Queen’s University - Smith Engineering

ELEC 475 — Lab 1 Fall 2025

Submitted by:

Erhowvosere Otubu — Student Number: 20293052

Mihran Asadullah — Student Number: 20285090

Date Submitted: Saturday, October 4th, 2025

Table of Contents

Table of Figures

Model Details

The model implemented and trained for this lab was a 4-layer MLP autoencoder. It was designed to compress and reconstruct MNIST dataset images of 28x28 pixel format. This model uses four methods in the **model.py** module - the **__init__(), forward(), encode(),** and **decode()** methods. The init method found in Figure 1 is the class constructor that sets up the autoencoder's layers and registers them with PyTorch. Here are some important things to note about this method:

- Line 8

```
def __init__(self, N_input=784, N_bottleneck=8, N_output=784):
```

  - Designed for 28 x 28 grayscale images flattened to 784 features
  - The input and output are equal to the same value, 784, because the reconstruction (output) should be almost exactly as how the input was when it entered the model

- Line 10

```
N2 = 392
```

  - This sets up the hidden, fully connected width to half the size of the input

- Lines 11-12

```
self.fc1 = nn.Linear(N_input, N2)
self.fc2 = nn.Linear(N2, N_bottleneck)
```

  - This is the encoder section of the MLP autoencoder (784 -> 392 -> bottleneck)

- Lines 13-14

```
self.fc3 = nn.Linear(N_bottleneck, N2)
self.fc4 = nn.Linear(N2, N_output)
```

  - This is the decoder section of the MLP autoencoder (bottleneck -> 392 -> 784)

```
8       def __init__(self, N_input=784, N_bottleneck=8, N_output=784):
9           super(autoencoderMLP4Layer, self).__init__()
10          N2 = 392
11          self.fc1 = nn.Linear(N_input, N2)
12          self.fc2 = nn.Linear(N2, N_bottleneck)
13          self.fc3 = nn.Linear(N_bottleneck, N2)
14          self.fc4 = nn.Linear(N2, N_output)
15          self.type = 'MLP4'
16          self.input_shape = (1, 28*28)
```

Figure 1: Completed __init__() method from model.py module

The forward method in Figure 2 is how the data flows through the autoencoder. It propagates the information from layer to layer. It returns the decode method which in turn returns the encode method. Here are some important things to note about these methods and how they work together:

- Line 21

```
def encode(self, X):
```

  - Maps the input vector into the bottleneck

- Line 23 & 25

```
X = F.relu(X)
```

  - ReLU (Rectified Linear Unit) is called to introduce non-linearity

- Line 29

```
def decode(self, X):
```

  - Expands the bottleneck back into the input space to produce reconstruction

```
18      def forward(self, X):
19          return self.decode(self.encode(X))
20
21      def encode(self, X):  3 usages
22          X = self.fc1(X)
23          X = F.relu(X)
24          X = self.fc2(X)
25          X = F.relu(X)
26
27          return X
28
29      def decode(self, X):  4 usages
30          X = self.fc3(X)
31          X = F.relu(X)
32          X = self.fc4(X)
33          X = torch.sigmoid(X)
34
35          return X
```

Figure 2: Completed forward(), encode(), and decode() methods from model.py module

Figure 3 shows a visual representation of a simple MLP autoencoder that the model.py module represents with Python code.

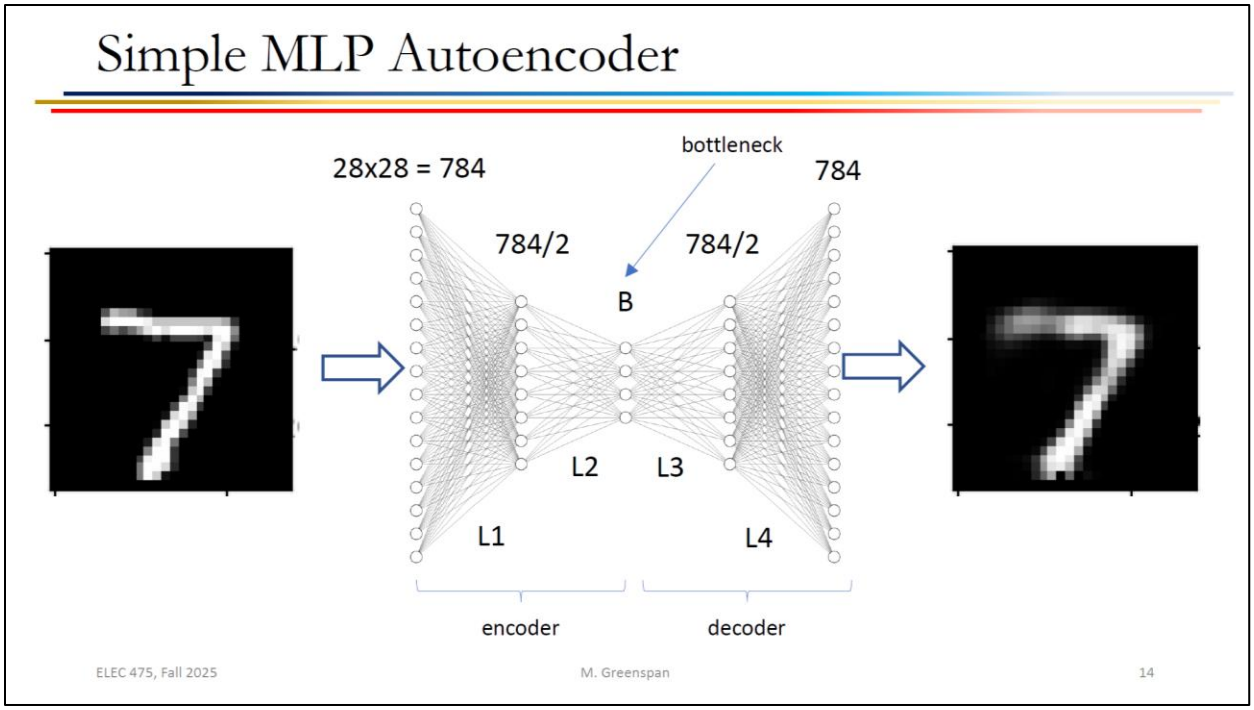

Figure 3: Visual representation of simple MLP autoencoder from Professor Greenspan lecture slides [1]

In general, the purpose of this autoencoder is to use the encoder to compress the images to a compact bottleneck representation and use the decoder to reconstruct the image based on the compressed, latent space. Figure 4 below shows a layer-by-layer configuration generated by calling to **torchsummary**.

```
               bottleneck size =  8
               n epochs =  50
               batch size =  2048
               save file =  MLP.8.pth
               plot file =  loss.MLP.8.png
               using device  cpu
==========================================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================================
├─Linear: 1-1                            [-1, 1, 392]              307,720
├─Linear: 1-2                            [-1, 1, 8]                3,144
├─Linear: 1-3                            [-1, 1, 392]              3,528
├─Linear: 1-4                            [-1, 1, 784]              308,112
==========================================================================================
Total params: 622,504
Trainable params: 622,504
Non-trainable params: 0
Total mult-adds (M): 0.62
==========================================================================================
Input size (MB): 0.00
Forward/backward pass size (MB): 0.01
Params size (MB): 2.37
Estimated Total Size (MB): 2.39
==========================================================================================
training ...
```

Figure 4: The call to torchsummary returned output

Training Details

The training for this lab was performed using PyTorch within the PyCharm environment. The dataset used for training this model was the MNIST training set. It is a dataset containing around 60,000 images. The model was trained by executing the following command in the PyCharm terminal:

```
python train.py -z 8 -e 50 -b 2048 -s MLP.8.pth -p loss.MLP.8.png
```

Figure 5: Command line arguments called in the PyCharm terminal to run the training process

Here are some things to note about the command:

- -z 8 → the size of the bottleneck
- -e 50 → the number of training epochs
- -b 2048 → the batch size
- -s → the name of the file used to save the train model parameters
- -p → the name of the file used to save the loss curve plot

The optimization method used in the train.py module was the Adam optimizer which updates the model's weights after each batch to minimize the loss. To simply put, this optimizer updates the weights to minimize loss.

```
optimizer = optim.Adam(model.parameters(), lr=1e-3, weight_decay=1e-5)
```

- Learning rate (lr = 1e-3) → hyperparameter used to controls how big each step in weight space is
- Weight decay (1e-5) → hyperparameter that adds a small penalty to large weights to prevent overfitting

The scheduler used was ReduceLROnPlateau. Its purpose is to reduce the learning rate when a metric has stopped improving and in this lab it was applied once per epoch based on the running training loss. It was assigned the 'min' mode which means the learning rate would be reduced when the quantity monitored has stopped decreasing [1].

```
scheduler = optim.lr_scheduler.ReduceLROnPlateau(optimizer,'min')
```

The loss function used the Mean Squared Error (MSE) loss function. It measures how different the reconstructed image is from the original image. The autoencoder learns to rebuild the image and so the

MSE loss tells it how close the reconstruction is. A high loss means the reconstruction looks very different from the input whereas a low loss means the model is accurately recreating the image.

```
loss_fn = nn.MSELoss(size_average=None, reduce=None, reduction='mean')
```

In general, the model adjusted its weights through the Adam optimizer to minimize this MSE loss, meaning it learned to reproduce cleaner, more accurate images over time.

Results

The autoencoder successfully learned a compression representation of MNIST dataset images and produced a mostly visually accurate reconstruction. The model reconstructed most digits and persevered overall structure and key features. Some reconstructions showed slight blurring especially with images that had finer details like 8 or 9 as seen below in Figure 6.

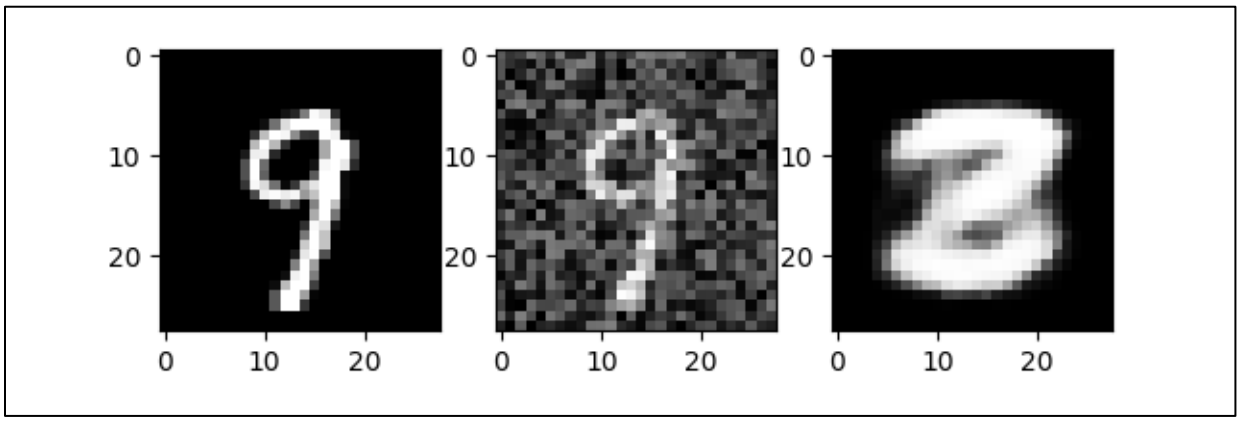

Figure 6: Image denoising example of the number '9'

When adding noise to input images the autoencoder was able to successfully denoise and analyze the underlying digit. Although the denoised outputs showed to be smoother than the originals, the digit recognition remained clear. During linear interpolation the bottleneck was able to produce smooth transitions between two images, effectively transforming one digit into another as see in Figure 7.

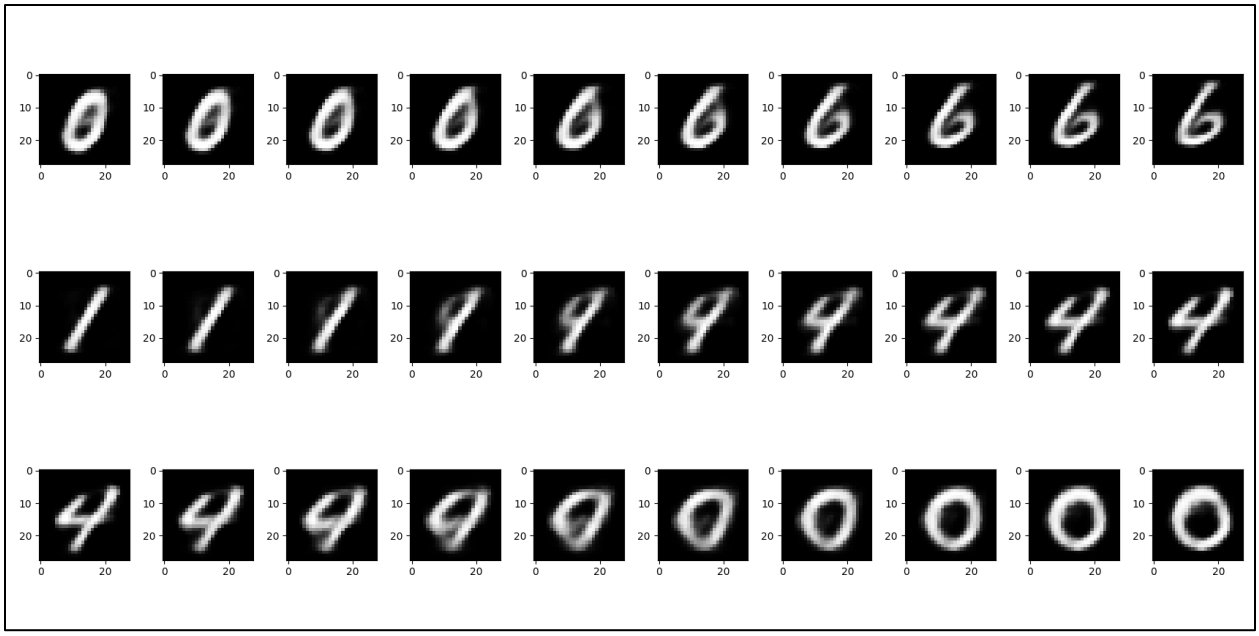

Figure 7: Plot of linear interpolation of 8 steps between two images

Figure 8 below shows the training loss over 50 epochs. As seen in the plot, the loss decreases rapidly during the first few epochs (~1 to 10 epochs), indicating that the model quickly learned to reconstruct the MNIST images with lower error. After 10-15 epochs, the curve begins to flatten, showing that the learning rate of improvement slows down as the model approaches convergence. By the end of training, the loss stabilizes around 0.02. We believe that the autoencoder effectively minimized reconstruction errors and achieved consistent performance. Based off the smooth and steadily trend decrease, it appears that the training process was stable, the Adam optimizer was effective, and the learning rate scheduler helped fine-tune the optimization. Overall, the shape of the curve confirms that the model trained successfully.

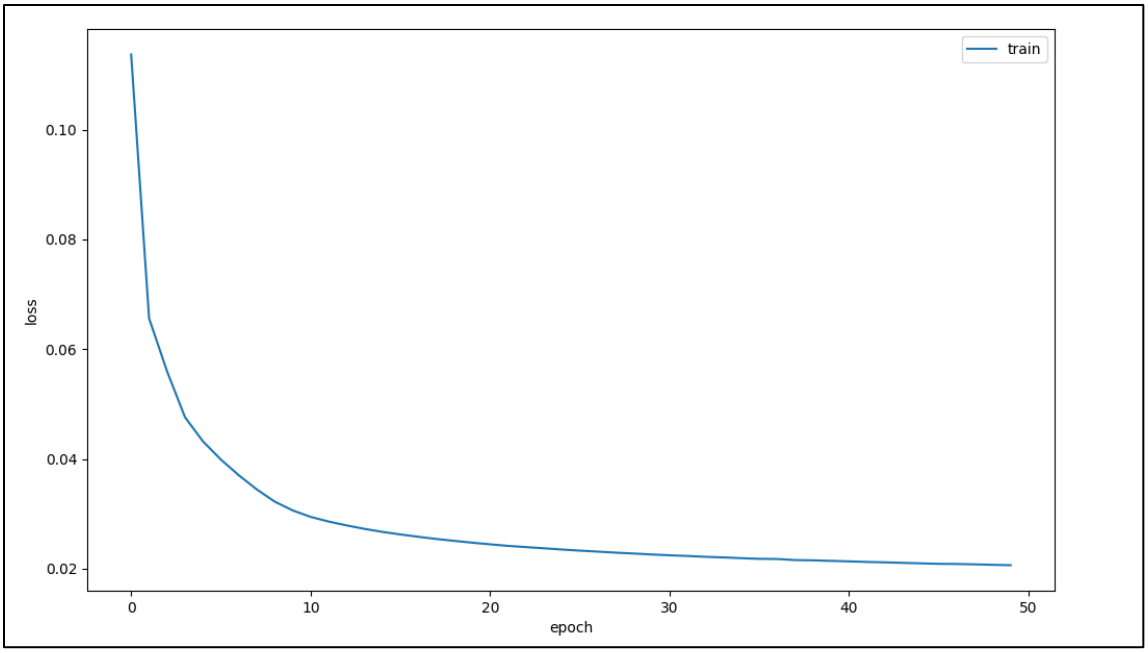

Figure 8: Loss curve plot generated from training session

Works Cited

[1] M. Greenspan, "ELEC 475 MLP Autoencoder," Queen's University. [Online]. [Accessed 27 September 2025].

[2] Keras Team, "ReduceLROnPlateau," Kereas, [Online]. Available: https://keras.io/api/callbacks/reduce_lr_on_plateau/. [Accessed 27 September 2025].