

Deep Learning Report

Minor Exam - 1

Take Home Assignment

Ayush Abrol

B20AI052

Question 01

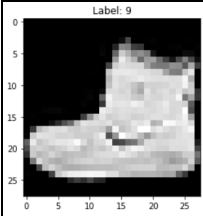
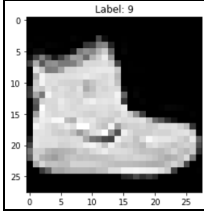
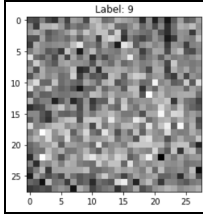
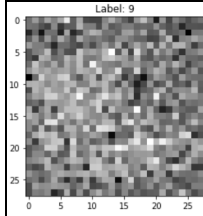
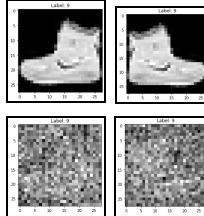
Aim: Train a CNN with the given details.

1. Following are the details which are variable for each student and my details are as follows:
 - 1. DD/MM/YY - 18/03/02
 - 2. ABC = 052
 - 3. FIRST = Ayush
 - 4. LAST = Abrol
 - 5. PROG - My Program
2. As ABC is even, dataset is [*FashionMNIST*](#).
3. As MM is odd, weight initialization is “*He initialization*”.
4. As DD is even, *flip and noise data augmentation* to be used.
5. As MM is odd, *MaxPool* to be used.
6. As Sum of DD, MM and YY is odd, *target 5-classes are 1, 3, 5, 7, 9* for the FashionMNIST dataset.
7. As last digit of $ABC < 6$ and ABC is even, therefore CNN architecture should have *4 conv layers and 1 pool layer*.
8. As ABC is even, network should have *10 filters in the first layer*.
9. As sum of digits of ABC is odd, network should have *1 Fully Connected layer with 256 nodes*.

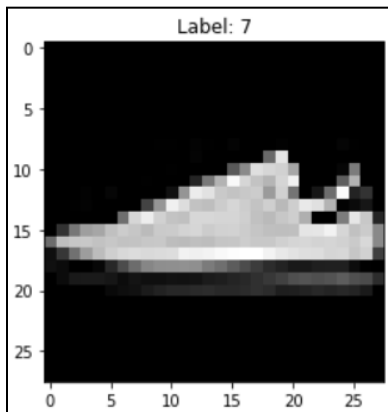
Procedure:

- All the necessary dependencies and libraries were taken care of initially like:
 - numpy, pandas, matplotlib for data manipulation and visualization.

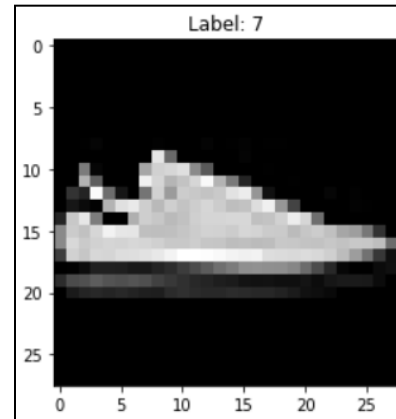
- Torch, torchvision, torch.nn, PIL for implementing Deep Learning architectures where necessary using PyTorch.
- Defining the **target classes** = ['1', '2', '3', '4', '5']
- Loading of the training and the testing **FashionMNIST dataset**. **60k training** samples and **10k testing** samples are loaded and converted into Torch format Tensors initially.
- Training Dataset Details:
 - Number of original training samples = 60000
 - Number of training samples after **selecting target classes** = **30000**
 - Number of extra training samples generated after **flip data augmentation** = **30000**
 - Number of training samples after applying **random noise data augmentation on original** selected samples = **30000**
 - Number of training samples after applying **random noise data augmentation on flipped** samples = **30000**
 - **Total** number of training samples after concatenation = $30000 * 4 = 120000$
- Testing Dataset Details:
 - Number of original test samples = 10000
 - Number of test samples after selecting target classes
- Data Augmentation visualized as follows:

Original Samples after selecting target classes	Samples generated after flipping the original.	Samples obtained after adding random noise to the original	Samples generated after random noise addition are flipped.	Total Number of Samples in the Training Dataset. (Concatenated Dataset)
				
30000	30000	30000	30000	1,20,000

- For flipping the images, I used PyTorch's `torch.flip` on the data with `dims=[3]` for flipping across y-axis (Horizontal Flip) because there would be no meaning of vertical flip on FashionMNIST dataset.

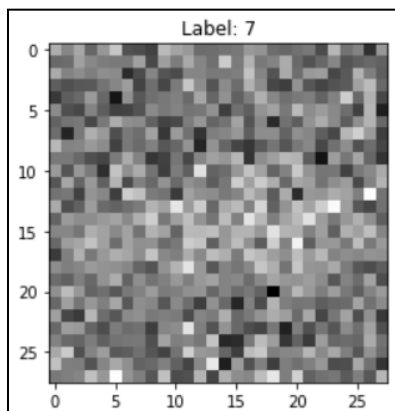


Original

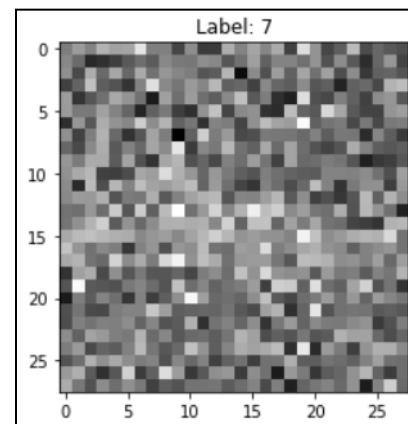


Flipped

- For adding random noise to the dataset, I used `torch.rand`.



Added random noise to original



Added random noise to flipped

- Final shapes of the dataset (`train_loader` and `test_loader`):

```
Train data shape: torch.Size([120000, 1, 28, 28])
Train labels shape: torch.Size([120000])
Test data shape: torch.Size([5000, 1, 28, 28])
Test labels shape: torch.Size([5000])
```

- Here, we are using **batch_size = 100**.
- Implemented the **CNN architecture** using torch.nn with 4 convolutional layers and 1 MaxPooling layer where:
 - Conv1: in_channels = 1, **out_channels = 10**, **filter/kernel size = 5x5**, padding = 1 (Here, number of filters used are 10 as mentioned in the question.)
 - Conv2: in_channels = 10, out_channels = 20, kernel size = 5x5, padding = 1
 - Conv3: in_channels = 20, out_channels = 40, kernel size = 5x5, padding = 1
 - Conv4: in_channels = 40, out_channels = 64, kernel size = 5x5, padding = 1
 - **MaxPooling layer: stride = 2, padding = 2**
 - Then, I also initialized the weights for the Conv layers and the FC layer using **kaiming_normal_method** of torch.nn.init which is also known as He initialization of weights.
 - Kaiming Initialization, or **He Initialization**, is an initialization method for neural networks that takes into account the non-linearity of activation functions, such as ReLU activations.
 - Also, the activation function used is **ReLU** from torch.nn.functional module.
 - Input to the first fully connected hidden layer is **64*10*10** where 64 are the number of channels obtained and 10*10 are the dimensions obtained by the last convolutinal layer. The output obtained from the last convolutional layer is firstly flattened to **(-1, 64*10*10)** and then passed to the FC layer.
 - The size of the first and only **Fully Connected layer is 256**.
 - Output layer is created at last after the FC layer of size 10. The size of the output layer is selected to be 10 instead of 5 because if select output_layer size=5, then for each image our output would be of the dimension 1x5 and our labels are 1, 3, 5, 7, 9. So, while calculating the losses if a sample with label 5, 7 or 9 comes in, it would be out of index for the obtained output vector of size 1x5. Therefore, if we select the output vector to be 1x10, this problem would not happen and losses would be calculated and propagated back in a smooth fashion.
- Then created the model object of the FashionCNN class and defined the loss criterion as **CrossEntropyLoss()** from torch.nn library.

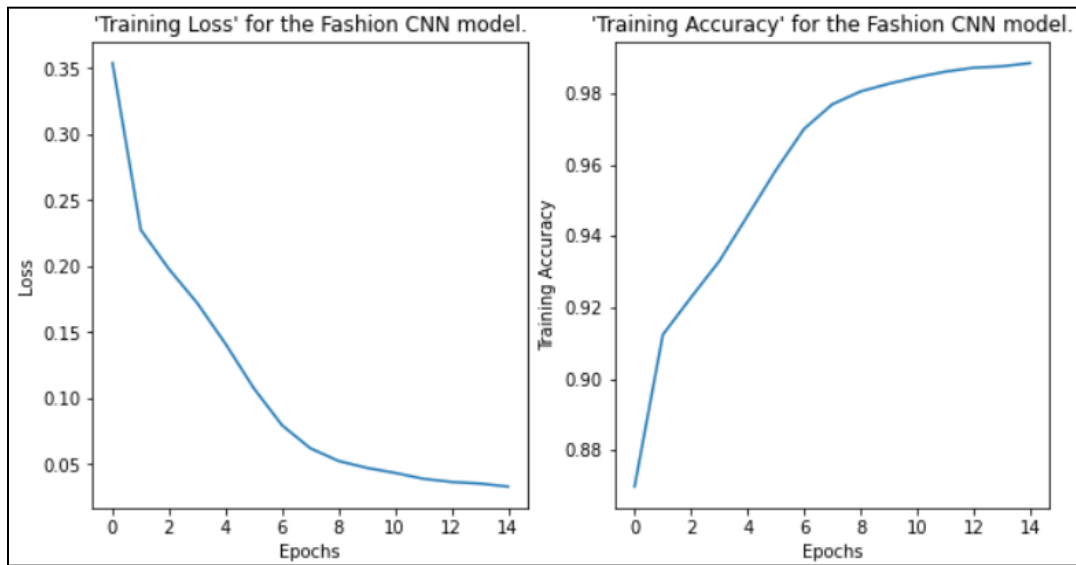
$$H(X) = \begin{cases} -\int_x p(x) \log p(x), & \text{if } X \text{ is continuous} \\ -\sum_x p(x) \log p(x), & \text{if } X \text{ is discrete} \end{cases}$$

[Source](#)

- Then, I defined optimizer as **Adam optimizer** using the **model parameters** as the optimizers' parameter and set the **learning rate as 0.001**.
- Then, I defined my train_model function with model, optimizer and number of epochs as its parameters. The functions first runs the epoch loop and inside the epoch loop, it enumerates over the training dataset, **initializes the gradient as 0**, then calculates the **loss between the output and the labels** batchwise using the loss criterion defined. After which the **loss is propagated backwards** and then **weights are updated** using optimizer.step().
- Average loss and average training accuracies for every epoch are stored in their corresponding arrays defined.
- I trained my model for 15 epochs and got the following results.

```
Epoch: 1 Training Loss: 0.3538566531551381 Training Accuracy: 0.8698166666666667
Epoch: 2 Training Loss: 0.2275140141385297 Training Accuracy: 0.912375
Epoch: 3 Training Loss: 0.1977256413642317 Training Accuracy: 0.9228333333333333
Epoch: 4 Training Loss: 0.1720017840899527 Training Accuracy: 0.933
Epoch: 5 Training Loss: 0.14115210263368985 Training Accuracy: 0.9456833333333333
Epoch: 6 Training Loss: 0.1075409666278089 Training Accuracy: 0.95845
Epoch: 7 Training Loss: 0.0794710762546553 Training Accuracy: 0.9700416666666667
Epoch: 8 Training Loss: 0.062070962660751926 Training Accuracy: 0.9769416666666667
Epoch: 9 Training Loss: 0.052417502853786574 Training Accuracy: 0.9805333333333334
Epoch: 10 Training Loss: 0.047192345543589906 Training Accuracy: 0.9826916666666666
Epoch: 11 Training Loss: 0.04337463993782876 Training Accuracy: 0.984525
Epoch: 12 Training Loss: 0.038947843393980296 Training Accuracy: 0.986075
Epoch: 13 Training Loss: 0.03651605885509828 Training Accuracy: 0.987175
Epoch: 14 Training Loss: 0.03531791846015646 Training Accuracy: 0.9875666666666667
Epoch: 15 Training Loss: 0.033015375331572915 Training Accuracy: 0.988475
Training complete!
```

- Then, I plotted the **loss curve and the training accuracy curve** for the results obtained.



- Then, I tested my FashionCNN model on the **test dataset of 5000 samples**.
- Got the **testing accuracy** on 5000 samples as **97.91%**.

END OF QUESTION 1

Question 02

Aim: Train an autoencoder with the given details:

- Following are the details which are variable for each student and my details are as follows:
 - 1. DD/MM/YY - 18/03/02
 - 2. ABC = 052
 - 3. FIRST = Ayush
 - 4. LAST = Abrol
 - 5. PROG - My Program
- As ABC is even, dataset is [FashionMNIST](#).
- As MM is odd, weight initialization is “*He initialization*”.
- As DD is even, *flip and noise data augmentation* to be used.
- As Sum of DD, MM and YY is odd, *target 5-classes* are 1, 3, 5, 7, 9 for the FashionMNIST dataset.
- As ABC is even, therefore the **number of encoding layers will be 4 and number of decoding layers will also be 4**.
- As MM is odd, the classification layer will be a **single Fully Connected layer with 256 nodes**.

Procedure:

- Used the same finally obtained dataset as above with the following details:

```
Train data shape: torch.Size([120000, 1, 28, 28])
Train labels shape: torch.Size([120000])
Test data shape: torch.Size([5000, 1, 28, 28])
Test labels shape: torch.Size([5000])
```

- This dataset includes all the data augmentation samples as well. Just like mentioned in the explanation of the previous question.
- Here also, we are using **batch_size = 100**.
- Then, I defined the AutoEncoder Class and implemented the Linear layers using torch.nn module, the details about the models are as follows:
 - Encoder layer 1: Linear layer with input_size=28*28, output_size=392.
 - Encoder layer 2: Linear layer with input_size=392, output_size=196.

- Encoder layer 3: Linear layer with input_size=196, output_size=98.
- Encoder layer 4: Linear layer with input_size=98, output_size=32.
- Decoder layer 1: Linear layer with input_size=32, output_size=98.
- Decoder layer 2: Linear layer with input_size=98, output_size=196.
- Decoder layer 3: Linear layer with input_size=196, output_size=392.
- Decoder layer 4: Linear layer with input_size=392, output_size=784.
- Then, I also initialized the weights for the Conv layers and the FC layer using **kaiming_normal_ method** of torch.nn.init which is also known as He initialization of weights.
- Kaiming Initialization, or **He Initialization**, is an initialization method for neural networks that takes into account the non-linearity of activation functions, such as ReLU activations.
- Also, the activation function used is **ReLU** from torch.nn.functional module.
- Then, I defined a Fully Connected layer for classification with input_size=output_size of the encoder layer 4 = 32 and output size of the FC layer is 256 according to question constraints provided.
- After which I defined the output layer with size=10 instead of 5 because of the same reason as explained in the previous question.
- Then I defined four different methods inside the AutoEncoder Class.
 - Encoder_func - This stacks all the encoder layers sequentially and also applying activation function ReLU.
 - Decoder_func - This stacks all the decoder layers sequentially also applying activation function ReLU.
 - Fully_connected_layer - This stacks the FC layer and the output layer sequentially and also applying the activation function ReLU.
 - Forward - Here input is first passed through the encoding layers and then through the decoding layers.
- I separately defined these four functions because later on we would only need the encoding part and the fully connected layers for classification purposes.
- Then I created the AutoEncoder object using the build class. Object has input_size= 784 and output_size=32 and is set to run in GPU runtime.
- Defined the loss criterion as **MSELoss()** from torch.nn library for training the AutoEncoder.

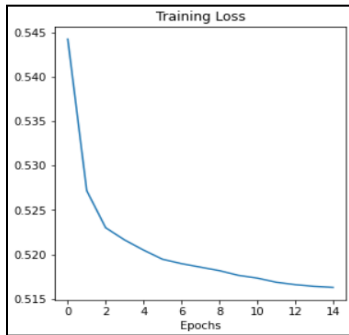
$$\text{MSE} = \frac{1}{N} \sum_{i=1}^N (y_i - \hat{y}_i)^2$$

[Source](#)

- Then, I defined optimizer as Adam optimizer using the model parameters as the optimizers' parameter and set the learning rate as 0.001.
- Then, I defined my train_model function with model, optimizer and number of epochs as its parameters. The functions first runs the epoch loop and inside the epoch loop, it enumerates over the training dataset, generates the output after applying both encoding and decoding layers, initializes the gradient as 0, then calculates the loss between the output image and the reconstructed image batchwise using the loss criterion defined. After which the loss is propagated backwards and then weights are updated using optimizer.step().
- Trained the AutoEncoder model for 15 epochs.
- Training loss and original images and the reconstructed are stored in their corresponding arrays created.
- Results for the training losses were:

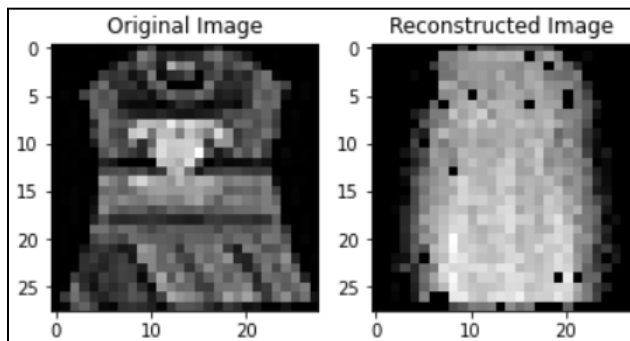
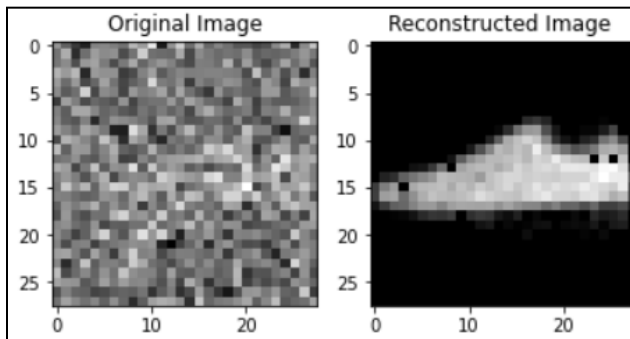
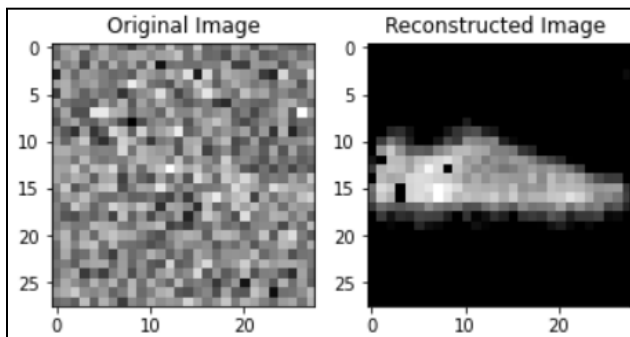
```
Epoch: 1 Training Loss: 0.5442641466856003
Epoch: 2 Training Loss: 0.5271821204572916
Epoch: 3 Training Loss: 0.5229976496348778
Epoch: 4 Training Loss: 0.5216205772012472
Epoch: 5 Training Loss: 0.5204752751191457
Epoch: 6 Training Loss: 0.5194438963631789
Epoch: 7 Training Loss: 0.5189500854164362
Epoch: 8 Training Loss: 0.5185533070067565
Epoch: 9 Training Loss: 0.5181595030178626
Epoch: 10 Training Loss: 0.5176326874395212
Epoch: 11 Training Loss: 0.5173179779946804
Epoch: 12 Training Loss: 0.5168532699843248
Epoch: 13 Training Loss: 0.5165804214775562
Epoch: 14 Training Loss: 0.5163878605763117
Epoch: 15 Training Loss: 0.5162704050292571
Training complete!
```

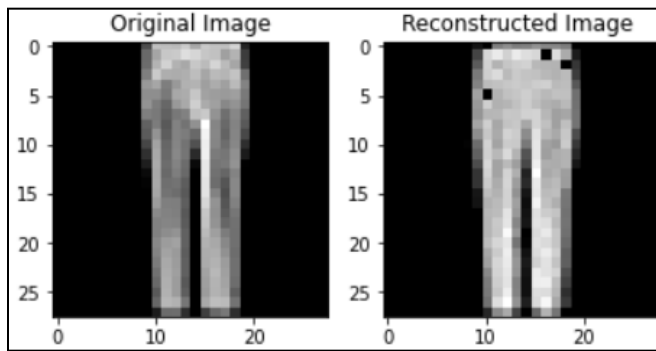
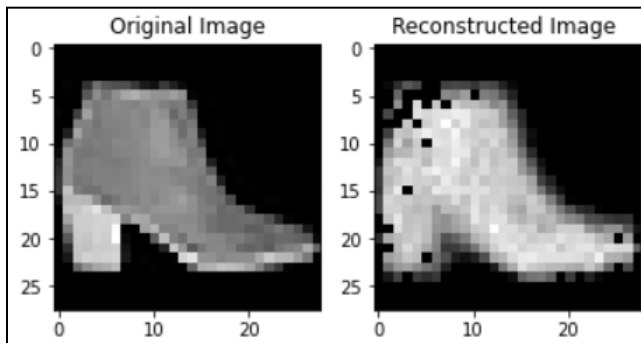
- Loss Curve obtained for AutoEncoder training:



Note: Loss is not converging that much here, this is due to the reason that this loss between the original images and the recon images.

- Then, I plotted the Original vs the Reconstructed images.



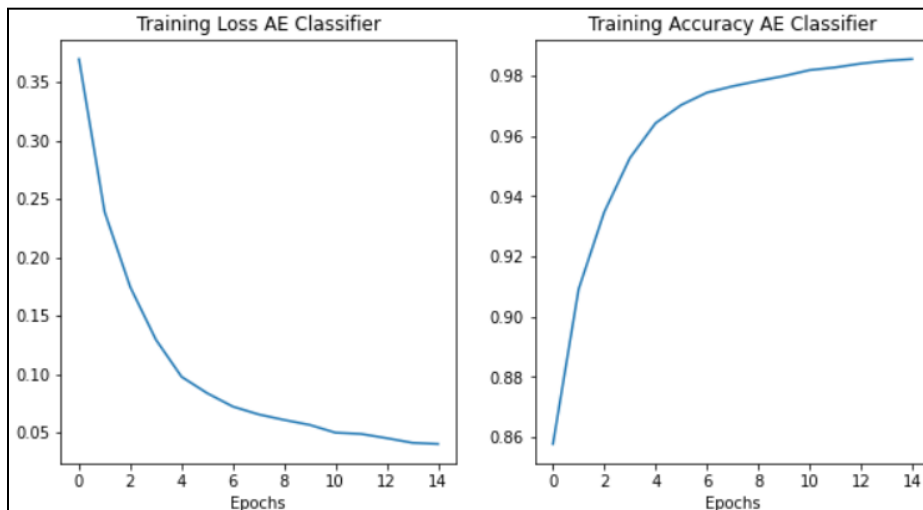


- Note that, images with added random noise are reconstructed to their original denoised form upto quite an extent.
- Then, I created another AutoEncoder object model with the same parameters but now the loss is defined as `CrossEntropyLoss()` and rest is the same.
- Now, while training this model, the function first runs the epoch loop and inside the epoch loop, it enumerates over the training dataset, generates the output predictions after applying both encoding and classification layers, initializes the gradient as 0, then calculates the loss between the predictions generated and the labels of the train data batchwise using the loss criterion defined. After which the loss is propagated backwards and then weights of the encoder and classifier are updated using `optimizer.step()`.
- Trained this classifier for 15 epochs.
- Training losses and training accuracies after epoch were stored in their respective arrays created.

- Results of the classification were as follows:

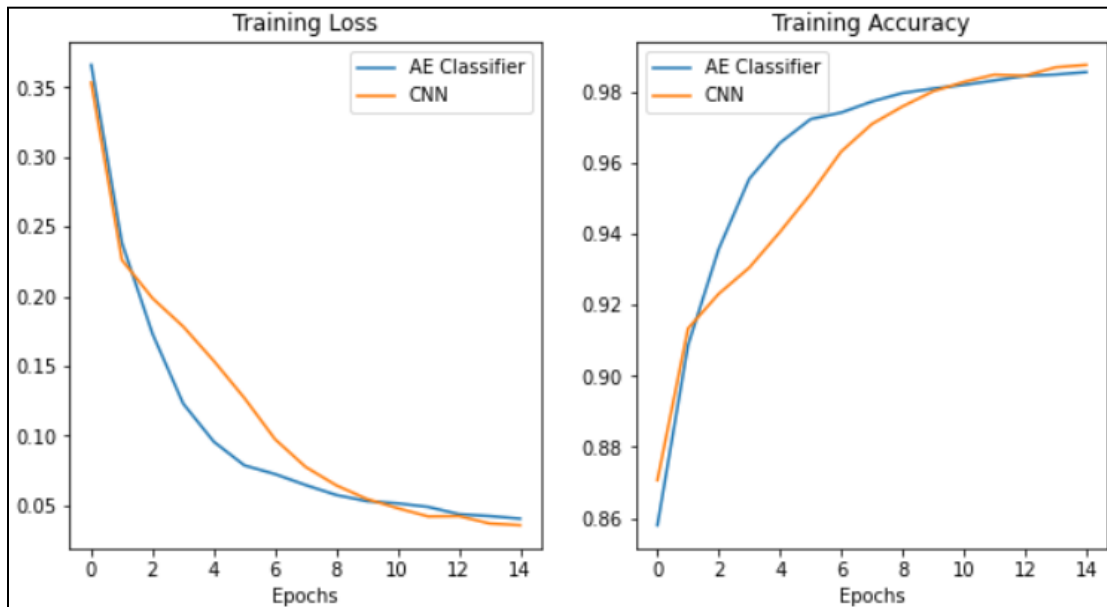
```
Epoch: 1 Training Loss: 0.369516979418695 Training Accuracy: 0.8577416666666666
Epoch: 2 Training Loss: 0.23874378515407443 Training Accuracy: 0.909175
Epoch: 3 Training Loss: 0.17442175868277748 Training Accuracy: 0.9348
Epoch: 4 Training Loss: 0.12945327683507155 Training Accuracy: 0.95275
Epoch: 5 Training Loss: 0.0977155584981665 Training Accuracy: 0.964325
Epoch: 6 Training Loss: 0.08367079127424706 Training Accuracy: 0.9703583333333333
Epoch: 7 Training Loss: 0.07221952037148488 Training Accuracy: 0.9744583333333333
Epoch: 8 Training Loss: 0.06554989252181258 Training Accuracy: 0.9765666666666667
Epoch: 9 Training Loss: 0.06079142937669531 Training Accuracy: 0.9783416666666667
Epoch: 10 Training Loss: 0.056541302589466796 Training Accuracy: 0.9799583333333334
Epoch: 11 Training Loss: 0.04990391247265506 Training Accuracy: 0.9819333333333333
Epoch: 12 Training Loss: 0.04882122898377323 Training Accuracy: 0.982825
Epoch: 13 Training Loss: 0.04518656421113216 Training Accuracy: 0.9841
Epoch: 14 Training Loss: 0.04122283932820816 Training Accuracy: 0.9850416666666667
Epoch: 15 Training Loss: 0.04022920399457992 Training Accuracy: 0.9856
Training complete!
```

- Then, I plotted the training loss and accuracy curves for the AutoEncoder classifier model after 15 epochs.



- Then, I tested my AutoEncoder model on the **test dataset of 5000 samples**.
- Got the **testing accuracy** on 5000 samples as **97.12%**.

- Comparison of Results between CNN and AutoEncoder:



- Comparison between the Test Accuracies of CNN and AutoEncoder Classifier:

Test Accuracy of CNN: 97.91999999999999
Test Accuracy of AE Classifier: 97.12