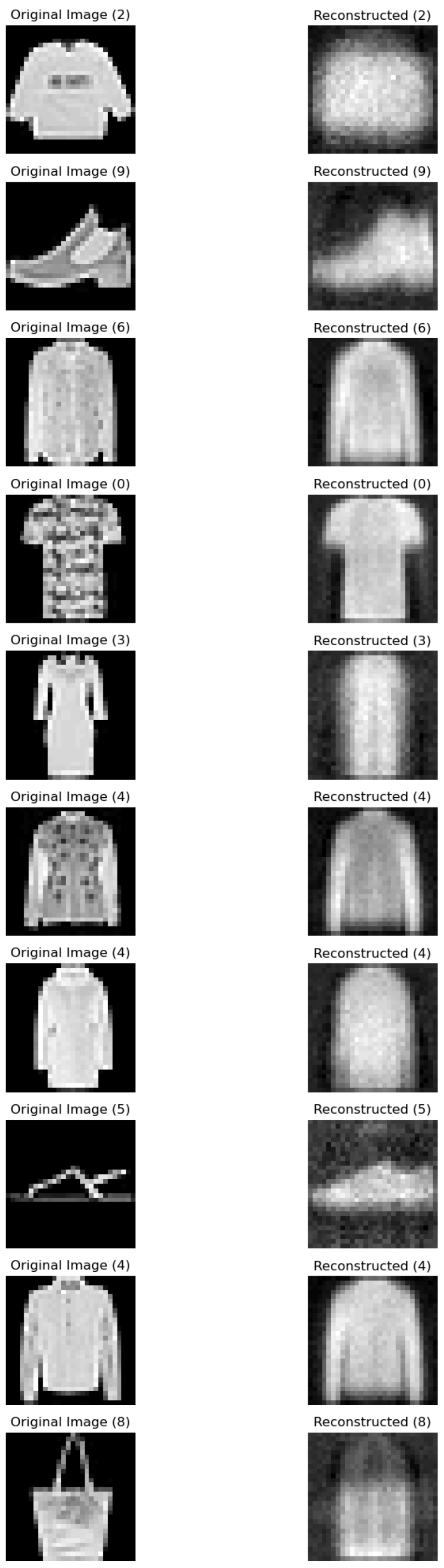
   Plot for relu activation



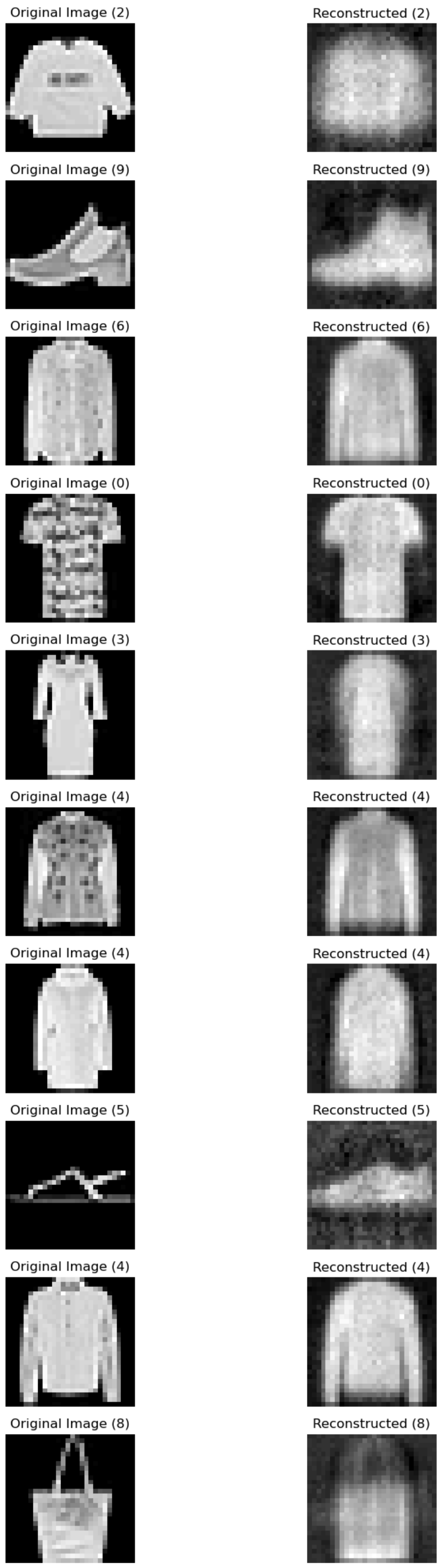
Plot for identity activation



Visualization for relu activation layer


Original Image (2)
Reconstructed (2)
Original Image (9)
Reconstructed (9)
Original Image (6)
Reconstructed (6)
Original Image (0)
Reconstructed (0)
Original Image (3)
Reconstructed (3)
Original Image (4)
Reconstructed (4)
Original Image (4)
Reconstructed (4)
Original Image (5)
Reconstructed (5)
Original Image (4)
Reconstructed (4)
Original Image (8)
Reconstructed (8)

Visualization for identity activation layer



Since there is a loss in training and validation for both the activation layers, the model does not achieve a 100% accuracy rate, and hence, during regeneration, we notice some blur and distortion in the regenerated image as compared to the original image, indication some fine details are lost during the training process. We notice that the outline of the images remains preserved, indicating that the model can capture basic structural information but not the finer details.

5. We are extracting the feature vector from an already-trained model. This extracted feature represents the image in low dimensions, which captures less relevant information than raw pixels. We use this extracted feature to train a new model with (a, a) layers. It gives a decent performance because it has already captured a decent amount of information from input data, helping the model to classify the images. Hence, the new model gives a decent performance compared to the model from Part 2.