1-MLP Autoencoder

Setareh Soltanieh

Dataset:

In this lab, we are using the **MNIST dataset** (Modified National Institute of Standards and Technology), a widely-used dataset in machine learning and computer vision. It consists of grayscale images, each measuring 28x28 pixels, depicting handwritten digits ranging from 0 to 9. Each image is labeled with the corresponding digit. The dataset is typically split into **60,000 training images** and **10,000 testing images**.

Preprocessing:

We have applied a **tensor transformation** to normalize the images, ensuring that their pixel values are scaled within the range of 0 to 1.

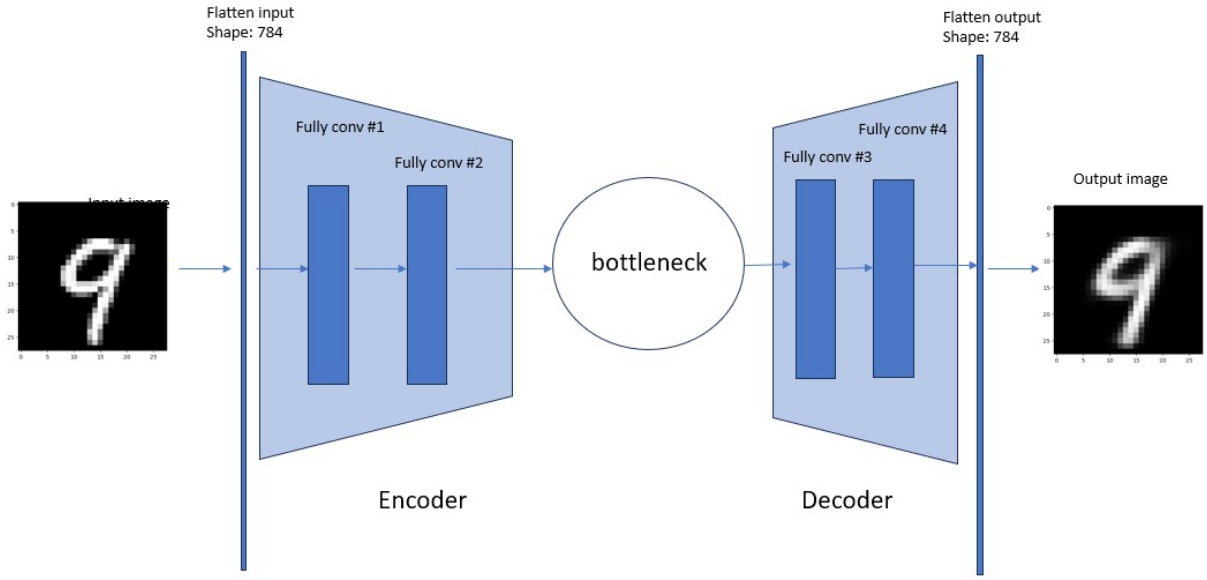
Model:

The model we used is an **autoencoder**. It takes a **flattened input** (the 28x28 image transformed into a 784-dimensional vector) and passes it through a **fully connected layer** with an output size of 392. This layer is followed by a **ReLU activation function**.

Next, the data is passed through another fully connected layer that reduces the output to a size of 8, representing the **bottleneck**. After another ReLU activation, this completes the **Encoder** phase, which compresses the input data to a smaller representation that retains essential information.

The **Decoder** phase begins by passing the 8-dimensional bottleneck representation through a fully connected layer that expands it back to 392 dimensions, again followed by a ReLU activation. Finally, another fully connected layer with a **sigmoid activation function** is applied, transforming the data back to 784 dimensions, the same size as the original input image.

This architecture forms the complete autoencoder, where the **Encoder** reduces the data and the **Decoder** reconstructs the input from the compressed representation.



The summary of our model is as follows:

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Linear-1               [-1, 1, 392]         307,720
            Linear-2                 [-1, 1, 8]           3,144
            Linear-3               [-1, 1, 392]           3,528
            Linear-4               [-1, 1, 784]         308,112
================================================================
Total params: 622,504
Trainable params: 622,504
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.00
Forward/backward pass size (MB): 0.01
Params size (MB): 2.37
Estimated Total Size (MB): 2.39
----------------------------------------------------------------
```

This model has 622504 parameters that we need to train them.

Training Details

We implemented a `train` function that takes the following inputs:

n_epochs: The total number of training iterations for the model. In this case, we trained the model for **50 epochs**.

Optimizer: The optimizer minimizes the loss function by adjusting the model's parameters. We used the **Adam optimizer** with a learning rate of **1e-3**.

Model: The model to be trained. In this case, it is **autoencoderMLP4Layer()**.

Loss Function: We used the **Mean Square Error (MSE)** loss function to calculate the difference between the model's output and the true labels.

train_loader: The training dataset, which contains **60,000 images**, is loaded using the **DataLoader** function, which creates data batches and shuffles them for each epoch.

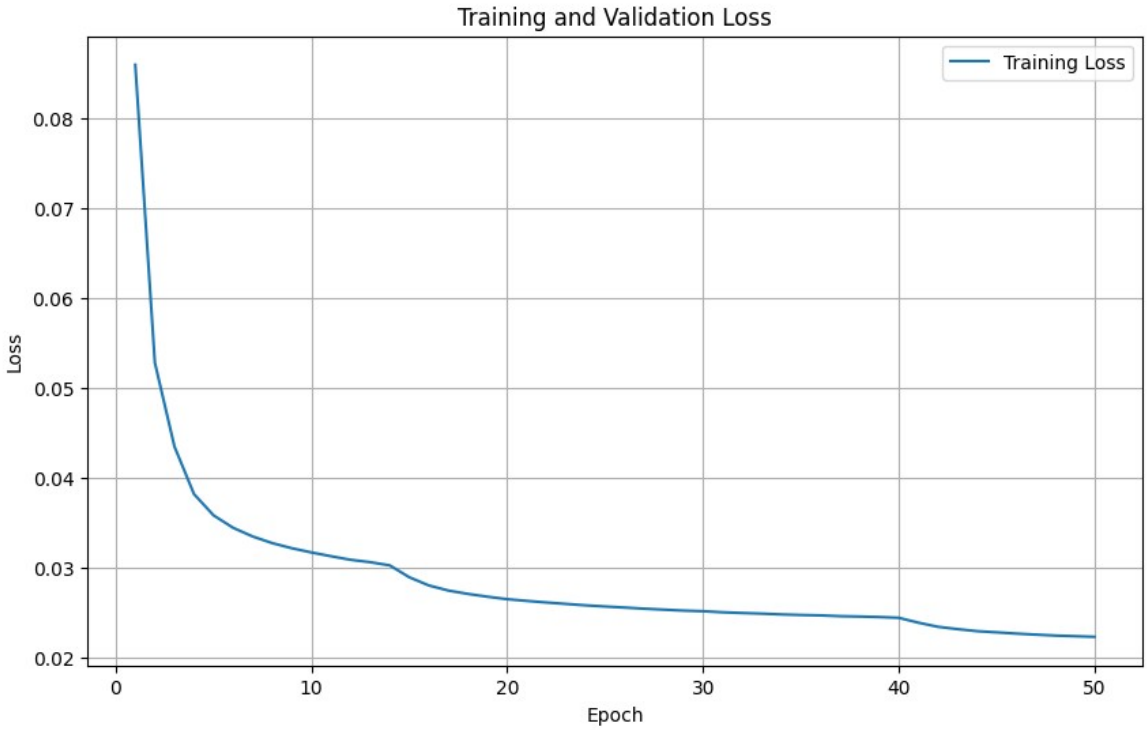
Scheduler: We used a **Reduce on Plateau** scheduler, which decreases the learning rate if the loss does not improve for 5 consecutive epochs. It reduces the learning rate with a weight decay of **1e-5**.

Device: This indicates whether a **GPU** is available for training, which improves performance.

During training, a `for` loop iterates through each epoch. For each batch of data, the model processes the input, the loss is calculated, and the model's weights are updated accordingly. We printed the loss for each epoch to monitor the model's performance.

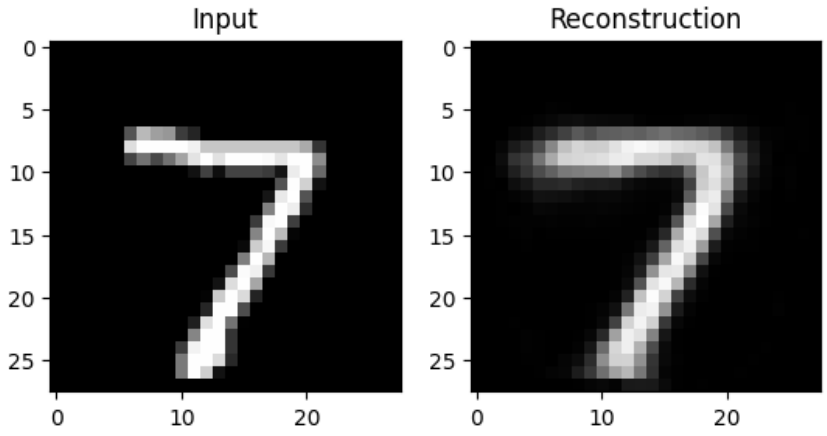
Results

The model was successfully trained, and the following are the results of the training losses.



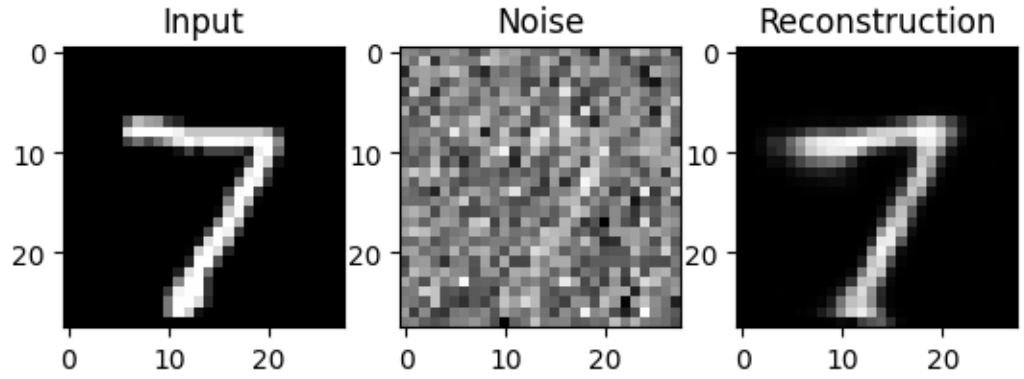
As shown in the loss function plot, the loss consistently decreases throughout the training process. There are noticeable bends in the plot around **epoch 15** and **epoch 40**, which can be attributed to the scheduler reducing the learning rate at these points. This adjustment helps optimize the loss function more effectively in the later stages of training.

After completing the training, we tested the model on the **testing set**. We input a test image into the model and displayed both the original input and the resulting reconstructed output, as shown in the following image.



As shown, the model performs well, successfully learning to reconstruct the digit **7** and generating a new handwritten version of it.

In addition to reconstruction, **autoencoders** can also be used for **image denoising**. We added random noise to the images and passed the noisy images through the model. The model effectively removed the noise and reconstructed the clean images. The following figure displays the original input, the noisy image, and the denoised output generated by the model.



As demonstrated, our model successfully removed noise from the noisy images and was able to reconstruct the clean images.

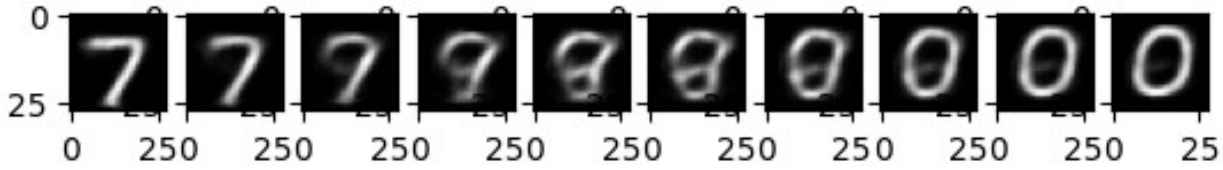
Bottleneck Interpolation

In this section, we extended the model by adding two new methods: **encoder** and **decoder**. These were previously described in the model section. Here, we defined a function that takes two images as input, calculates their **bottleneck vectors**, and applies the following interpolation formula:

```
interpolated_bottleneck = bottleneck_1 + (i / n_interpolation) *
(bottleneck_2 - bottleneck_1)
```

This function computes a weighted average between the two bottleneck vectors, producing new bottlenecks. The number of interpolated bottlenecks depends on the value of **n_interpolations**. After calculating the interpolated bottleneck, we pass it through the decoder to generate the final image.

The following images show the results for **10 interpolations** between the digits **7** and **0**. As the interpolation progresses, the images gradually shift to resemble one of the input images more closely.



As shown, the interpolated images gradually resemble their respective input images more closely as they approach the original digit during the interpolation process.