ELEC 475 – Lab 01

Department of Electrical and Computer Engineering
Queen's University

Composed By
Nicholas Seegobin (20246787)

Date of Submission
2024 September 24

"I hereby attest that the work submitted is my own individual work and that no portion of this submission has been copied in whole or in part from another source, with the possible exception of properly referenced material."

Introduction

The purpose of the Multilayer Perceptron (MLP) autoencoder is to take input data, such as the MNIST images, and compress it into a smaller, more simplified form. By reducing the size of the data (dimensionality), the model focuses on the most important features while discarding unnecessary details. Even after this compression, the key information needed to represent the data is preserved. The autoencoder then uses these essential features to reconstruct the original data with minimal loss in quality.

To achieve this, the input image, which is originally 28x28 pixels, must first be converted into a 1D vector of size 784 before passing through the network. Fully connected layers, which are used in this autoencoder, require inputs in a flat, vector format rather than a 2D grid. Flattening the image arranges all the pixel values into a single row, treating each pixel as an independent input. These layers assign weights to each input to learn the relationships between them. During training, the model adjusts these weights based on performance, gradually learning which pixels are more important and how they relate to each other. This process allows the autoencoder to identify patterns, such as edges or shapes, within the image. As the weights are updated, the model becomes better at compressing and reconstructing the image by focusing on the most important relationships between pixels, ultimately reducing errors and preserving accuracy.

In the diagram of the MLP autoencoder shown in Figure 1, each dot represents a neuron in a layer. The input layer (L1) contains 784 neurons, one for each pixel of the image. Neurons in the hidden layers (L2 and L3) take inputs from all the neurons in the previous layer, performing a weighted sum of these inputs, followed by an activation function (like ReLU or Sigmoid) to produce an output.

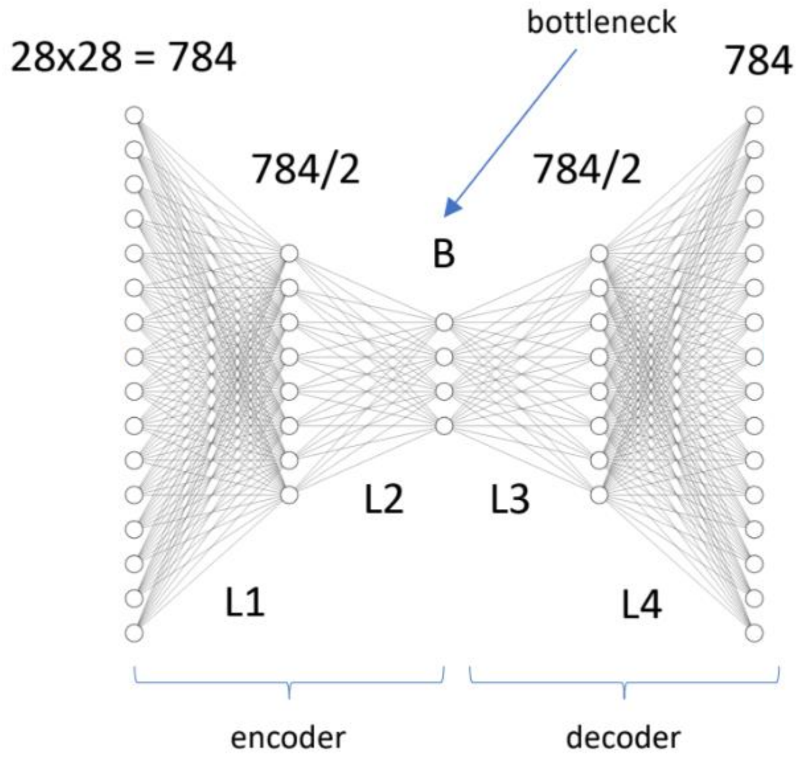


Figure 1: A diagram of the 4-Layer MLP Autoencoder used in the lab.

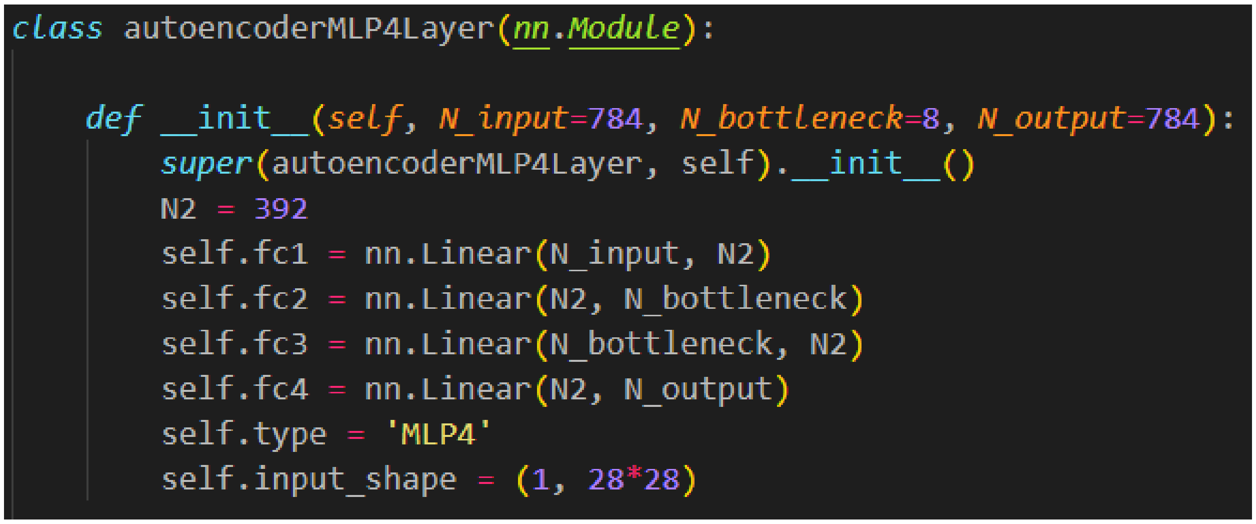
The lines connecting the neurons represent the weights between consecutive layers, which determine the strength of the signal passed from one neuron to the next. Each weight is multiplied by the corresponding input and summed with the others before being passed through an activation function. During training, these weights are adjusted to reduce errors, allowing the autoencoder to learn how to compress and reconstruct the input data.

Each neuron has multiple weights because it is connected to all neurons in the previous layer. This connectivity allows the model to capture complex relationships between pixels or features. As data passes through the layers, it is compressed in the bottleneck layer (B), where the dimensionality is reduced to 8 neurons. This compressed information is then used to reconstruct the image in the output layer (L4), minimizing the loss of important details while maintaining accuracy.

Part 1: Model Details

The autoencoder implemented is a 4-layer Multilayer Perceptron (MLP) designed to compress input images into a smaller representation and then reconstruct the original image from that compressed form. The model consists of an encoder and a decoder, with the goal of learning a compressed representation that retains the most important information about the input image while discarding unnecessary details.

The __init__() method (see Figure 2) initializes the model with three key parameters: N_input, N_bottleneck, and N_output. By default, the input size is 784 (representing a flattened 28x28 MNIST image), the output size is also 784 to match the input, and the bottleneck, which stores the compressed version of the input, has 8 neurons. The model is built with four fully connected layers: fc1 reduces the input from 784 to 392 neurons, fc2 further reduces it to 8 neurons (the bottleneck), fc3 expands the 8 neurons back to 392, and fc4 reconstructs the 392 neurons into the original 784-pixel image. This structure ensures the data is compressed efficiently and then restored with minimal loss.

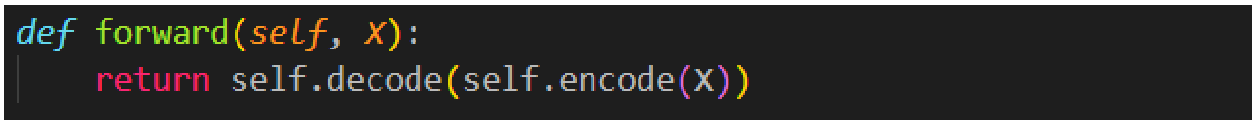


```
class autoencoderMLP4Layer(nn.Module):

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

Figure 2: An image of the __init__() method.

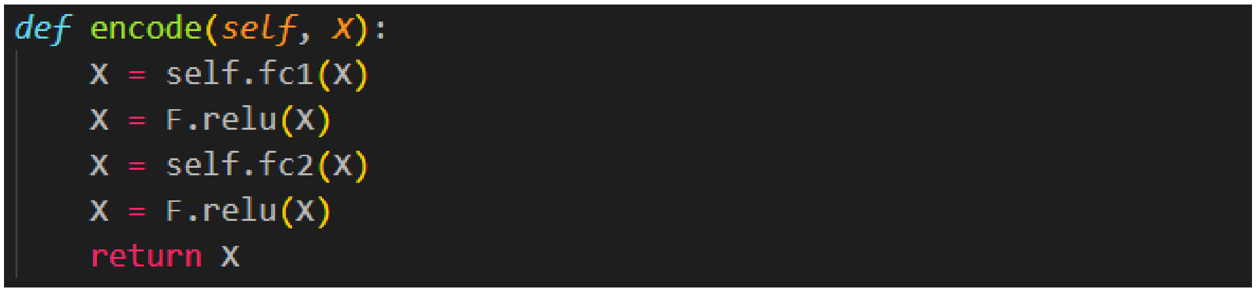
The forward() method (see Figure 3) defines how data flows through the network. It processes the input first through the encoder, where it is compressed, and then through the decoder to reconstruct the original data. The method combines the functionality of encode() (see Figure 4) and decode() (see Figure 5), moving the data from the input to the bottleneck and back to the original image format in one pass.



```python
def forward(self, X):
    return self.decode(self.encode(X))
```

Figure 3: An image of the forward() method.

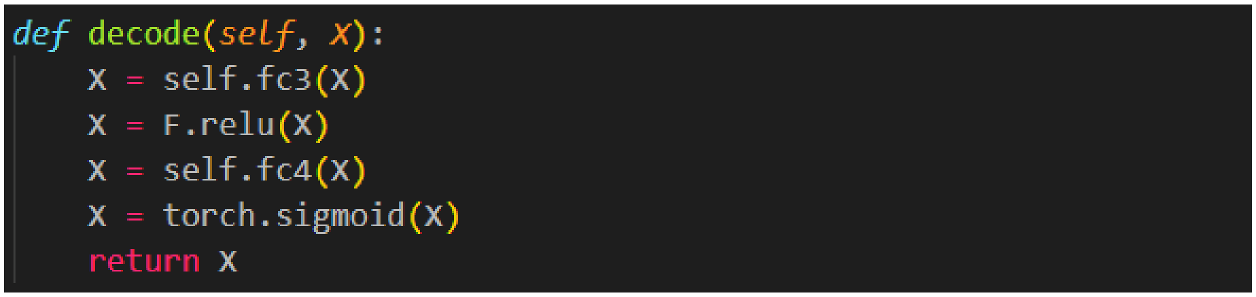
In the encode() method (see Figure 4), the input is passed through fc1, reducing the dimensionality from 784 to 392 neurons. A ReLU activation function is applied after this layer to introduce non-linearity, which allows the model to learn complex relationships in the data. The compressed representation is further reduced by fc2 to the bottleneck size of 8 neurons, followed by another ReLU activation to maintain non-linearity during compression. This process creates the smaller, more efficient representation of the input image.



```python
def encode(self, X):
    X = self.fc1(X)
    X = F.relu(X)
    X = self.fc2(X)
    X = F.relu(X)
    return X
```

Figure 4: An image of the encode() method.

The decode() method (see Figure 5) reconstructs the compressed representation. First, fc3 expands the bottleneck layer back to 392 neurons, again followed by a ReLU activation. Then, the final layer fc4 restores the 392 neurons back to 784 neurons, reconstructing the original input. A Sigmoid activation function is applied at the end, ensuring that the output values are between 0 and 1, matching the pixel intensity range of the original MNIST images. The use of ReLU activations in the hidden layers allows the model to capture non-linear patterns during both compression and reconstruction, while the Sigmoid activation ensures that the output is suitable for image reconstruction.



```python
def decode(self, X):
    X = self.fc3(X)
    X = F.relu(X)
    X = self.fc4(X)
    X = torch.sigmoid(X)
    return X
```

Figure 5: An image of the decode() method.

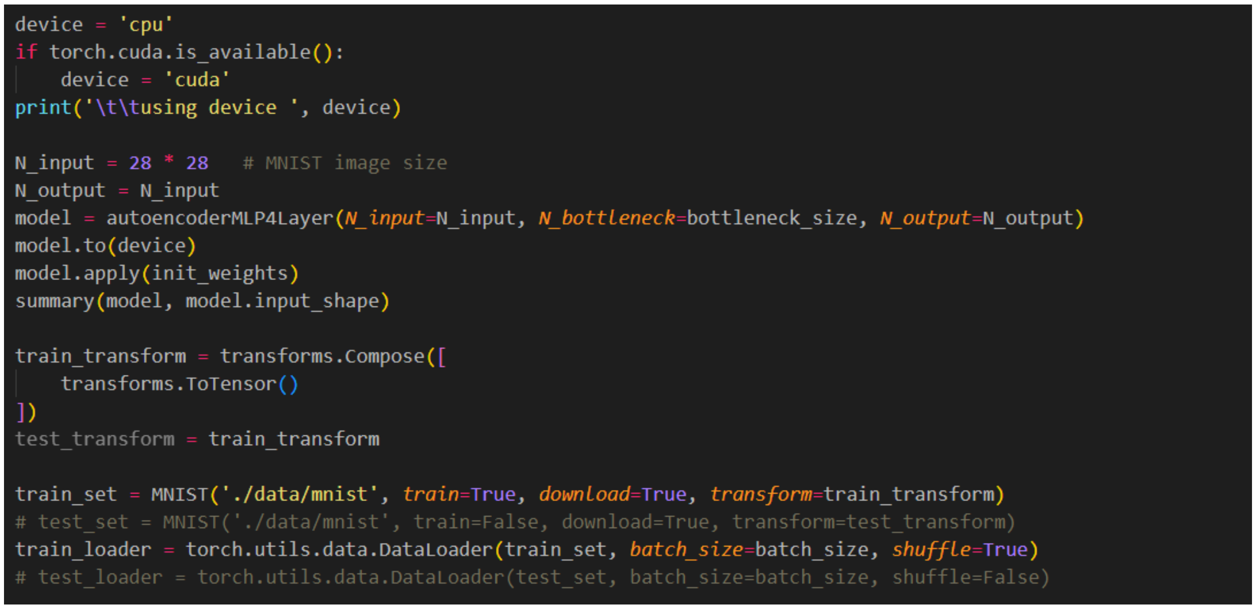
Part 2: Training Details

Main Function

In this lab, the MLP autoencoder is trained using the command "*python train.py -z 8 -e 50 -b 2048 -s MLP.8.pth -p loss.MLP.8.png*," which specifies several important training parameters. The argument -z 8 sets the bottleneck size to 8, meaning the input images will be compressed to 8 neurons in the bottleneck layer before being reconstructed. The -e 50 argument indicates that the model will be trained for 50 epochs, allowing the network to repeatedly learn and adjust its parameters over multiple iterations. The batch size is set to 2048 with -b 2048, meaning that the model processes 2048 images in each batch, which helps the model generalize better by averaging the gradients over a larger set of images. The output model weights are saved to the file MLP.8.pth using the -s argument, and the loss curve, which shows how the error decreases over time, is saved to loss.MLP.8.png with the -p argument.

The training process begins by loading the MNIST dataset using the torchvision.datasets.MNIST class and applying a ToTensor() transformation, which converts the input images from 28x28 pixel grayscale images into torch. Tensor objects with values normalized between 0 and 1. The data is then passed to a DataLoader, which handles batching and shuffling of the data during training. The batch size, specified as 2048, defines how many images are processed in each iteration.

Next, the autoencoder model is initialized with an input size of 784 (the flattened MNIST images), a bottleneck size of 8 (as defined by -z), and an output size of 784 to match the input. The model is then transferred to the appropriate device, either CPU or GPU, depending on availability, ensuring efficient training (see Figure 6). To promote stable and efficient learning, the weights are initialized using the Xavier uniform initialization method, which helps prevent vanishing or exploding gradients during backpropagation (see Figure 7).



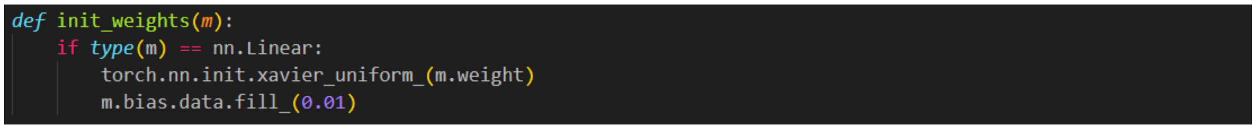
```python
device = 'cpu'
if torch.cuda.is_available():
    device = 'cuda'
print('\t\tusing device ', device)

N_input = 28 * 28   # MNIST image size
N_output = N_input
model = autoencoderMLP4Layer(N_input=N_input, N_bottleneck=bottleneck_size, N_output=N_output)
model.to(device)
model.apply(init_weights)
summary(model, model.input_shape)

train_transform = transforms.Compose([
    transforms.ToTensor()
])
test_transform = train_transform

train_set = MNIST('./data/mnist', train=True, download=True, transform=train_transform)
# test_set = MNIST('./data/mnist', train=False, download=True, transform=test_transform)
train_loader = torch.utils.data.DataLoader(train_set, batch_size=batch_size, shuffle=True)
# test_loader = torch.utils.data.DataLoader(test_set, batch_size=batch_size, shuffle=False)
```

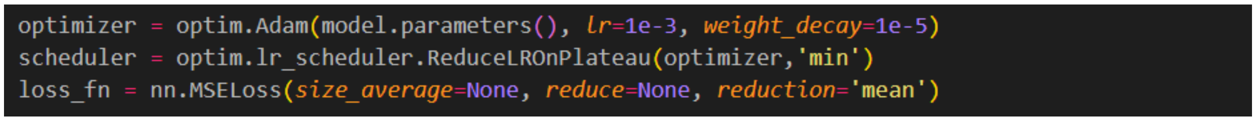
Figure 6: A snippet of the code showing the device and model setup.



```python
def init_weights(m):
    if type(m) == nn.Linear:
        torch.nn.init.xavier_uniform_(m.weight)
        m.bias.data.fill_(0.01)
```

Figure 7: A snippet of the code showing the init_weights() method.

The model is optimized using the Adam optimizer with an initial learning rate of 1e-3 and a weight decay of 1e-5 to regularize the model and prevent overfitting. Additionally, a ReduceLROnPlateau learning rate scheduler is used to dynamically adjust the learning rate when the loss reaches a plateau, ensuring fine-tuning of the model's performance. The loss function used for training is Mean Squared Error (MSE), which compares the reconstructed images to the originals and calculates the error. The code for the optimizer, scheduler, and loss function can be seen in Figure 8.



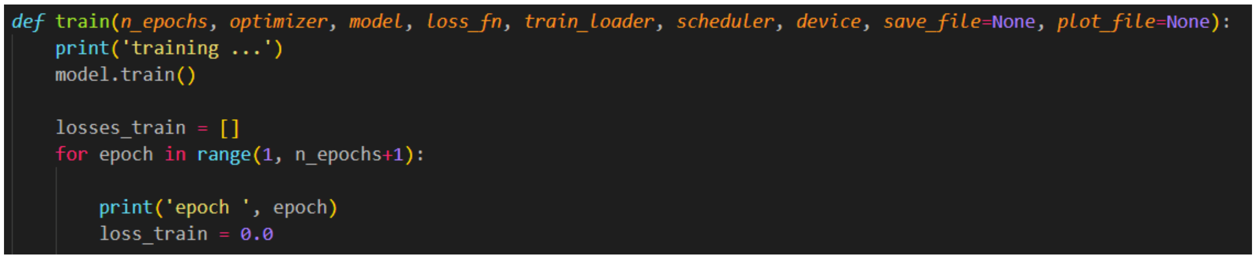
```
optimizer = optim.Adam(model.parameters(), lr=1e-3, weight_decay=1e-5)
scheduler = optim.lr_scheduler.ReduceLROnPlateau(optimizer,'min')
loss_fn = nn.MSELoss(size_average=None, reduce=None, reduction='mean')
```

Figure 8: A snippet of code showing the optimizer, scheduler, and loss function.

Once all components are set up, the train() function is called, handling the entire training process.

Train Function

The train() function manages the core training process of the autoencoder. At the start, the model is set to training mode (see Figure 9) using model.train(). The training proceeds in a loop for the specified number of epochs, and for each epoch, the cumulative training loss (loss_train) is initialized to zero to track the loss across all batches in that epoch.



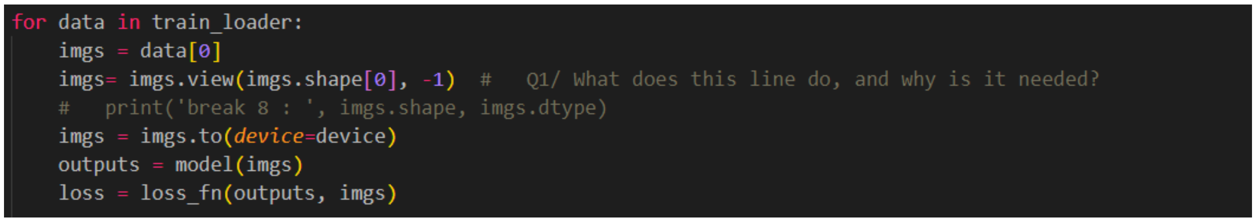
```
def train(n_epochs, optimizer, model, loss_fn, train_loader, scheduler, device, save_file=None, plot_file=None):
    print('training ...')
    model.train()

    losses_train = []
    for epoch in range(1, n_epochs+1):

        print('epoch ', epoch)
        loss_train = 0.0
```

Figure 9: A code snippet showing the model being set to training mode.

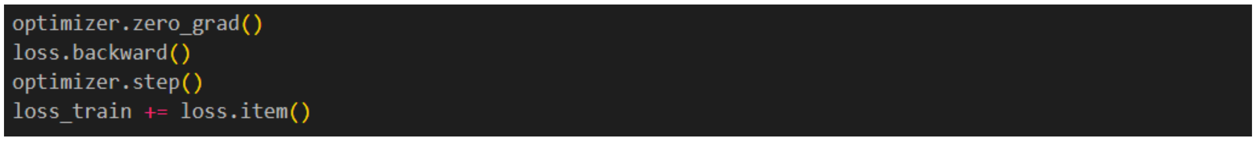
For each mini-batch of images provided by the DataLoader, the images are first reshaped into 1D vectors using imgs.view(imgs.shape[0], -1) to match the input size of the fully connected layers. These reshaped images are then transferred to the appropriate device (CPU or GPU) using imgs.to(device). The model performs a forward pass where the input is compressed by the encoder and then reconstructed by the decoder. The reconstructed output is compared to the original input using the Mean Squared Error (MSE) loss function, which calculates the difference between the original and reconstructed images (see Figure 10).



```
for data in train_loader:
    imgs = data[0]
    imgs= imgs.view(imgs.shape[0], -1)  #   Q1/ What does this line do, and why is it needed?
    #   print('break 8 : ', imgs.shape, imgs.dtype)
    imgs = imgs.to(device=device)
    outputs = model(imgs)
    loss = loss_fn(outputs, imgs)
```

Figure 10: A code snippet showing the model preparation (reshaping), execution (forward pass), and evaluation (loss function).

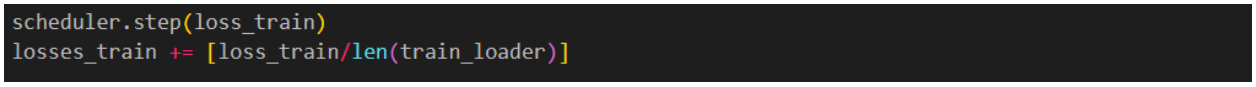
After calculating the loss for the current batch, the gradients are cleared using optimizer.zero_grad() to ensure that previous gradients do not accumulate. The loss is then backpropagated through the network using loss.backward(), which computes the gradients of the loss with respect to the model's parameters. The optimizer, which in this case is Adam, updates the model's weights using these gradients via optimizer.step(). The code for the optimization of the model weights can be seen in Figure 11.



```
optimizer.zero_grad()
loss.backward()
optimizer.step()
loss_train += loss.item()
```

Figure 11: A snippet of code showing how the model uses the Adam to optimize the model weights.

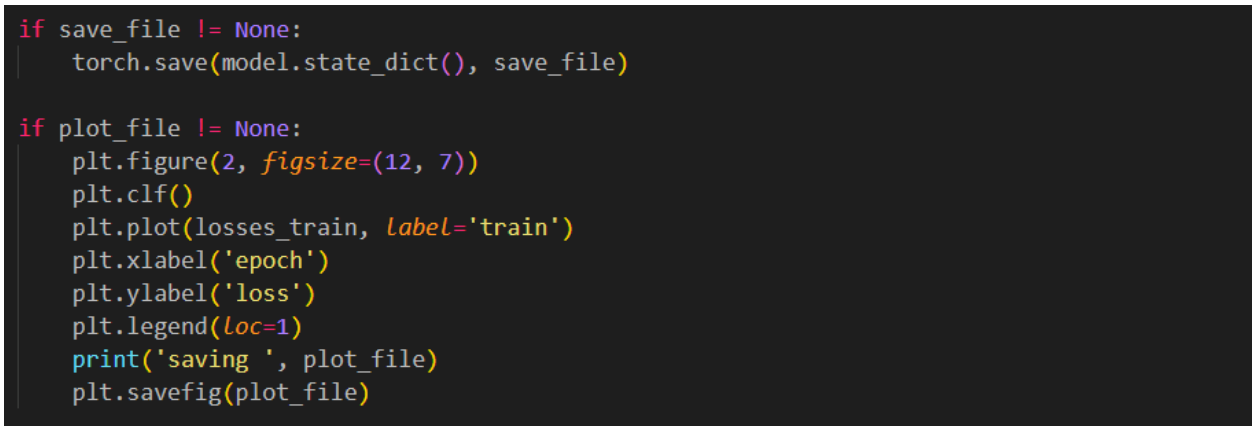
The cumulative training loss is updated by adding the loss for each batch. Once all batches in an epoch are processed, the average training loss is computed by dividing the cumulative loss by the number of batches. The learning rate scheduler, ReduceLROnPlateau, is called with scheduler.step(loss_train) to adjust the learning rate if the loss stagnates, helping the model improve further (see Figure 12).



```
scheduler.step(loss_train)
losses_train += [loss_train/len(train_loader)]
```

Figure 12: The figure shows the learning rate being updated and the average training loss being added to the losses_train list.

At the end of each epoch, the model's weights are saved to the file specified by save_file, allowing the model to be reloaded later for testing or further training. Additionally, the training losses across epochs are saved in a list and used to generate a loss curve, which is plotted and saved to the file specified by plot_file. This curve visually tracks the model's progress over time, showing how the error decreases as the model trains. Throughout the training, the function prints out the current epoch and the average loss to provide real-time feedback on the model's performance (see Figure 13).



```
if save_file != None:
    torch.save(model.state_dict(), save_file)

if plot_file != None:
    plt.figure(2, figsize=(12, 7))
    plt.clf()
    plt.plot(losses_train, label='train')
    plt.xlabel('epoch')
    plt.ylabel('loss')
    plt.legend(loc=1)
    print('saving ', plot_file)
    plt.savefig(plot_file)
```

Figure 13: This code snippet shows the model's weights being saved and a training loss plot being generated and saved if specified.

Part 3: Results

This section outlines the outcomes of the training process and evaluates the model's performance in image reconstruction, image denoising, and bottleneck interpolation.

Step 3: Training Results

The training of the MLP autoencoder over 50 epochs showed a consistent decrease in loss. The model was trained using a batch size of 2048, and the loss function was the Mean Squared Error (MSE). The model was optimized using the Adam optimizer, with a learning rate of 1×10^{-3} and a weight decay of 1×10^{-5}. A ReduceLROnPlateau scheduler was used to dynamically adjust the learning rate when the loss plateaued.

The Loss Curve (see Figure 14) provides a clear visualization of how the loss evolved throughout the training. In the initial epochs, the loss dropped steeply from around 0.1 to approximately 0.03 by epoch 10, indicating that the model quickly learned to capture essential features. As training progressed, the loss decreased more gradually, stabilizing around 0.02 in the final epochs, reflecting the model's fine-tuning of the weights.

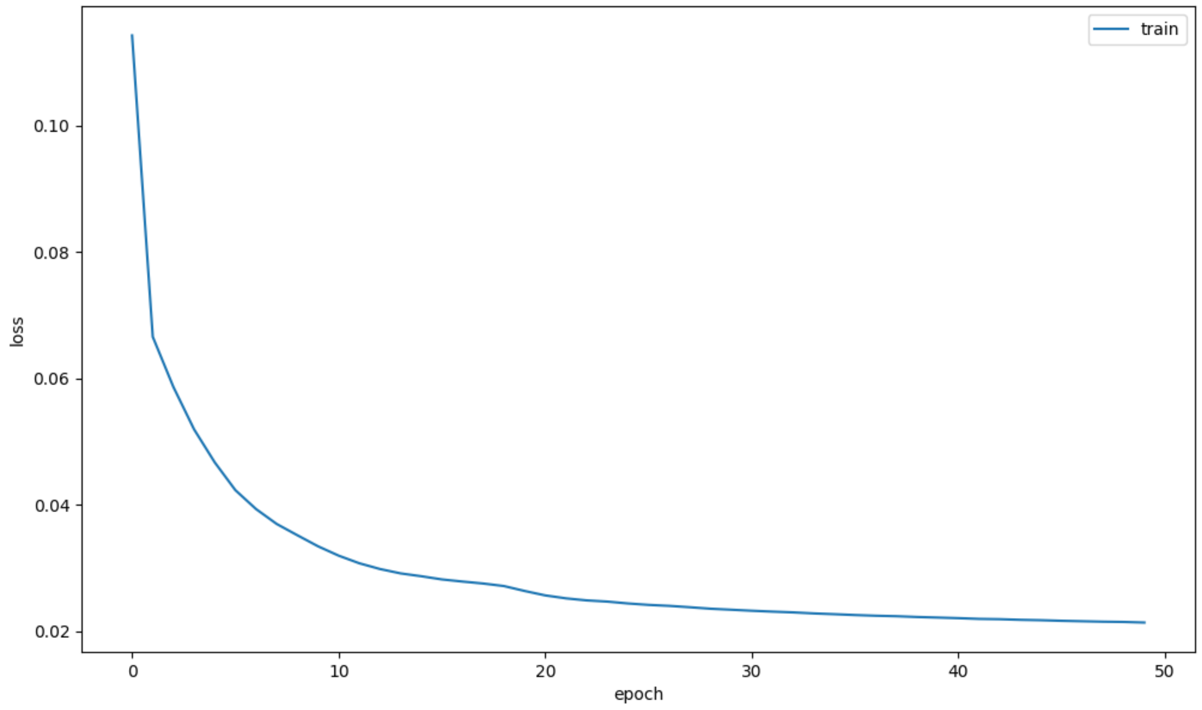
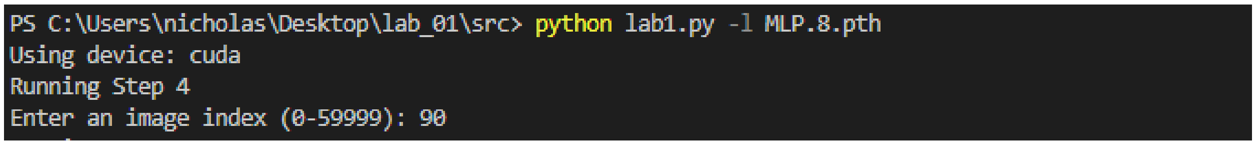


Figure 14: A plot showing how the error decreases as the number of epochs increase.

The smooth loss curve, coupled with the steady decrease in reconstruction error, indicates that the training was successful. There were no significant fluctuations or signs of overfitting, which confirms that the selected hyperparameters and training strategy (e.g., learning rate scheduling and weight decay) were appropriate for this task. By the end of the 50 epochs, the model was well-trained to compress and reconstruct MNIST images with minimal loss.

Step 4: Testing the Autoencoder

In Step 4, the autoencoder's ability to reconstruct images was tested (see Figure 15 and Figure 16). The MNIST images were passed through the trained autoencoder, and the original and reconstructed images were displayed side-by-side using matplotlib. The model successfully recreated the input images with minimal loss, showing that it could compress the original 28x28 pixel images into an 8-dimensional bottleneck and then reconstruct them back to the original format with reasonable accuracy. The results, when visualized, displayed slight pixel blurring, which is expected due to the dimensionality reduction. Overall, the autoencoder functioned as intended, capturing the essential features of the input images.



```
PS C:\Users\nicholas\Desktop\lab_01\src> python lab1.py -l MLP.8.pth
Using device: cuda
Running Step 4
Enter an image index (0-59999): 90
```

Figure 15: The input for the test case shown in Figure 16.

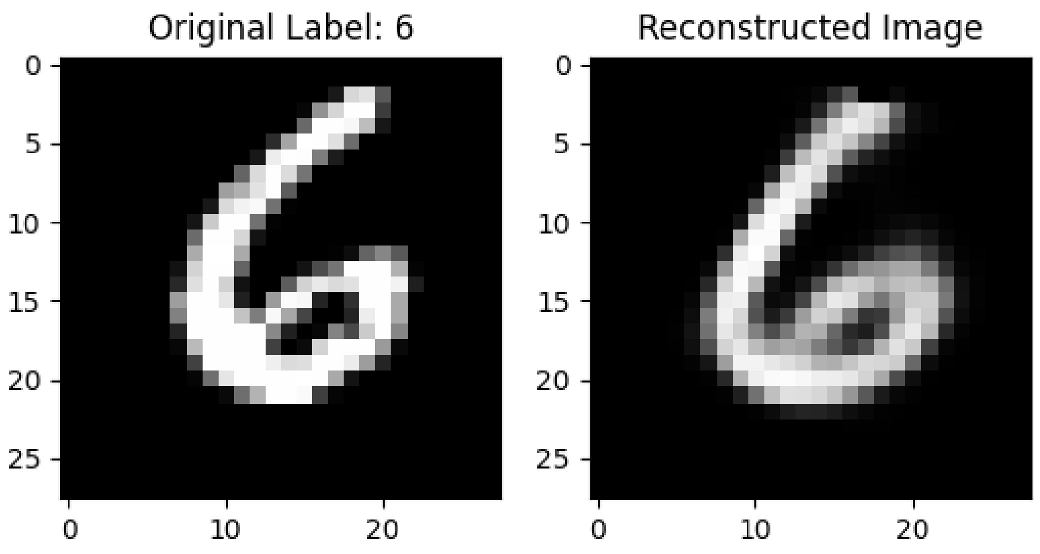


Figure 16: The original image and the reconstructed image for the index 90.

Step 5: Image Denoising

Step 5 tested the autoencoder's ability to remove noise from images. At first, the model struggled to denoise the images, producing poor results where the noisy images were barely improved, and the reconstructions were highly distorted (see Figure 17). This issue was due to the way noise was being added, which made it harder for the autoencoder to learn the correct patterns for denoising. The approach was then revised by using Gaussian noise with a controlled level of variation, applied as noisy_img = img + (0.1**0.5) * torch.randn(28, 28). This method worked better than the original approach of adding random values with torch.rand(), which generated unbalanced noise that brightened the images too much and led to more distortion. The Gaussian noise, on the other hand, introduced both positive and negative noise in a balanced way, allowing the model to more easily distinguish between real image features and noise. By controlling the noise level with (0.1**0.5), the autoencoder was able to filter out the noise more effectively, resulting in denoised images that closely resembled the originals (see Figure 18). This experiment showed that the autoencoder could not only reconstruct clean images but also learn to remove noise from corrupted inputs.

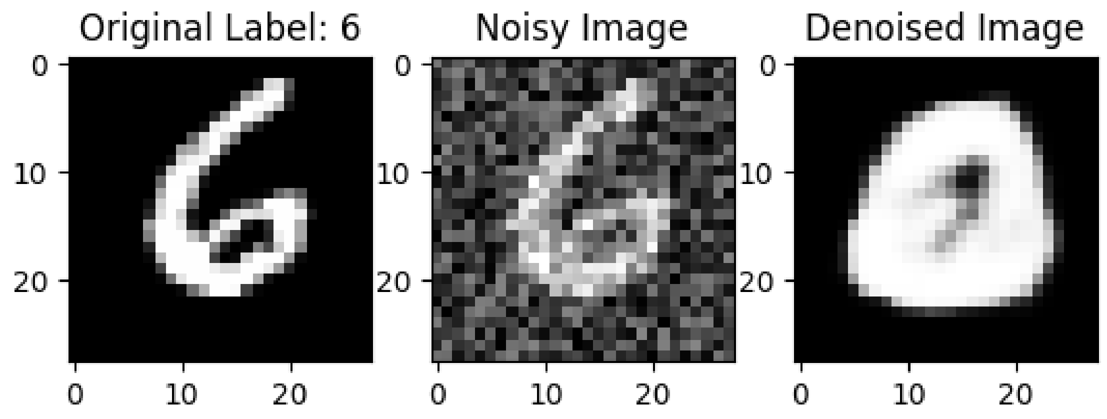


Figure 17: Initial attempt at denoising with poor results, producing a distorted denoised image.

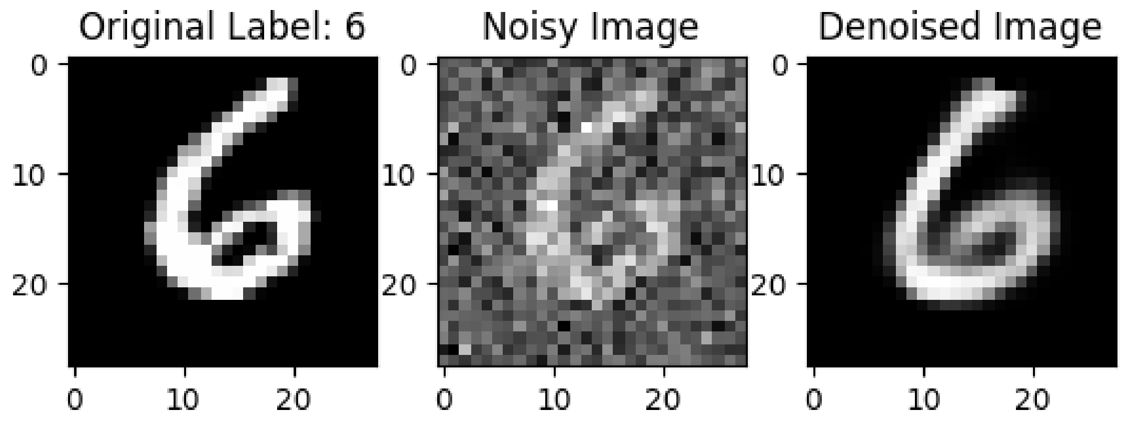


Figure 18: Revised denoising method with improved results, closely resembling the original image.

Step 6: Bottleneck Interpolation

In Step 6, the latent space representation was explored by performing bottleneck interpolation. Two images were encoded into their bottleneck representations, and intermediate latent representations were generated through linear interpolation between the two bottleneck tensors. These interpolated representations were then passed through the decoder to visualize a smooth transformation between the two images (see Figure 19).

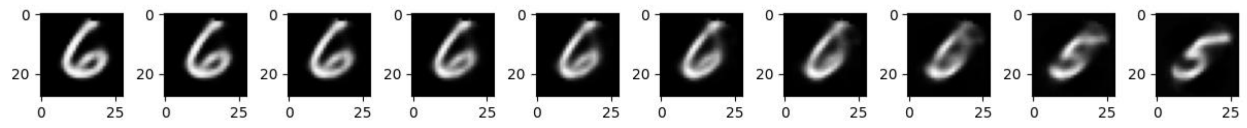


Figure 19: Visualization of bottleneck interpolation showing a smooth transformation between two images, gradually morphing from a '6' to a '5' as intermediate representations are decoded.

This process demonstrated how the compressed representations captured key features of the input images, with the interpolation producing a gradual morphing effect from one image to another. The successful implementation of bottleneck interpolation highlights the potential of the autoencoder for image generation and shows how the model captures meaningful relationships in the data at a compressed level.

Integration into lab1.py

The final implementation was integrated into lab1.py, which combines the steps of loading the model, testing the reconstruction, denoising images, and performing bottleneck interpolation. The model is run and tested using the command line:

```
python lab1.py -l MLP.8.pth
```

Each step can be executed by invoking the corresponding methods, and the results are visualized using matplotlib. The overall flow worked seamlessly after addressing the initial issues with noise addition, allowing for effective reconstruction, denoising, and interpolation.