Experiment #	<TO BE FILLED BY STUDENT>	Student ID	<TO BE FILLED BY STUDENT>
Date	<TO BE FILLED BY STUDENT>	Student Name	[@KLWKS_bot] THANOS

Experiment 12: Building a Simple Autoencoder on MNIST Data using Pytorch.

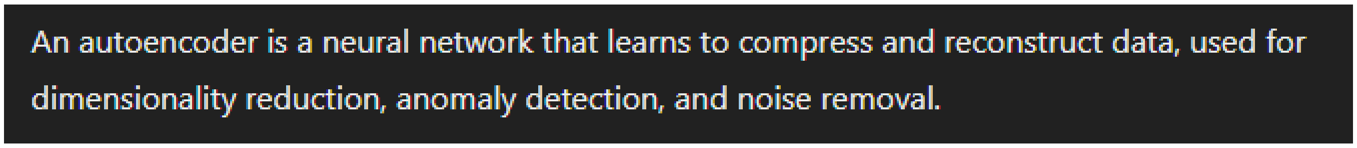
Aim/Objective: To implement a simple autoencoder using the Pytorch library on the MNIST dataset for image compression and reconstruction.

Description: Autoencoders are neural network models designed for unsupervised learning. They aim to learn efficient representations of data, typically for the purpose of dimensionality reduction or feature learning. In this experiment, we'll use the popular MNIST dataset of handwritten digits to build a simple autoencoder using the Pytorch library.

Pre-Requisites: Basic knowledge of neural networks and deep learning concepts, PyTorch Basics, Python Programming, Familiarity with the library.

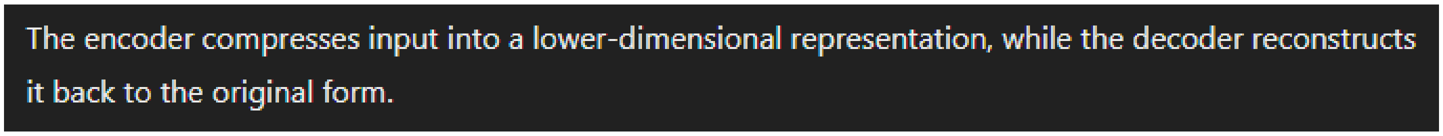
Pre-Lab:

1. What is an autoencoder, and what is its primary purpose in neural network architecture?



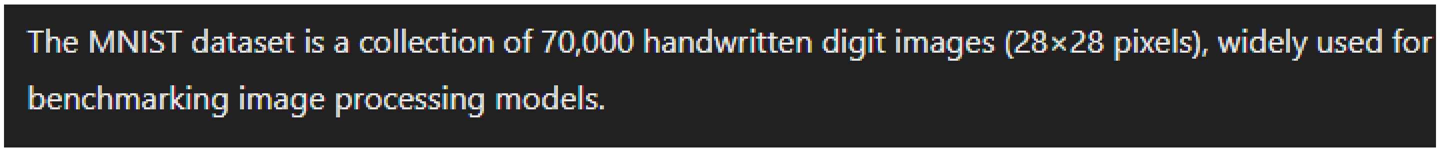
An autoencoder is a neural network that learns to compress and reconstruct data, used for dimensionality reduction, anomaly detection, and noise removal.

2. Explain the concept of encoder and decoder in the context of an autoencoder.



The encoder compresses input into a lower-dimensional representation, while the decoder reconstructs it back to the original form.

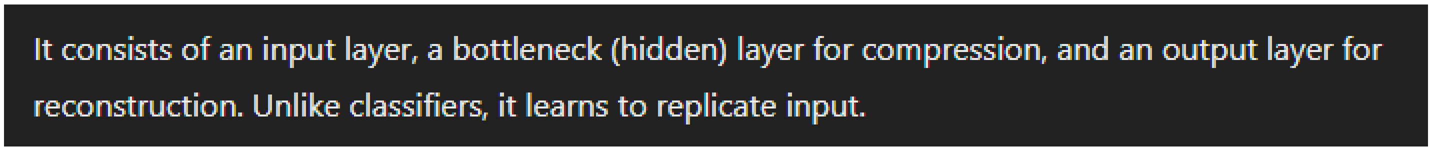
3. What is the MNIST dataset, and why is it commonly used in image processing tasks?



The MNIST dataset is a collection of 70,000 handwritten digit images (28×28 pixels), widely used for benchmarking image processing models.

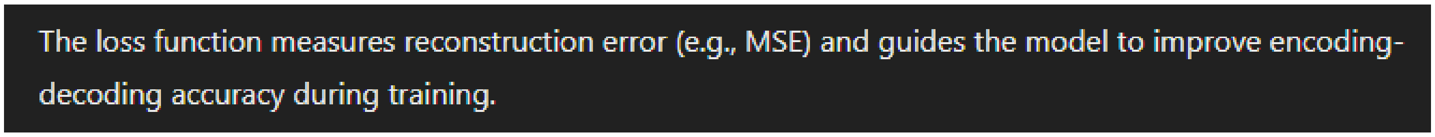
Course Title	DEEP LEARNING	ACADEMIC YEAR: 2024-25
Course Code(s)	23AD2205R/A	Page **1**

4. Describe the architecture of a simple autoencoder. How does it differ from other neural network architectures?



It consists of an input layer, a bottleneck (hidden) layer for compression, and an output layer for reconstruction. Unlike classifiers, it learns to replicate input.

5. How does the loss function in an autoencoder contribute to the model's training?



The loss function measures reconstruction error (e.g., MSE) and guides the model to improve encoding-decoding accuracy during training.

In-Lab:

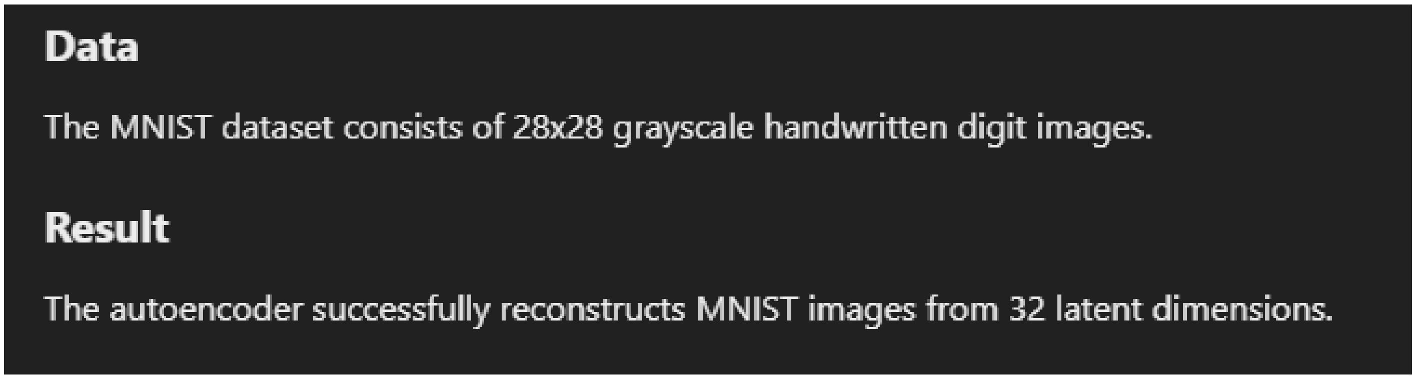
Program 1: Build a simple auto encoder on MNIST data using Pytorch library and help them in building a simple Auto Encoder. (note: encode the image into from 784 to 32 pixels).

Procedure/Program:

```
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision.transforms as transforms
import torchvision.datasets as datasets
from torch.utils.data import DataLoader
import matplotlib.pyplot as plt

class Autoencoder(nn.Module):
    def __init__(self):
        super().__init__()
        self.encoder = nn.Sequential(
            nn.Linear(784, 128), nn.ReLU(),
            nn.Linear(128, 64), nn.ReLU(),
            nn.Linear(64, 32), nn.ReLU()
        )
        self.decoder = nn.Sequential(
            nn.Linear(32, 64), nn.ReLU(),
            nn.Linear(64, 128), nn.ReLU(),
            nn.Linear(128, 784), nn.Sigmoid()
        )

    def forward(self, x):
        return self.decoder(self.encoder(x))

transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Lambda(lambda x: x.view(-1))
])
```

```
train_loader = DataLoader(
    datasets.MNIST(root="./data", train=True, transform=transform, download=True),
    batch_size=64, shuffle=True
)

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = Autoencoder().to(device)
criterion = nn.MSELoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

for epoch in range(5):
    for images, _ in train_loader:
        images = images.to(device)

        optimizer.zero_grad()
        loss = criterion(model(images), images)
        loss.backward()
        optimizer.step()

    print(f"Epoch [{epoch+1}/5], Loss: {loss.item():.4f}")

sample, _ = next(iter(train_loader))
with torch.no_grad():
    reconstructed = model(sample.to(device))

fig, ax = plt.subplots(2, 10, figsize=(10, 3))
for i in range(10):
    ax[0, i].imshow(sample[i].view(28, 28), cmap="gray")
    ax[1, i].imshow(reconstructed[i].cpu().view(28, 28), cmap="gray")
    [a.axis("off") for a in ax[:, i]]

plt.show()
```

- **Data and Results:**



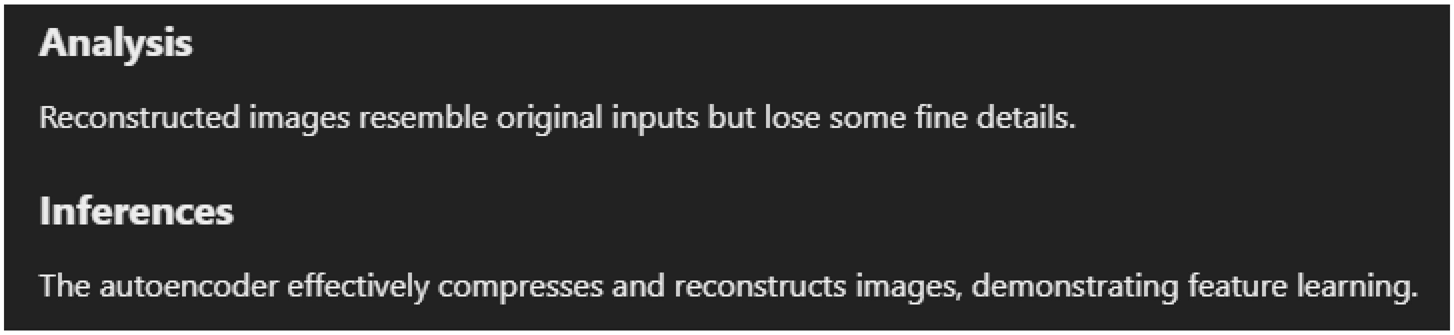
Data

The MNIST dataset consists of 28x28 grayscale handwritten digit images.

Result

The autoencoder successfully reconstructs MNIST images from 32 latent dimensions.

- **Analysis and Inferences:**



Analysis

Reconstructed images resemble original inputs but lose some fine details.

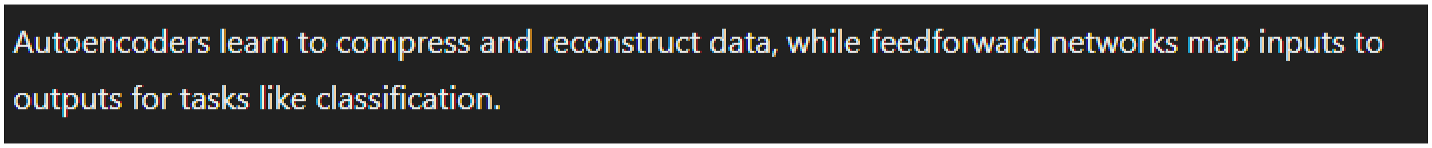
Inferences

The autoencoder effectively compresses and reconstructs images, demonstrating feature learning.

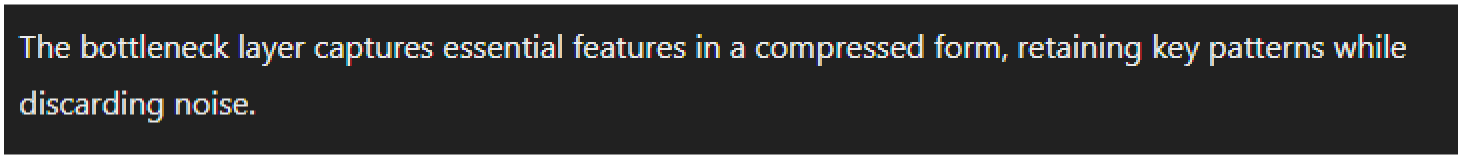
Experiment #	<TO BE FILLED BY STUDENT>	Student ID	<TO BE FILLED BY STUDENT>
Date	<TO BE FILLED BY STUDENT>	Student Name	[@KLWKS_bot] THANOS

Sample VIVA-VOCE Questions (In-Lab):

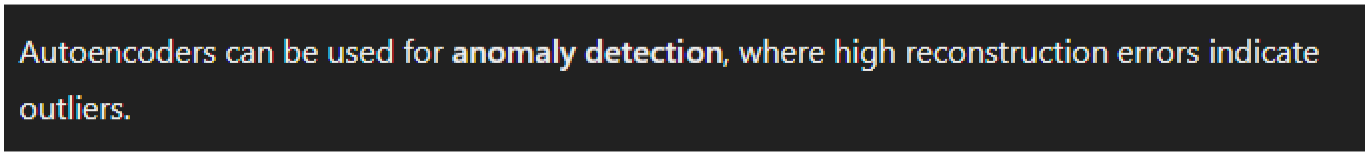
1. How does an autoencoder differ from a traditional feedforward neural network?


Autoencoders learn to compress and reconstruct data, while feedforward networks map inputs to outputs for tasks like classification.

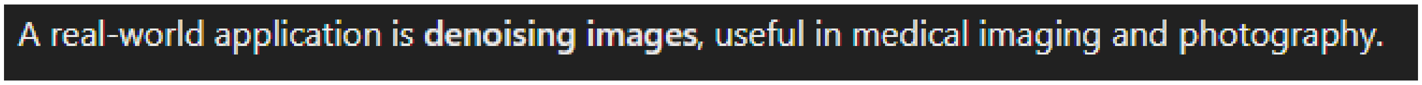
2. Why is the bottleneck layer often referred to as the 'latent space' in an autoencoder?


The bottleneck layer captures essential features in a compressed form, retaining key patterns while discarding noise.

3. Can an autoencoder be used for tasks other than dimensionality reduction and data compression? Provide an example.

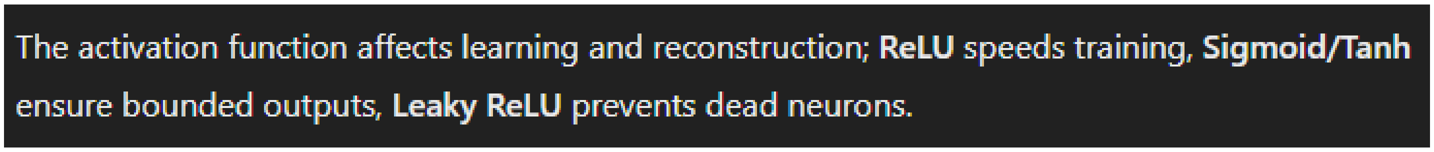

Autoencoders can be used for **anomaly detection**, where high reconstruction errors indicate outliers.

4. Discuss one potential application of autoencoders in real-world scenarios.



A real-world application is **denoising images**, useful in medical imaging and photography.

5. How does the choice of activation function impact the performance of an autoencoder?



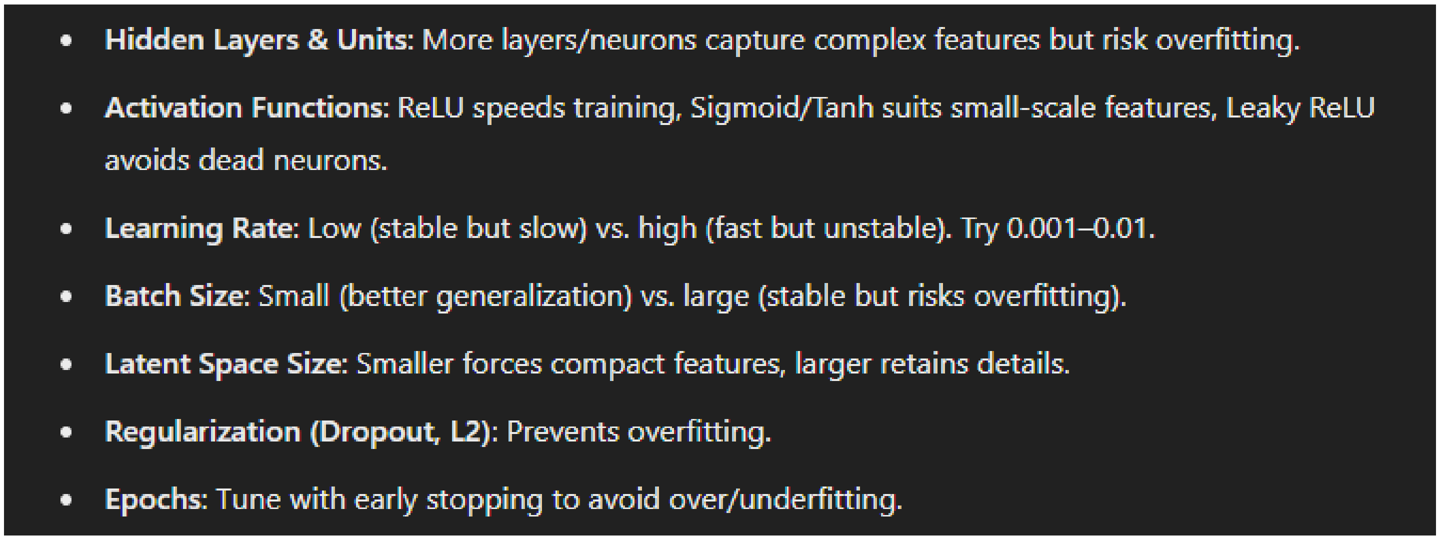
The activation function affects learning and reconstruction; **ReLU** speeds training, **Sigmoid/Tanh** ensure bounded outputs, **Leaky ReLU** prevents dead neurons.

Experiment #	<TO BE FILLED BY STUDENT>	Student ID	<TO BE FILLED BY STUDENT>
Date	<TO BE FILLED BY STUDENT>	Student Name	[@KLWKS_bot] THANOS

Post-Lab:

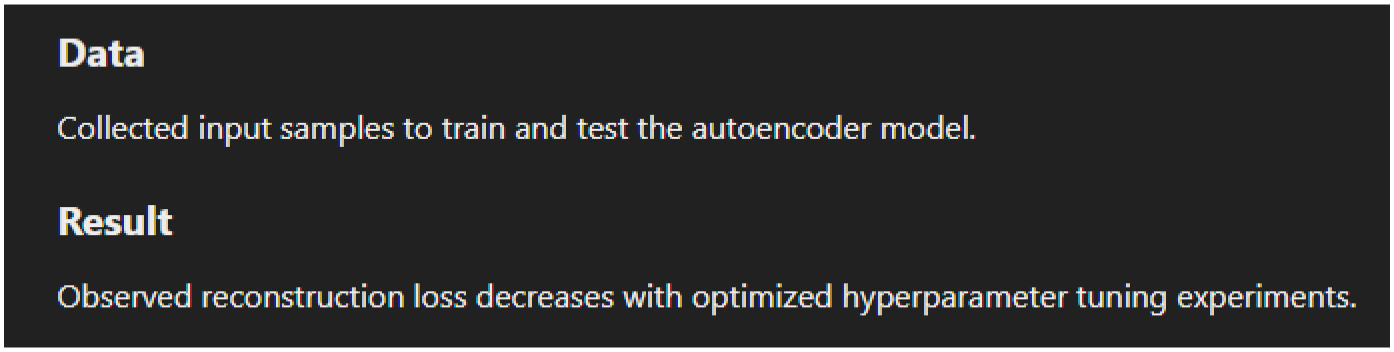
Program 2: Tuning Hyperparameters, Explore the impact of changing hyperparameters such as the number of hidden layers, units in the layers, and activation functions on the performance of the autoencoder.

Procedure/Program:



- **Hidden Layers & Units:** More layers/neurons capture complex features but risk overfitting.
- **Activation Functions:** ReLU speeds training, Sigmoid/Tanh suits small-scale features, Leaky ReLU avoids dead neurons.
- **Learning Rate:** Low (stable but slow) vs. high (fast but unstable). Try 0.001–0.01.
- **Batch Size:** Small (better generalization) vs. large (stable but risks overfitting).
- **Latent Space Size:** Smaller forces compact features, larger retains details.
- **Regularization (Dropout, L2):** Prevents overfitting.
- **Epochs:** Tune with early stopping to avoid over/underfitting.

- **Data and Results:**



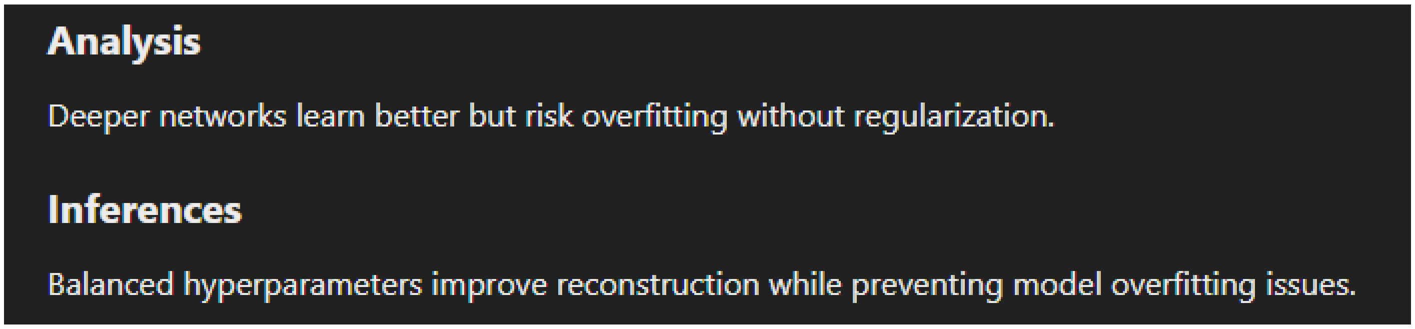
Data

Collected input samples to train and test the autoencoder model.

Result

Observed reconstruction loss decreases with optimized hyperparameter tuning experiments.

- **Analysis and Inferences:**



Analysis

Deeper networks learn better but risk overfitting without regularization.

Inferences

Balanced hyperparameters improve reconstruction while preventing model overfitting issues.

Evaluator Remark (if Any):	**Marks Secured ______ out of 50**
	Signature of the Evaluator with Date

Evaluator MUST ask Viva-voce prior to signing and posting marks for each experiment.