**EXPERIMENT 6** Date:

**Problem Definition:** Implementation of an Autoencoder

**Packages Used:** PyTorch, matplotlib

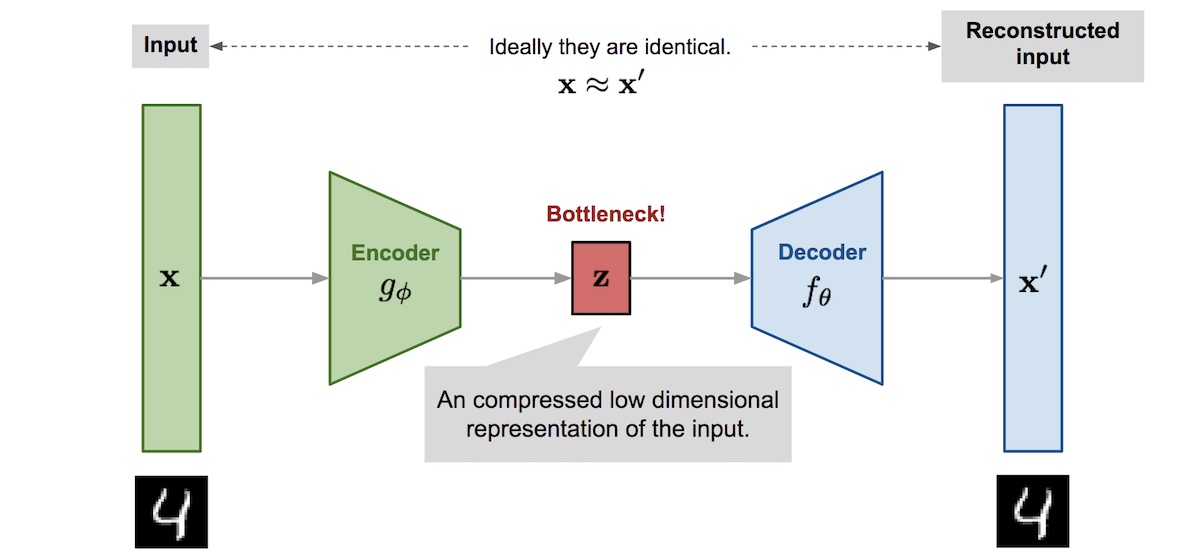
**Dataset Used:** MNIST dataset

**Theory:**

Autoencoders are a type of artificial neural network used primarily for unsupervised learning. Their primary objective is to encode the input data into a compressed, low-dimensional representation and then reconstruct the data from this representation. Autoencoders can be useful in applications like data compression, denoising, and feature extraction. The network consists of two main parts:

* **Encoder**: Reduces the input dimensions, creating a compressed representation.
* **Decoder**: Reconstructs the original input from the compressed data.

Autoencoders are trained to minimize reconstruction loss, enabling them to learn the most important features in the data.

****

**Autoencoder Structure**

1. Input Layer: Takes the raw input data.
2. Encoder: Compresses the data by learning a lower-dimensional representation.
3. Bottleneck: Represents the compressed form of the input data.
4. Decoder: Reconstructs the input data from the bottleneck representation.
5. Output Layer: The reconstructed data, which ideally should resemble the input.

**Mathematical Formulation of a Plain Autoencoder**

The autoencoder can be defined in terms of the following mathematical functions:

1. **Encoding Function**:



where:

* + x is the input data.
  + Wenc​ and bencb​ are the weights and biases of the encoder.
  + σ is an activation function, typically a non-linear function like ReLU or Sigmoid.
  + h is the encoded or latent representation.

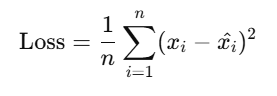
1. **Decoding Function**:



where:

* + x^ is the reconstructed output.
  + Wdec​ and bdec​ are the weights and biases of the decoder.

1. **Loss Function**:
   * The aim of the autoencoder is to minimize the reconstruction error, which measures how close x^ is to x. A commonly used loss function is Mean Squared Error (MSE):



This loss function guides the network in adjusting its weights to improve the accuracy of the reconstructed data.

**Working Mechanism of an Autoencoder**

1. **Input Data**: The original data, xxx, is fed into the encoder.
2. **Encoding (Bottleneck)**: The encoder maps xxx to a compressed latent space representation, hhh, capturing the most relevant features. This part of the model learns an efficient encoding that retains essential information while discarding noise and redundancy.
3. **Decoding and Reconstruction**: The decoder takes hhh and attempts to reconstruct the original input xxx as x^\hat{x}x^. The quality of reconstruction is evaluated using a loss function.
4. **Training**: The network is trained by backpropagating the reconstruction loss, which helps the encoder-decoder weights improve the encoding and reconstruction process iteratively.

**Applications of Plain Autoencoders**

Autoencoders have several applications in machine learning and data science:

1. **Dimensionality Reduction**: Autoencoders can reduce data dimensionality without explicit labels, like Principal Component Analysis (PCA) but in a non-linear way.
2. **Data Denoising**: By training on noisy input data, autoencoders can learn to filter out noise during the reconstruction process.
3. **Anomaly Detection**: If an autoencoder is trained on normal data, it will struggle to reconstruct anomalies, making reconstruction error a potential indicator of outliers.
4. **Data Compression**: The encoding from the bottleneck layer can serve as a compressed representation, useful for tasks like image and audio compression.
5. **Image Generation**: Autoencoders can be used to generate new samples by interpolating within the latent space.

**Advantages and Limitations of Autoencoders**

**Advantages**:

* Simple to implement and effective for data representation learning.
* Useful for unsupervised learning tasks where labels are not available.
* Capable of learning complex non-linear relationships in data.

**Limitations**:

* Prone to overfitting, especially on small datasets.
* Do not explicitly enforce disentangling of features, leading to entangled representations.
* Limited to reconstructing data similar to the training set, making them less suitable for tasks that require high diversity in the data.

**Plain Autoencoder vs. Variants**

Plain autoencoders are the simplest type of autoencoder, with no added constraints. However, there are several other types with different objectives:

* **Denoising Autoencoder**: Learns to reconstruct clean data from noisy input, making it robust to noise.
* **Sparse Autoencoder**: Adds a sparsity constraint on the hidden layer, forcing the network to learn more efficient representations by activating only a subset of neurons.
* **Variational Autoencoder (VAE)**: Introduces probabilistic layers to learn a distribution over the latent space, allowing for sampling and generating new data points.

Each variant serves a specific purpose or improves on limitations of plain autoencoders.

**Implementation of Autonencoder in Pytorch:**

import torch

import torch.nn as nn

import torch.optim as optim

from torchvision import datasets, transforms

import matplotlib.pyplot as plt

transform = transforms.ToTensor()

mnist\_data = datasets.MNIST(root='./data', train=True, download=True, transform=transform)

data\_loader = torch.utils.data.DataLoader(dataset=mnist\_data, batch\_size=64, shuffle=True)

dataiter = iter(data\_loader)

images, labels = next(dataiter)

print(torch.min(images), torch.max(images))



class Autoencoder(nn.Module):

  def \_\_init\_\_(self):

    # N, 784

    super().\_\_init\_\_(

    )

    self.encoder = nn.Sequential(

        nn.Linear(28\*28, 128),  #N, 784 -> N,128

        nn.ReLU(),

        nn.Linear(128, 64),

        nn.ReLU(),

        nn.Linear(64, 12),

        nn.ReLU(),

        nn.Linear(12, 3) # -> N, 3

    )

    self.decoder = nn.Sequential(

        nn.Linear(3, 12),

        nn.ReLU(),

        nn.Linear(12, 64),

        nn.ReLU(),

        nn.Linear(64, 128),

        nn.ReLU(),

        nn.Linear(128, 28\*28),

        nn.Sigmoid() # N, 3 -> N,784

    )

  def forward(self, x):

    encoded = self.encoder(x)

    decoded = self.decoder(encoded)

    return decoded

  #note: [-1, 1] -> nn.Tanh()

model = Autoencoder()

criterion = nn.MSELoss()

optimizer = torch.optim.Adam(model.parameters(), lr=1e-3, weight\_decay=1e-5)

num\_epochs = 10

outputs = []

for epoch in range(num\_epochs):

  for (img, \_) in data\_loader:

img = img.reshape(-1, 28\*28)

    recon = model(img)

    loss = criterion(recon, img)

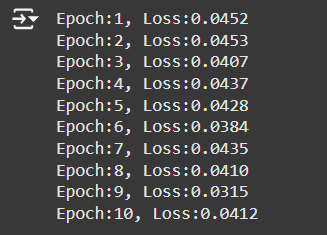
    optimizer.zero\_grad()

    loss.backward()

    optimizer.step()

  print(f'Epoch:{epoch+1}, Loss:{loss.item():.4f}')

  outputs.append((epoch, img, recon))



for k in range(0, num\_epochs, 4):

  plt.figure(figsize=(9, 2))

  plt.gray()

  imgs = outputs[k][1].detach().numpy()

  recon = outputs[k][2].detach().numpy()

  for i, item in enumerate(imgs):

    if i >= 9:

      break

    plt.subplot(2, 9, i+1)

item = item.reshape(-1, 28, 28)

    #item: 1, 28, 28

    plt.imshow(item[0])

  for i, item in enumerate(recon):

    if i >= 9:

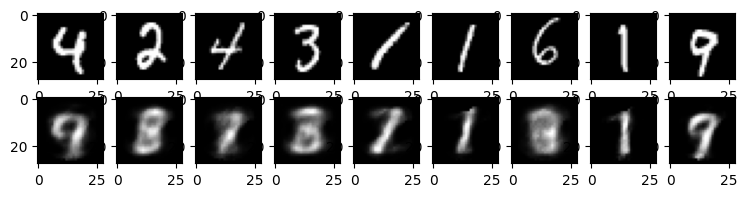
      break

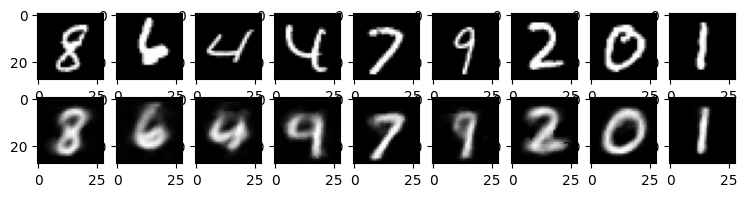
    plt.subplot(2, 9, 9+i+1) #row\_length + i + 1

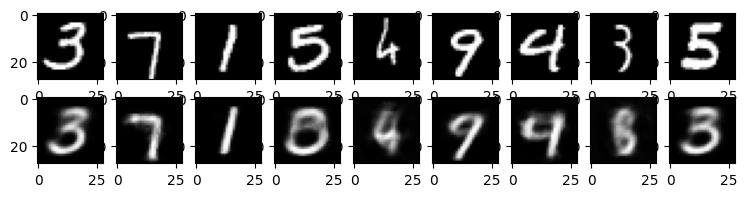
    item = item.reshape(-1, 28, 28)

    #item: 1, 28, 28

    plt.imshow(item[0])





**Conclusion:**

Autoencoder was studied and implemented successfully.