```python
# Lab 10: MNIST Compression using Autoencoder (PyTorch)

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

# ----------------------------
# 1. Hyperparameters
# ----------------------------
batch_size = 256
learning_rate = 1e-3
num_epochs = 10
latent_dim = 32

# ----------------------------
```

```python
# ----------------------------
# 2. MNIST Dataset
# ----------------------------
transform = transforms.Compose([transforms.ToTensor()])

train_dataset = datasets.MNIST(root='./data', train=True, transform=transform, download=True)
test_dataset = datasets.MNIST(root='./data', train=False, transform=transform, download=True)

train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=False)

# ----------------------------
# 3. Autoencoder Model
# ----------------------------
class Autoencoder(nn.Module):
    def __init__(self):
        super(Autoencoder, self).__init__()
        # Encoder
        self.encoder = nn.Sequential(
            nn.Flatten(),
```

{} Variables   Terminal

```python
            nn.Linear(28*28, 128),
            nn.ReLU(),
            nn.Linear(128, 64),
            nn.ReLU(),
            nn.Linear(64, latent_dim)
        )
        # Decoder
        self.decoder = nn.Sequential(
            nn.Linear(latent_dim, 64),
            nn.ReLU(),
            nn.Linear(64, 128),
            nn.ReLU(),
            nn.Linear(128, 28*28),
            nn.Sigmoid(),        # output between 0 and 1
            nn.Unflatten(1, (1,28,28))
        )

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

{} Variables   Terminal

```
# 4. Model, Loss, Optimizer
# ----------------------------
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = Autoencoder().to(device)
criterion = nn.MSELoss()
optimizer = optim.Adam(model.parameters(), lr=learning_rate)

# ----------------------------
# 5. Training
# ----------------------------
losses = []  # To store average loss per epoch

for epoch in range(num_epochs):
    model.train()
    train_loss = 0
    for data, _ in train_loader:
        data = data.to(device)
        optimizer.zero_grad()
        output = model(data)
        loss = criterion(output, data)
        loss.backward()
```

{} Variables   Terminal

```
            optimizer.step()
            train_loss += loss.item() * data.size(0)  # sum loss over batch

        avg_loss = train_loss / len(train_loader.dataset)
        losses.append(avg_loss)
        print(f"Epoch [{epoch+1}/{num_epochs}], Loss: {avg_loss:.6f}")

    # ----------------------------
    # Plot Loss vs Epoch
    # ----------------------------
    plt.figure(figsize=(8,5))
    plt.plot(range(1, num_epochs+1), losses, marker='o')
    plt.title('Training Loss vs Epoch')
    plt.xlabel('Epoch')
    plt.ylabel('Loss (MSE)')
    plt.grid(True)
    plt.show()

    # ----------------------------
    # 6. Testing & Visualization
    # ----------------------------
```

{ } Variables   Terminal

```python
model.eval()
with torch.no_grad():
    for data, _ in test_loader:
        data = data.to(device)
        reconstructed = model(data)
        break  # take first batch

# Display original and reconstructed images
n = 10
plt.figure(figsize=(20,4))
for i in range(n):
    # Original
    ax = plt.subplot(2, n, i+1)
    plt.imshow(data[i].cpu().squeeze(), cmap='gray')
    plt.title("Original")
    plt.axis('off')

    # Reconstructed
    ax = plt.subplot(2, n, i+1+n)
    plt.imshow(reconstructed[i].cpu().squeeze(), cmap='gray')
    plt.title("Reconstructed")
```

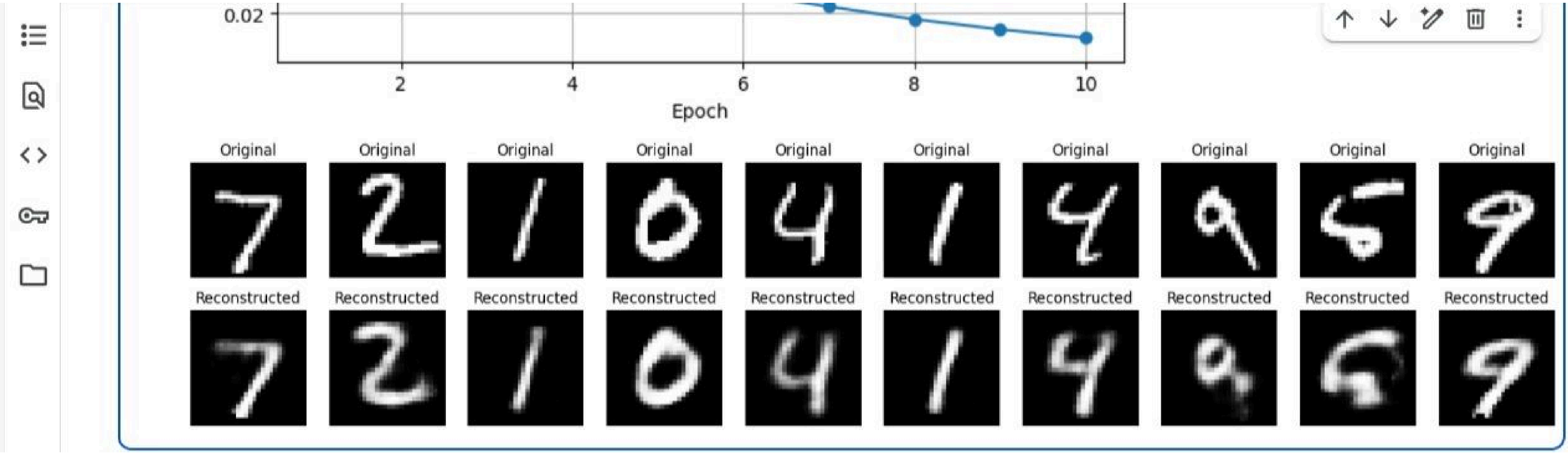
{} Variables   Terminal

```
    plt.axis('off')
plt.show()
```

```
Epoch [1/10], Loss: 0.074551
Epoch [2/10], Loss: 0.043603
Epoch [3/10], Loss: 0.032213
Epoch [4/10], Loss: 0.027764
Epoch [5/10], Loss: 0.024732
Epoch [6/10], Loss: 0.022521
Epoch [7/10], Loss: 0.020834
Epoch [8/10], Loss: 0.019298
Epoch [9/10], Loss: 0.018088
Epoch [10/10], Loss: 0.017080
```

Training Loss vs Epoch

{ } Variables
Terminal


0.02
2
4
6
8
10
Epoch
Original
Original
Original
Original
Original
Original
Original
Original
Original
Original
Reconstructed
Reconstructed
Reconstructed
Reconstructed
Reconstructed
Reconstructed
Reconstructed
Reconstructed
Reconstructed
Reconstructed

10. Perform Compression on MNIST dataset using auto encoder

Aim :

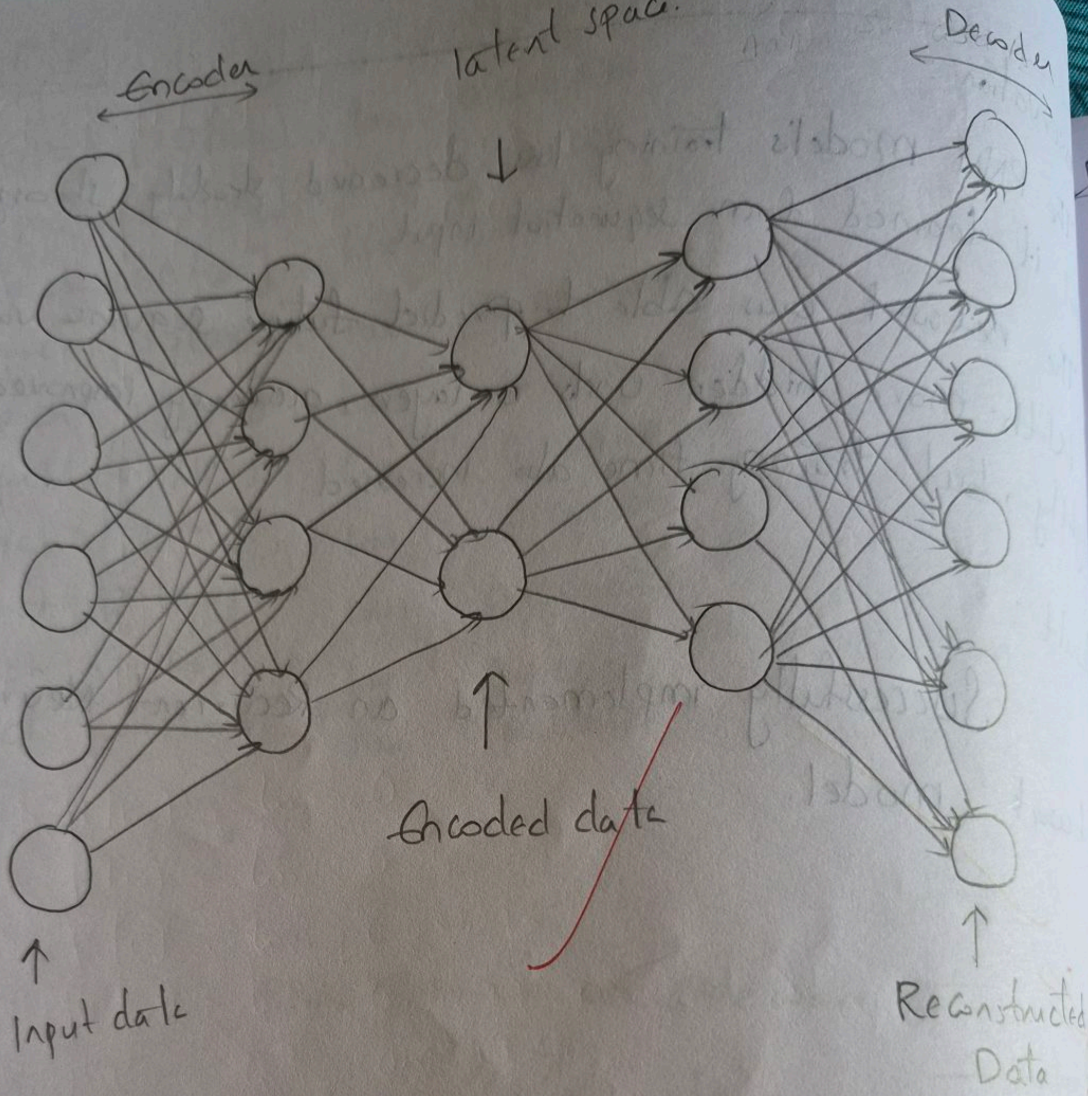
To Compress and reconstruct the MNIST handwritten digit dataset using an autoencoder.

Objective :

1. To understand the Concept of autoencoders for data Compression & reconstruction
2. To implement autoencoder using neural network framework
3. To Compress the MNIST dataset to lower dimensional representation
4. To reconstruct the original images from the Compressed representation
5. To Visualise Compressed representation of the digits.

Pseudo Code :

1. Import required libraries
2. Load MNIST dataset
   - x_train, x_test, y_train, y_test = mnist.load_data()
   - Normalize data to range [0, 1]
3. Flatten images to vectors (28x28 -> 784)
4. Define the Autoencoder architecture
   - Input layer = 784
   - Encoder layers = 128 -> 64 -> 32
   - Decoder layer = 32 -> 64 -> 128 -> 784
   - Activation function : ReLU for hidden layers, sigmoid for Output layer



Architecture of Encoder

5. Compile the model

6. Train the autoencoder
   - Input = x_train, output = x_train

7. Evaluate the model
   - Reconstruct images from x_test

8. Visualize
   - Original vs Reconstructed images

Observation:

1. The autoencoder Successfully Compressed MNIST images from 784 dimensions to 32 dimensions.

2. Reconstruction quality depends on the size of latent space & network depth.

3. Some fine details may be lost during compression

4. Loss gradually decreases during training, showing that the network learns to reconstruct images effectively.

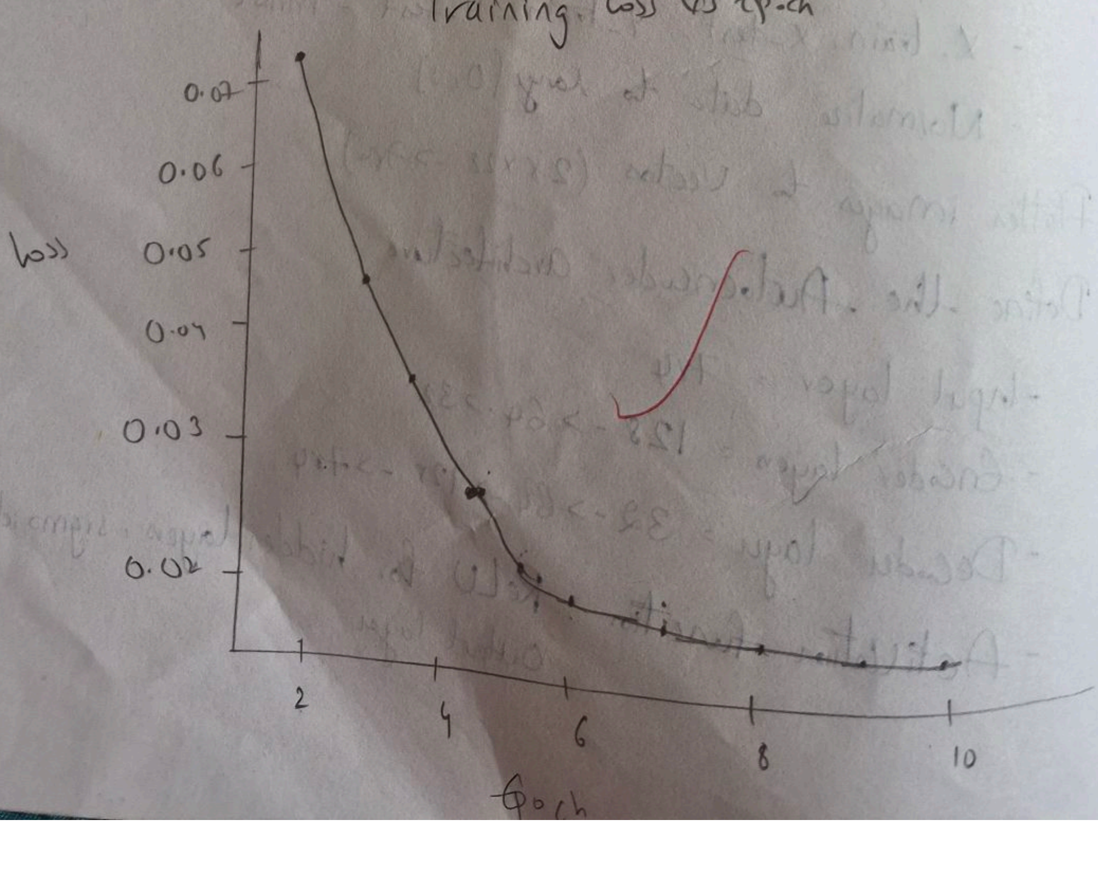
Result:

Successfully implemented and performed Compression on MNIST data using autoencoder

Epoch [1/20], loss : 0.076121
Epoch [2/20], loss : 0.041204
Epoch [3/20], loss : 0.030499
Epoch [4/20]
Epoch [5/20]
Epoch [6/20]
Epoch [7/20]
Epoch [8/20]
Epoch [9/20]
Epoch [10/20]
Epoch [11/20]
Epoch [12/20]
Epoch [13/20]
Epoch [14/20]
Epoch [15/20]
Epoch [16/20]
Epoch [17/20]
Epoch [18/20]
Epoch [19/20]
Epoch [20/20]

Epoch [1/10], loss : 0.074551
Epoch [2/10], loss : 0.043601
Epoch [3/10], loss : 0.032213
Epoch [4/10], loss : 0.027764
Epoch [5/10], loss : 0.024732
Epoch [6/10], loss : 0.022521
Epoch [7/10], loss : 0.020834
Epoch [8/10], loss : 0.017298
Epoch [9/10], loss : 0.018008
Epoch [10/10], loss : 0.17080

Training loss Vs Epoch



5. Compile the
6. Train the
- Input =
7. Evaluate the
- Recon
8. Visualize
- Orig

Observation

1. The autoe
784 dimens
2. Reconstruct
depth.
3. Some fin
4. Loss a
the netw

Result :
S
on MNI

```
#lab 11
import torch
import torch.nn as nn
import torch.optim as optim
from torchvision import datasets, transforms
from torch.utils.data import DataLoader
import matplotlib.pyplot as plt

# ----------------------------
# Hyperparameters
# ----------------------------
batch_size = 128
learning_rate = 1e-3
num_epochs = 10
latent_dim = 20
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# ----------------------------
# MNIST Dataset
# ----------------------------
```

{} Variables   Terminal

```python
# ----------------------------
# MNIST Dataset
# ----------------------------
transform = transforms.ToTensor()
train_dataset = datasets.MNIST(root='./data', train=True, transform=transform, download=True)
test_dataset = datasets.MNIST(root='./data', train=False, transform=transform, download=True)
train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=False)

# ----------------------------
# VAE Model
# ----------------------------
class VAE(nn.Module):
    def __init__(self):
        super(VAE, self).__init__()
        # Encoder
        self.fc1 = nn.Linear(28*28, 400)
        self.fc_mu = nn.Linear(400, latent_dim)
        self.fc_logvar = nn.Linear(400, latent_dim)
        # Decoder
        self.fc3 = nn.Linear(latent_dim, 400)
```

{ } Variables   Terminal

```
    def encode(self, x):
        h1 = torch.relu(self.fc1(x))
        return self.fc_mu(h1), self.fc_logvar(h1)

    def reparameterize(self, mu, logvar):
        std = torch.exp(0.5*logvar)
        eps = torch.randn_like(std)
        return mu + eps*std

    def decode(self, z):
        h3 = torch.relu(self.fc3(z))
        return torch.sigmoid(self.fc4(h3))

    def forward(self, x):
        mu, logvar = self.encode(x.view(-1, 28*28))
        z = self.reparameterize(mu, logvar)
        return self.decode(z), mu, logvar

# ----------------------------
# Loss Function
```

{ } Variables   Terminal

```python
# ----------------------------
# Loss Function
# ----------------------------
def loss_function(recon_x, x, mu, logvar):
    BCE = nn.functional.binary_cross_entropy(recon_x, x.view(-1, 28*28), reduction='sum')
    KLD = -0.5 * torch.sum(1 + logvar - mu.pow(2) - logvar.exp())
    return BCE + KLD

# ----------------------------
# Model, Optimizer
# ----------------------------
model = VAE().to(device)
optimizer = optim.Adam(model.parameters(), lr=learning_rate)

# ----------------------------
# Training
# ----------------------------
losses = []  # list to store average loss per epoch

for epoch in range(num_epochs):
    model.train()
```

{} Variables   Terminal

```
            data = data.to(device)
            optimizer.zero_grad()
            recon_batch, mu, logvar = model(data)
            loss = loss_function(recon_batch, data, mu, logvar)
            loss.backward()
            train_loss += loss.item()
            optimizer.step()
        avg_loss = train_loss / len(train_loader.dataset)
        losses.append(avg_loss)
        print(f'Epoch [{epoch+1}/{num_epochs}], Loss: {avg_loss:.4f}')

    # ----------------------------
    # Plot Loss vs Epoch
    # ----------------------------
    plt.figure(figsize=(8,5))
    plt.plot(range(1, num_epochs+1), losses, marker='o')
    plt.title('Training Loss vs Epoch')
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.grid(True)
    plt.show()
```

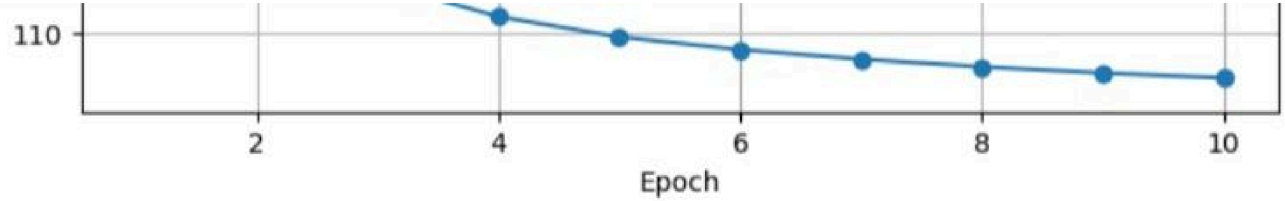
{} Variables   Terminal

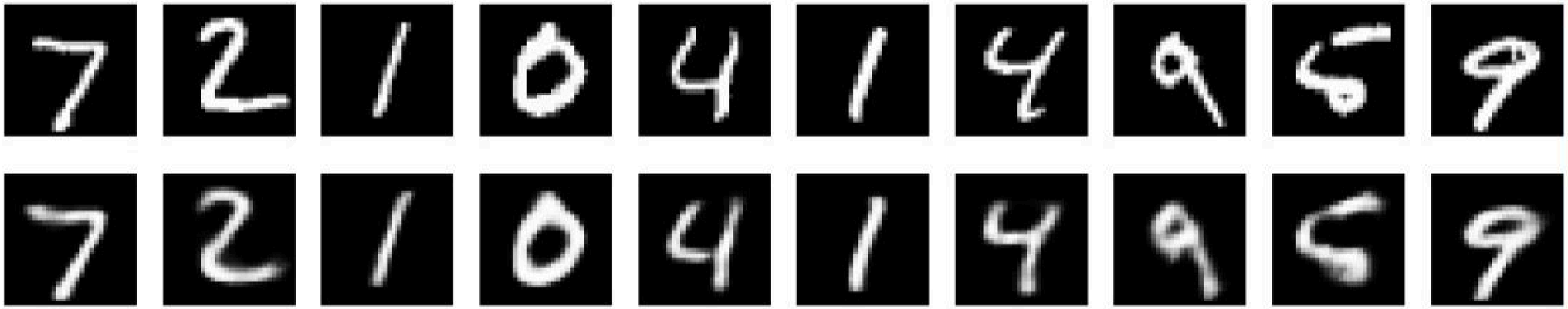
```python
# ------------------------------
model.eval()
with torch.no_grad():
    for data, _ in test_loader:
        data = data.to(device)
        recon, _, _ = model(data)
        break  # first batch only

# Display original and reconstructed images
n = 10
plt.figure(figsize=(20,4))
for i in range(n):
    # Original
    ax = plt.subplot(2, n, i+1)
    plt.imshow(data[i].cpu().squeeze(), cmap='gray')
    plt.axis('off')
    # Reconstructed
    ax = plt.subplot(2, n, i+1+n)
    plt.imshow(recon[i].cpu().view(28,28), cmap='gray')
    plt.axis('off')
plt.show()
```

{} Variables   Terminal

```
Epoch [1/10], Loss: 164.8592
Epoch [2/10], Loss: 121.2283
Epoch [3/10], Loss: 114.4237
Epoch [4/10], Loss: 111.4566
Epoch [5/10], Loss: 109.6797
Epoch [6/10], Loss: 108.5698
Epoch [7/10], Loss: 107.7450
Epoch [8/10], Loss: 107.0712
Epoch [9/10], Loss: 106.5329
Epoch [10/10], Loss: 106.1177
```


Training Loss vs Epoch
Loss
160
150
140
130
120
110
{ } Variables
Terminal


110
2
4
6
8
10
Epoch



21/10/25 11. Experiments Using Variational Autoencoder

Aim :

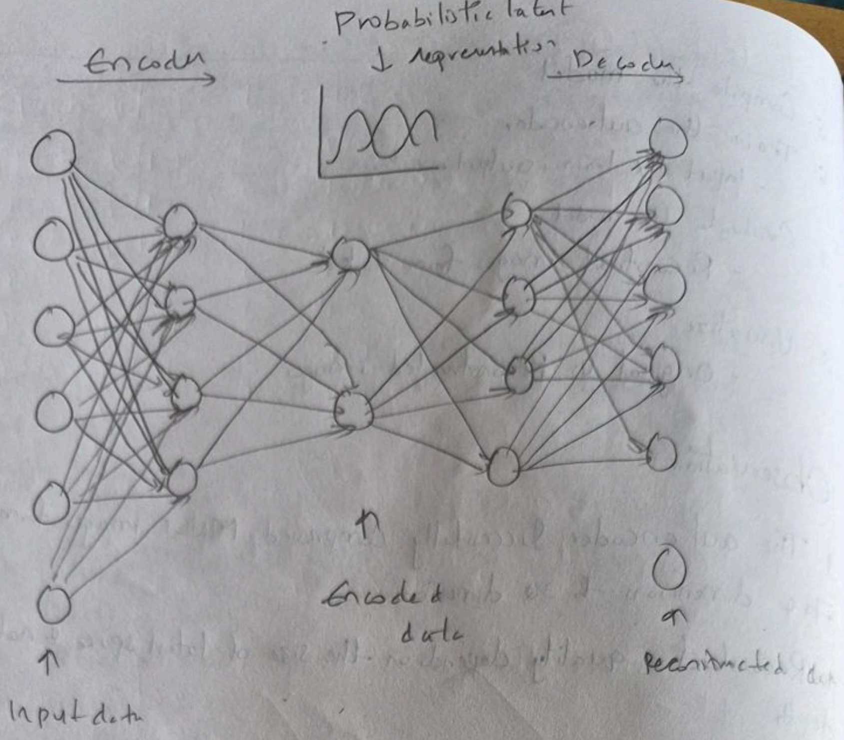
To implement a Variational autoencoder on MNIST dataset and observe its ability to learn a latent representation for generating and reconstructing images.

Objective :

1. To understand the concept of VAE and its difference from a standard autoencoder.
2. To implement a VAE using PyTorch
3. To compress the MNIST dataset into a latent space & sample new images from it.
4. To reconstruct original images and analyze the quality of generated outputs
5. To visualize the latent space and understand clustering of digit representations.

Pseudo Code :

1. Import required libraries
2. Load MNIST dataset and normalize
3. Define VAE model
   - Encoder : Output mean (mu) and log variance (logvar) of latent distribution
   - Reparameterization trick : z = mu + std * epsilon
4. Define loss function : Reconstruction loss, KL divergence loss
5. Train the VAE
   - Forward pass : Compute mu, logvar, z, reconstruction
   - Compute total loss : reconstruction + KL divergence
   - Backpropagate & optimize



Variational Autoencoder Architecture

6. Test VAE
   - Reconstruct test images
   - Generate new images by sampling latent space
7. Visualize
   - Original vs reconstructed image
   - Generated images

Observations:

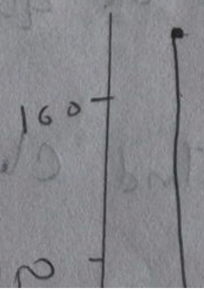
1. VAE compressed images into probabilistic latent space
2. KL divergence ensures that latent space follows a standard normal distribution.
3. Generated images from random samples of latent space resembles the digits
4. Reconstruction is slightly blurrier than standard autoencoders due to probabilistic sampling.

Result:

Successfully implemented the experiments using Variational autoencoder.

Epoch [1/10] loss : 164.6592
Epoch [2/10] loss : 121.2283
Epoch [3/10] loss : 114.4237
Epoch [4/10] loss : 111.4566
Epoch [5/10] loss : 109.6797
Epoch [6/10] loss : 108.5698
Epoch [7/10] loss : 107.7450
Epoch [8/10] loss : 107.0712
Epoch [9/10] loss : 106.5329
Epoch [10/10] loss : 106.1172

Training loss Vs Epoch



6. Test VAE
- Reconstruct test images
- Generate new images by sampl

7. Visualize
- Original vs reconstructed in
- Generated images

Observations :

1. VAE compressed images
2. KL divergence ensure
normal distribution.
3. Generated images fro
resembles the digits
4. Reconstruction is
due to probabilistic

Result :

Epoch [9/10]
Epoch [10/10] Loss: 106.1172

Training Loss Vs Epoch