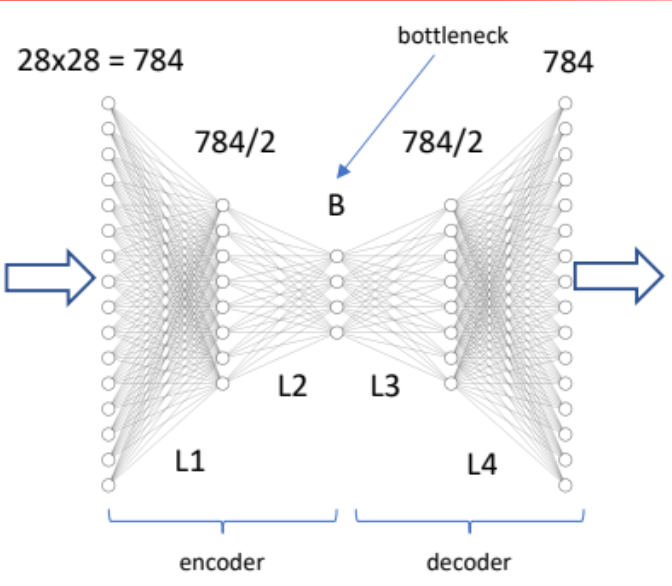
ELEC 475 Lab 1

Luka Gobovic 20215231

Ryan Anderson 20151125

# Model Details

The model being used is a multi-layer perceptron autoencoder as seen in the early lectures. The model consists of 4-layers which are fully interconnected between each other. The first half of the model is encoding, the final half of the model is encoding. The first layer accepts the input image (784 pixels in a column vector from a 28x28 pixel image) and the output is half of the input (392 pixels). The 2nd layer feeds into the smallest section of the model (the bottleneck) which consists of 8 pixels. Each encoding layer uses a ReLU activation function applied after the weights applied to the input function. This concludes the encoding section of the model. The decoding model is opposite to the encoding section, where instead of scaling down, we scale the bottle neck up to 392 pixels in the third layer, and 784 pixels in the fourth and final layer. The third layer uses another ReLU activation function and the final layer’s activation function is a sigmoid.



# Training Details

The model was trained on MNIST data of handwritten numbers. A total of 2048 random images were used from the dataset for training purposes. The training images are put into a “loader”, which send the images through the model. The model then determines the loss from the output images using a loss function which measures how well the model can recreate the handwritten number. As the images are fed through the model, the weights of layer are updated to be optimized in a way that will reduce the gradient. That is to say that the training optimizes the weights to ensure that the loss function is reduced as much as possible. A lower loss function correlates to better model performance (assuming there is no overfitting) which results in the autoencoder being able to better recreate and recognize the handwritten numbers.

The optimizer method used for this lab was Adam. Adam stands for adaptive moment estimation, which is another type of gradient descent. The main difference from SGD and why I was used for this lab was that it adapts the learning rate and maintains a separate value for each parameter, which also accelerating convergence and containing fewer extra hyperparameters. Faster convergence meant that the training process was quicker. The initial learning rate was set to 0.001, as per the example in the slides, with the learning rate schedule reduced on plateau. This allows the learning rate to reduce when a metric has stopped improving. The model often benefits from reducing the learning rate once the learning begins to stagnate.

# Results

## Autoencoder Test

To test the results of our model we can feed images in and observe the difference between the input/output pairs. The test involves doing a forward inference using the weights generated during training. As we can see from Figure 1**,** it appears our model can recreate the handwritten number, but Figure 2 shows some of the difficulties that our model has. It’s possible that the poor result seen in Figure 2 is due to our randomly chosen training data not including certain number styles (ex a 2 with a flat tail inFigure 2vs a 2 with a spiral tail inFigure 3) resulting in our model struggling to reproduce the correct output.

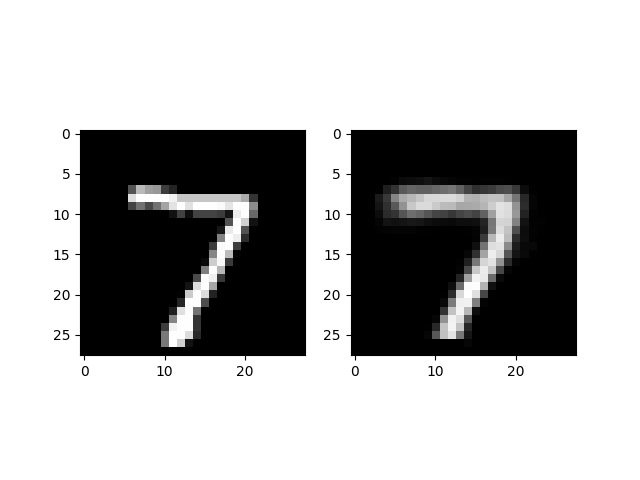


Figure 1: Input/output pair of a 7.

## 

Figure 2: Input/output pair of a 2.

## Image Denoising

To further test the model, gaussian noise was added to the input and we observed how the model performed. The results of this test appear to show our model is able to properly filter out the noise and is able to recreate the appropriate handwritten number with reasonable success. Depending on the input we may observe problems in some cases as seen in Figure 3 in which we may be able to recreate a number, but it does not match the input value. The outputs of this test appear to be blurrier and less defined which show some form of uncertainty.

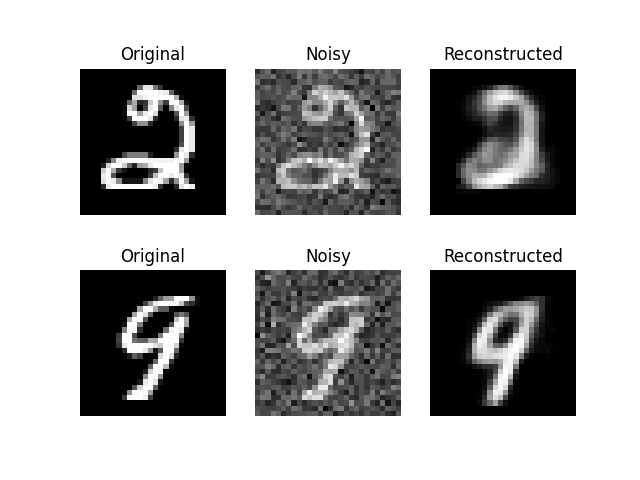


Figure 3: The original data showing added Gaussian noise, which is then fed to our model to create the reconstructed output.

## Bottleneck Interpolation

Using the encode portion of our model we can create a bottleneck tensor which represents a compressed representation of the input in a lower-dimensional space. We find 2 bottleneck tensors from different images. From these encoded bottleneck tensors, we can generate some intermediate tensors by interpolating between them. Feeding all of the resulting bottleneck tensors through the decode method results in seeing the final output image of each bottleneck tensor. Our results should then show one number morphing into another which occurs due to the linear interpolation between the 2 original image bottleneck tensors.

In Figure 4 we can see the interpolation between the following sets of numbers: 6 to 5, 8 to 2, and 7 to 9. The intermediate interpolation images we created helps visualize and understand feature variations and how our model has learned to generate different features by recognizing specific characteristics from an image.

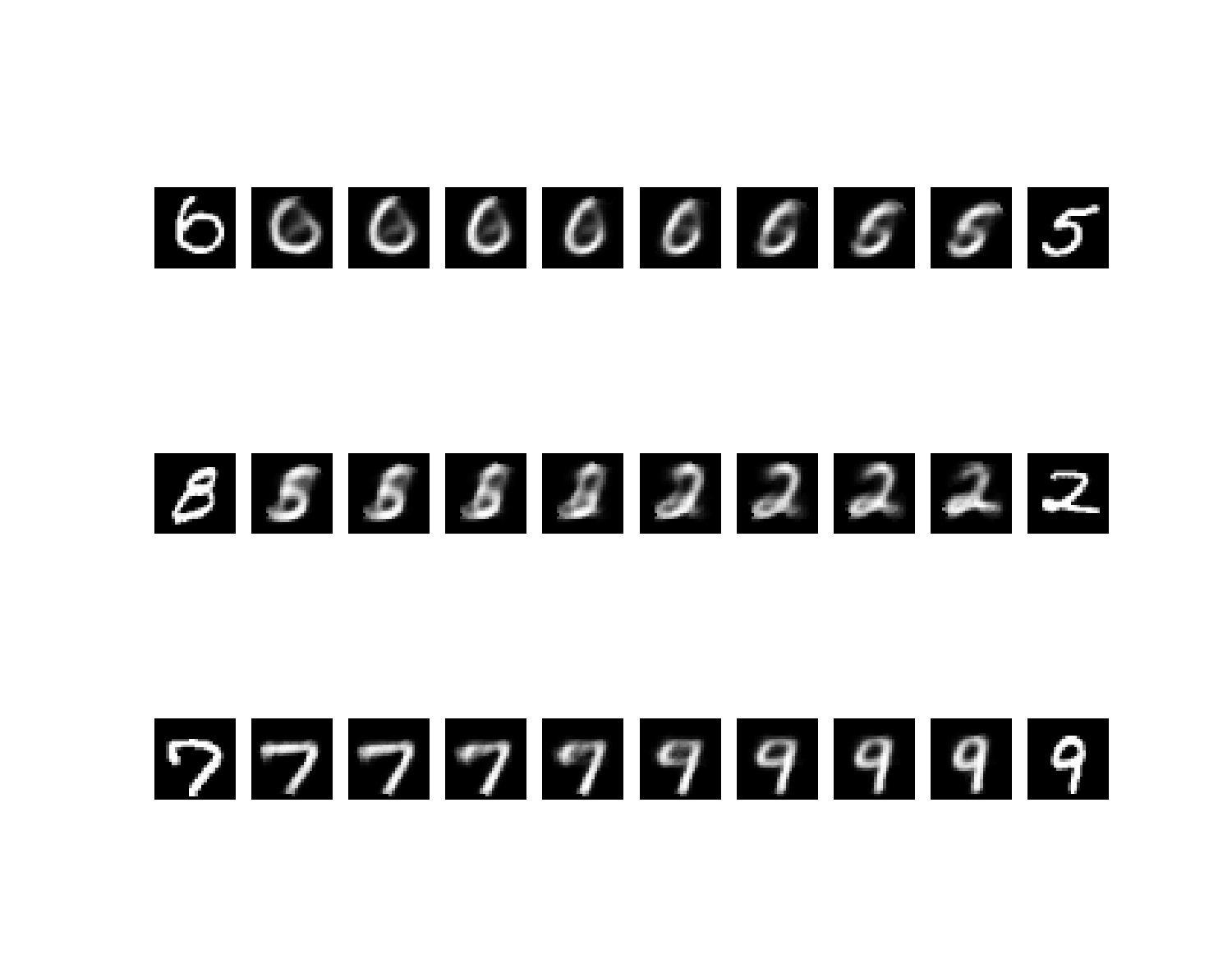


Figure : 3 sets of bottleneck interpolation in which our model interpolates between different numbers.

## Loss Curve:

A graph with a line

Description automatically generated

Figure : Loss Curve Plot for the training

Figure 5 shows the loss curve plot for the training loss. The above curve shows a steep drop in loss, followed by a levelling out. This indicates that the training session is behaving well, and that the model is not over or underfit. This means that the model is making relatively good evaluations.