1. About

This project is designed to guide you through the process of implementing and training a Multi-Layer Perceptron (MLP) autoencoder using PyTorch. The project covers everything from setting up your development environment, loading and visualizing the MNIST dataset, building and training an MLP autoencoder, to testing its performance on image reconstruction and denoising tasks. Additionally, you will explore bottleneck interpolation to generate new images by linearly interpolating between encoded representations of two images.

2. Environment Setup

Before diving into the coding tasks, ensure that your development environment is properly configured. If you haven't already, install Visual Studio Code (VSCode) or PyCharm as your Integrated Development Environment (IDE). Once installed, use the terminal in your IDE to install the necessary Python packages:

```
pip install numpy

pip install torch

pip install matplotlib

pip install torch-summary

pip install matplotlib
```

You can easily install any missing packages later in the same way via the terminal.

3. Visualizing the MNIST Dataset

In this section, you will write a Python program that loads the MNIST dataset, prompts the user for an index value between 0 and 59999, and displays the corresponding image along with its label. The dataset can be loaded using PyTorch's torchvision.datasets.MNIST class.

The syntax to read MNIST is:

```
train_transform = transforms.Compose([transforms.ToTensor()])
train_set = MNIST('./data/mnist', train=True, download=True,
                  transform=train_transform)
```

This will download the training partition of the dataset to your local directory **./data/mnist**. Once instantiated, you can access the images through the `train_set.data` field, and the corresponding image labels through the `train_set.targets` field.

The image at index `idx` can be displayed using the syntax:

```
plt.imshow(train_set.data[idx], cmap='gray')

plt.show()
```

4. Implementing and Training an MLP Autoencoder

Implement a 4 layer MLP autoencoder, by completing the **`_init_()`** and **`forward()`** methods in the **`model.py`** module. Train your model by invoking the **`train.py`** module with the following command line arguments in the PyCharm terminal:

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

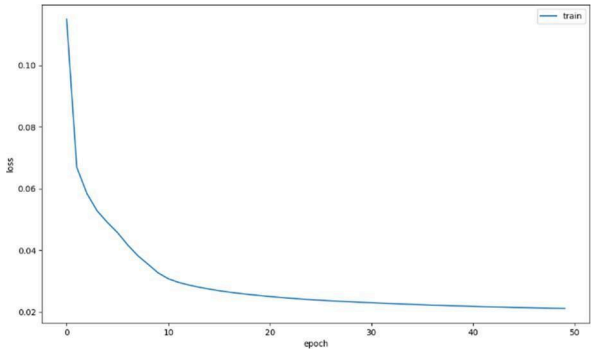
Hint 1: The call to **`torchsummary`** should return something like this:

```
----------------------------------------------------------------
        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
----------------------------------------------------------------
```

Hint 2: After each of the two fully connected layers in the encoder, and after the first fully connected layer in the decoder, apply a ReLU activation. After the second fully connected layer in the decoder, apply a sigmoid activation.

Hint 3: Use the **`loss.MLP.8.png`** plot to help you determine whether your system is training properly. A well-behaved training session should yield a loss curve that looks something like this:



Hint 4: Training on a cpu device for 50 epochs should take ~10 mins. Training on a gpu device will be faster.

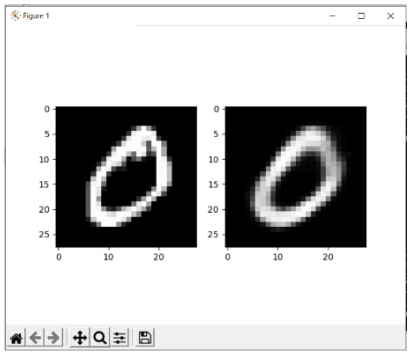
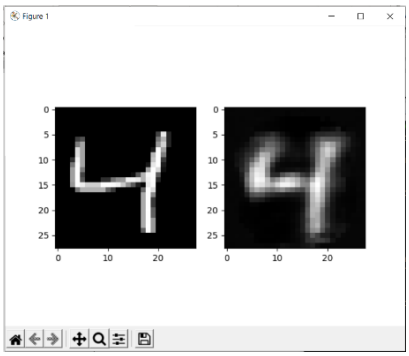
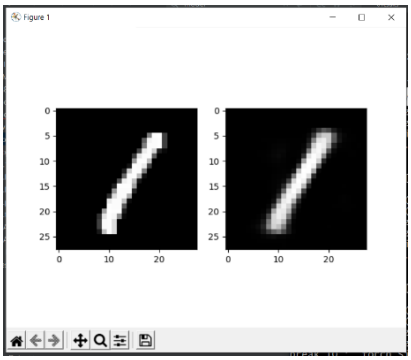
5. Testing Your Autoencoder

Once your autoencoder is trained, modify your visualization program from Step 2 to display both original and reconstructed images side-by-side. Ensure that:

- The input image is normalized between 0 and 1.
- The model is set to evaluation mode (model.eval()), and gradient calculations are disabled.

First, instantiate a version of your model, with the **`MLP.8.pth`** network weights that you generated during training. Then, pass each indexed image as input to the model, and display both the input and the output images side-by-side.

Use matplotlib to render images. Some example input/output pairs should look something like this:

Hint 5: Before you pass an image as an argument to the model, make sure that it is a 1x784 dimensional tensor of type **`torch.float32`**. Also make sure that its values are normalized to fall between 0 and 1.

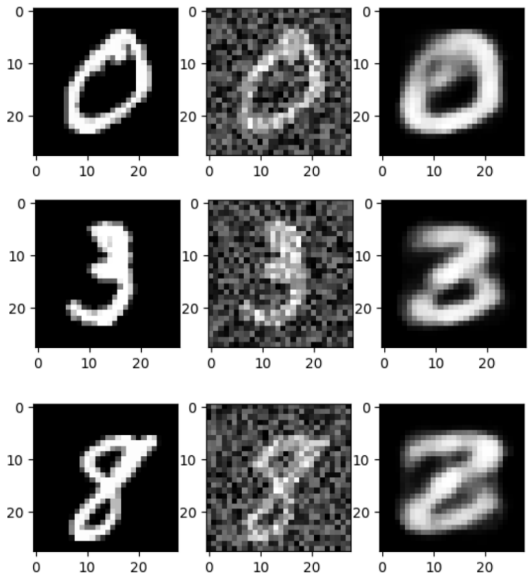
Hint 6: Before invoking your model (i.e. calling a forward inference), make sure that it is in **`eval`** mode, and that you disable gradient calculations.

Hint 7: Use **`matplotlib`** to render the images side-by-side, as follows:

```
f = plt.figure()
f.add_subplot(1,2,1)
plt.imshow(img, cmap='gray')
f.add_subplot(1,2,2)
plt.imshow(output, cmap='gray')
plt.show()
```

6. Image Denoising

Autoencoders can be used to remove noise from an image because they are able (and forced) to capture the general features of an image, as they can only represent it that way. Test your autoencoder's ability to remove image noise, by repeating the test in Step 4 with noise added to each image. The output should look something like this:



Hint 8: Uniform noise can be generated using the pytorch `torch.rand()` method.

7. Bottleneck Interpolation

The bottleneck is the layer where the input data is compressed into a lower-dimensional space (latent space). This layer is the opposite of what a sparse autoencoder would have, where the bottleneck compresses the dimensionality, rather than expanding it. This compression forces the model to learn essential features while discarding irrelevant information.

- Dimensionality Reduction: Reduces the input size, capturing only the most important features.
- Information Compression: Prevents overfitting by eliminating redundant data and focusing on patterns.
- Feature Extraction: Acts as an automatic feature extractor, learning abstract representations of the input.
- Reconstruction: The decoder uses this compressed representation to reconstruct the original input as accurately as possible.

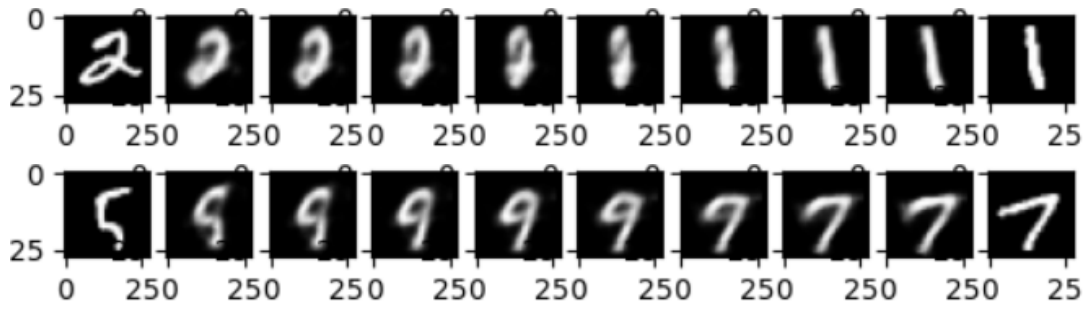
Applications:

- Denoising: Removes noise from data by learning clean representations.
- Latent Space Interpolation: Generates new data by interpolating between encoded representations.

Add two new methods to your model class, called **encode** and **decode**. The **encode** method takes the same input as the existing **forward** method (i.e. the flattened image tensor), and returns the bottleneck tensor. The **decode** method takes as input the bottleneck tensor, and returns the reconstructed image. These two new methods are really just the first and second half of the existing **forward** method, which could then be rewritten in the following way:

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

Next, create a module that passes two images through the **encode** method, returning their two bottleneck tensors. Then, linearly interpolate through these two tensors for **n** steps, creating a set of **n** new bottleneck tensors. Pass each of these new tensors through the **decode** method, and plot the results. Some examples of linear interpolations of 8 steps between two images are as follows.



8. Results

The submission should include a brief description of how well the system worked. Was it as expected, or were there some difficulties and surprises? Include the loss curve plot in this section, and specifically comment on its behavior.