ELEC 475 LAB 1

September 13, 2025

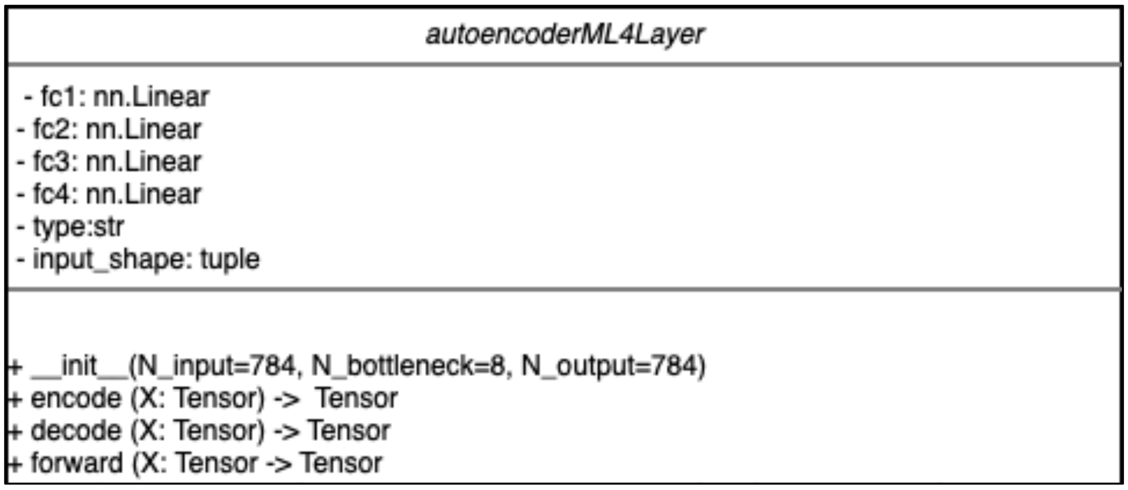
Nathan Duncan
20ntd1@queensu.ca
20283699

Daniel Dubinko
19dd34@queensu.ca
20229482

Model Details

The implemented MLP autoencoder model uses the same structure as discussed in lecture. Our model inherits from torch.nn.model, a pytorch base class for neural networks. The inputs are MNIST 28x28 pixel grayscale images of handwritten digits.

Model Class UML



autoencoderML4Layer

- fc1: nn.Linear
- fc2: nn.Linear
- fc3: nn.Linear
- fc4: nn.Linear
- type:str
- input_shape: tuple

+ __init__(N_input=784, N_bottleneck=8, N_output=784)
+ encode (X: Tensor) -> Tensor
+ decode (X: Tensor) -> Tensor
+ forward (X: Tensor -> Tensor

Class Definition

Subclass nn.Module

Initialize the Hidden Layer size:

- N2 = 392 (784/2)

Initialize four fully connected layers:

- fc1: input layer → hidden layer (784 → 392)
- fc2: hidden layer → bottleneck (392 → 8)
- fc3: bottleneck → hidden layer (8 → 392)
- fc4: hidden layer → output layer (392 → 784)

Store metadata attributes:

- type = 'MLP4'
- input_shape = (1, 28*28)

Encoder Method

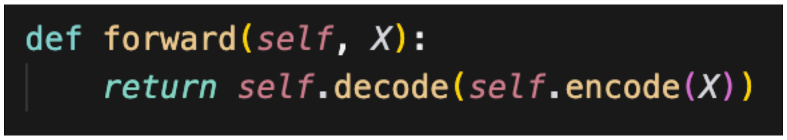
- Forward pass through fc1 -> ReLU -> fc2 -> ReLU
- The output is the compressed bottleneck representation

Decoder Method

- Forward pass through fc3 -> ReLU -> fc4 -> Sigmoid
- The output is the decoded 1x784 Tensor

Forward Pass Method

- Call encode() as an input to decode()
- Training/inferencing can be done with model(X). X being a 1x784 tensor



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

Graphical Representation

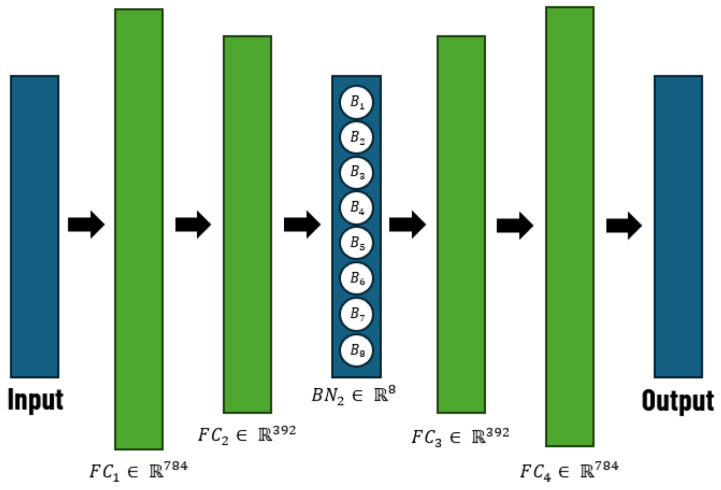

Input
B_1
B_2
B_3
B_4
B_5
B_6
B_7
B_8
$FC_1 \in \mathbb{R}^{784}$
$FC_2 \in \mathbb{R}^{392}$
$BN_2 \in \mathbb{R}^{8}$
$FC_3 \in \mathbb{R}^{392}$
$FC_4 \in \mathbb{R}^{784}$
Output

Figure 1: Graphical representation of the MLP autoencoder architecture.

Train Method

We used the provided training setup in "train.py" to train our model discussed in the previous section with a 1x8 bottleneck vector. The train() method trains an inputted model, the autoencoder model in this case, using a given dataset. The train() method updates the model weights and creates a file to store the said model weights.

Inputs

n_epochs: the number of the times the training will go through all the data = 50 Epochs

optimizer: used to update the model weights based on gradients = Adam

Learning rate: a scalar influencing the optimizer step size, 1e-3

Weight decay: a regularization scalar to limit the growth of network weights, 1e-5

model: model used for training = autoencoderMLP4Layer

loss_fn: the loss (or cost) function that measures how far predictions are from their ground truth labels. We used a mean squared error function from torch.nn.MSELoss with mean reduction of the output.

train_loader: iterator that provies mini-batches of training data, and has an input to shuffle the batches = Torch DataLoader

scheduler: adjusts the learning rate during training = ReduceLROnPlateau, 'min'

device: device on the local system that is used to train, such as CPU or GPU = 'cpu'

save_file: file path where trained model weights are saved (usually .pth) = "MLP.8.pth"

plot_file: file to plot training plots = "loss.MLP.8.png"

Running the Train Method

To run the training method, train.py, use command,

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

-z (bottleneck size), -e (epochs), -b (batch size), -s (model weight path), -p (loss plot)

The train method has the following steps:

- The first epoch is ran. The first batch is inputted.
- The training function calculates the output from the forward function of the model.
- Then, runs the loss function to compare the reconstructed output with the original input.
- Next, runs the back propagation by first running the zero_grad() to clear old gradients, then the loss.backward() to compute the gradients of the weights with respect to the loss, and finally updates the weights with optimizer.step().
- Before moving on to the next batch. The loss is added to the epoch loss total
- Once the epoch has finished, the final steps are to run a scheduler to see if the training loss is improving, if it is not the learning rate is reduced to make finer adjustments.

Results

The training loss per epoch shown in Figure 2 shows the loss decreased exponentially during the first ~20 epochs. At this point it begins nearing a plateau and the schedular likely begins reducing the learning rate for finer steps in tuning the model weights. The loss decreases marginally over the next 30 epochs to a final loss of about 0.02.

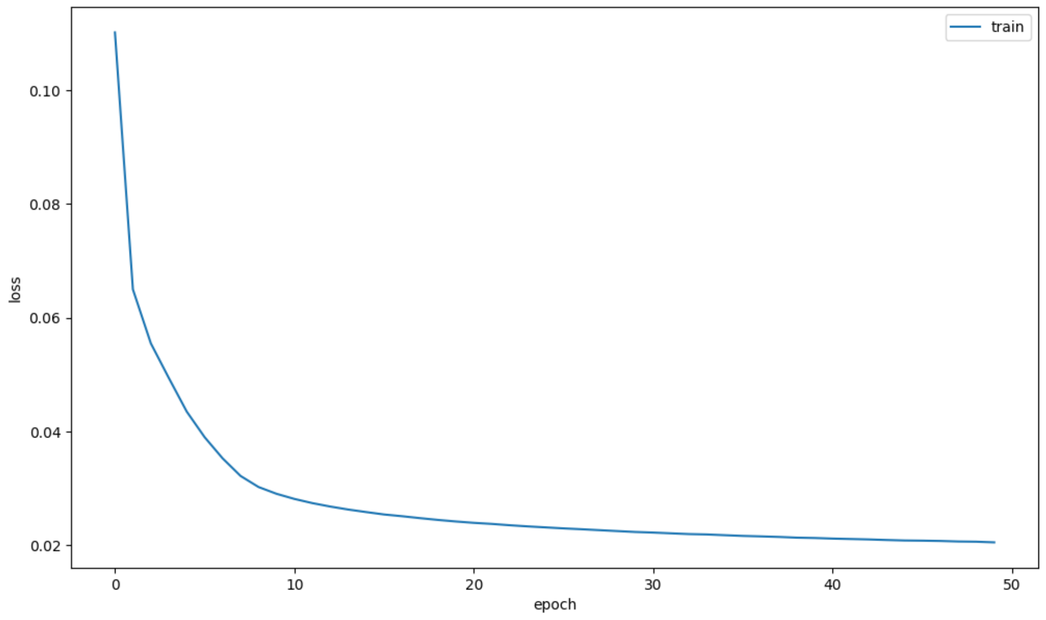


Figure 2: Loss vs training epochs of the MNIST dataset autoencoder.

Although we did not create any quantitative post-training evaluation methods of our own, qualitatively reconstructions of the autoencoder match many features of the input (i.e. graphics we visually determine to be a certain number, generally produce a reconstruction with is also visually recognizable as the same number). The biggest shortcoming of our model was in denoising images, which often produced illegible output images.

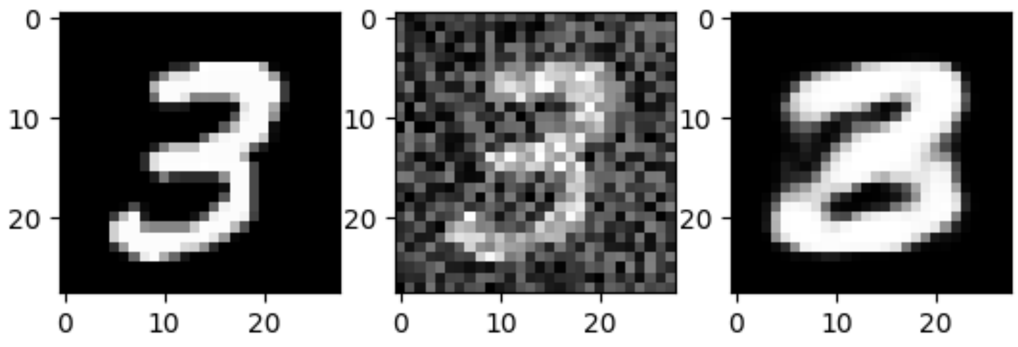


Figure 3 - Chosen Image with applied noise and then predicted by model

With the added noise the model struggles to predict the image.

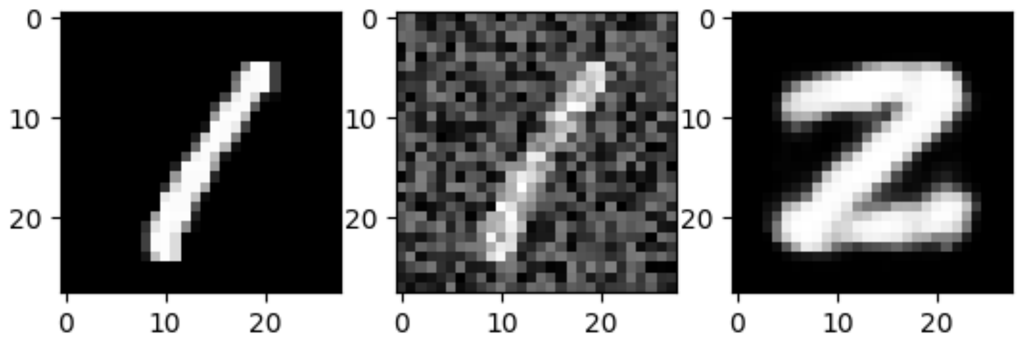


Figure 4 - Chosen Image with applied noise and incorrect prediction

Here the model predicts incorrectly. The reason for the mistake is the added noise. One option to improve denoising is to edit the size of the bottleneck reducing it further or possibly extending it slightly. Although a 1x8 is already a great compression from the original 1x784 input this is still one of the most promising avenues for improvement. Additionally, we could experiment with tuning of the training hyperparameters like number of epochs and select other optimizers for improved performance.

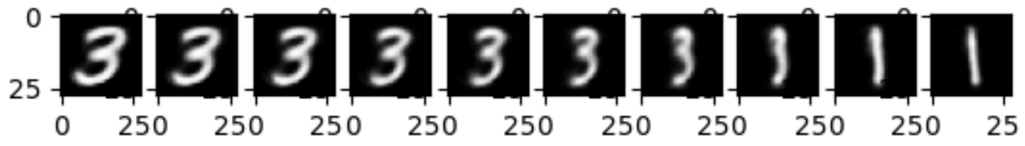


Figure 5 - bottleneck interpolation from 3 to 1

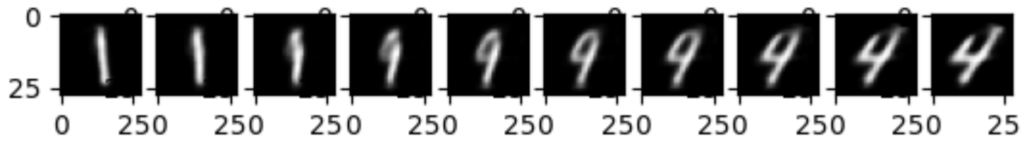


Figure 6 - bottleneck interpolation from 1 to 4

Here, the two bottleneck layers are interpolated. This demonstrates that even without the original image, it's still possible to decode the key information and reconstruct the image back to its original form. Furthermore, this shows that a bottleneck or any other latent space, when properly decoded, can be altered to achieve a desired result.