Latent Space Representation using AE & VAE

Objective

The objective of this module is to implement **Autoencoder (AE)** and **Variational Autoencoder (VAE)** models on the **MNIST** dataset, visualize their latent space representations using **PCA** and **t-SNE**, evaluate their reconstruction quality, and explore latent vector arithmetic for generative understanding.

Introduction

Autoencoders are unsupervised learning models designed to compress data into a latent representation and reconstruct the input from this encoding. While standard AEs learn deterministic mappings, **Variational Autoencoders (VAEs)** introduce a probabilistic framework that enables better generative capabilities.

Model Architectures

Autoencoder (AE)

Encoder Architecture:
Input Layer (784) → Hidden Layer (128) → Latent Space (2D)

Decoder Architecture:
Latent Space (2D) → Hidden Layer (128) → Output Layer (784)

Activation Functions:

- Hidden layers: ReLU
- Output layer: Sigmoid

Loss Function:
Binary Cross-Entropy (BCE)

Variational Autoencoder (VAE)

- **Encoder Architecture**:
  Input Layer (784) → Hidden Layer (128) → Latent Outputs: μ (mean), $\log\sigma^2$ (log-variance)

- **Reparameterization Trick**:
  z=μ+σ · εz = \mu + \sigma \cdot \varepsilonz=μ+σ · ε where ε~N(0,1)\varepsilon \sim \mathcal{N}(0, 1)ε~N(0,1)

- **Decoder Architecture**:
  Latent Vector (z, 2D) → Hidden Layer (128) → Output Layer (784)

- **Loss Function**:
  BCE + KL Divergence
  (KL term regularizes the latent space to resemble a standard normal distribution)

Training Results

AE Loss Curve

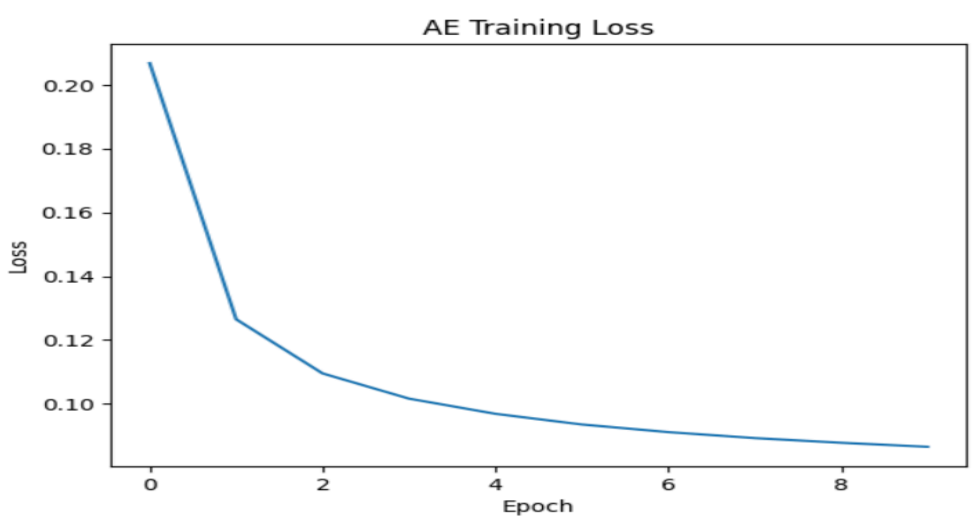
The AE (Autoencoder) loss curve shows how the model's reconstruction error decreases over training epochs. Since the AE uses **Binary Cross-Entropy (BCE)** loss, the curve typically shows a **smooth and steady decline**, indicating that the model is learning to compress and reconstruct the input data effectively.

Code:

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torchvision import datasets, transforms
from torch.utils.data import DataLoader
import matplotlib.pyplot as plt
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
transform = transforms.ToTensor()
train_dataset = datasets.MNIST(root='./data', train=True, transform=transform,
                               download=True)
train_loader = DataLoader(train_dataset, batch_size=128, shuffle=True)
class Autoencoder(nn.Module):
    def __init__(self):
        super(Autoencoder, self).__init__()
        self.encoder = nn.Sequential(
            nn.Flatten(),
            nn.Linear(784, 128),
            nn.ReLU(),
            nn.Linear(128, 32)
        )
        self.decoder = nn.Sequential(
            nn.Linear(32, 128),
            nn.ReLU(),
            nn.Linear(128, 784),
            nn.Sigmoid()
        )
     def forward(self, x):
        z = self.encoder(x)
        x_recon = self.decoder(z)
        return x_recon
```

```python
def train_ae(model, epochs=10):
    model.train()
    optimizer = optim.Adam(model.parameters(), lr=1e-3)
    criterion = nn.BCELoss()
    losses = []
    for epoch in range(epochs):
        total_loss = 0
        for images, _ in train_loader:
            images = images.to(device)
            recon = model(images)
            loss = criterion(recon, images.view(-1, 784))

            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
            total_loss += loss.item()

        avg_loss = total_loss / len(train_loader)
        print(f"Epoch [{epoch+1}/{epochs}], Loss: {avg_loss:.4f}")
        losses.append(avg_loss)
    # Plot the loss curve
    plt.plot(losses)
    plt.title("AE Training Loss")
    plt.xlabel("Epoch")
    plt.ylabel("Loss")
    plt.show()
    plt.close()
# Run AE training
ae = Autoencoder().to(device)
train_ae(ae)
```

Output:



VAE Loss Curve

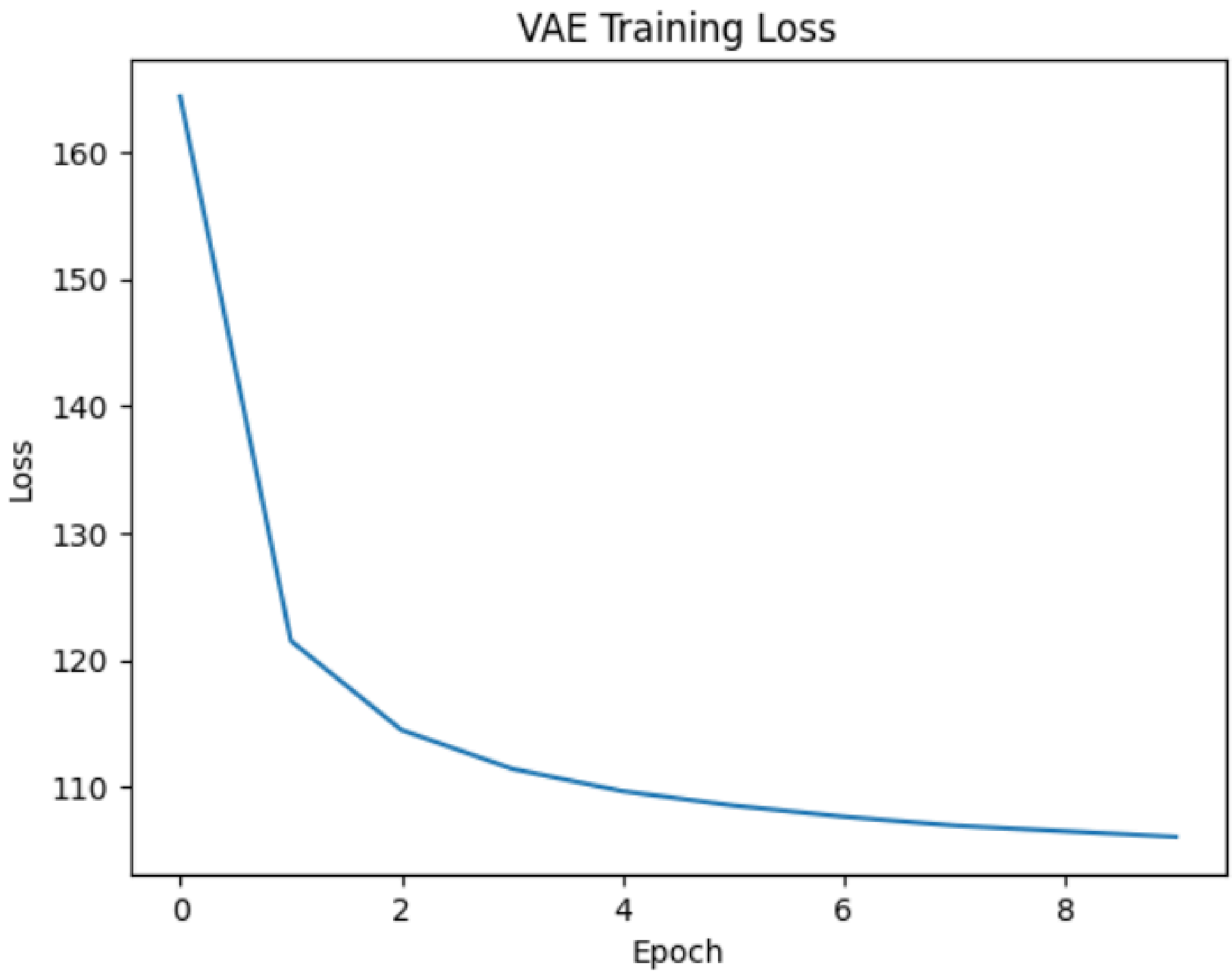
The VAE (Variational Autoencoder) loss curve is composed of two parts: Binary Cross-Entropy (BCE) and KL Divergence. Initially, the curve is higher than AE's because of the added KL divergence, which regularizes the latent space. Over time, the model balances reconstruction accuracy with latent space smoothness, leading to a steady decrease in total loss.

Code:

```
import torch
import torch.nn as nn
from torchvision import datasets, transforms
from torch.utils.data import DataLoader
import matplotlib.pyplot as plt
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
transform = transforms.ToTensor()
train_dataset = datasets.MNIST(root='./data', train=True, transform=transform,
                               download=True)
train_loader = DataLoader(train_dataset, batch_size=128, shuffle=True)
class VAE(nn.Module):
    def __init__(self):
        super(VAE, self).__init__()
        self.fc1 = nn.Linear(784, 400)
        self.fc21 = nn.Linear(400, 20)
        self.fc22 = nn.Linear(400, 20)
        self.fc3 = nn.Linear(20, 400)
        self.fc4 = nn.Linear(400, 784)
    def encode(self, x):
        h1 = F.relu(self.fc1(x))
        return self.fc21(h1), self.fc22(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 = F.relu(self.fc3(z))
        return torch.sigmoid(self.fc4(h3))
    def forward(self, x):
        mu, logvar = self.encode(x.view(-1, 784))
        z = self.reparameterize(mu, logvar)
        return self.decode(z), mu, logvar
```

```python
def train_vae(model, epochs=10):
    model.train()
    optimizer = optim.Adam(model.parameters(), lr=1e-3)
    losses = []
    for epoch in range(epochs):
        total_loss = 0
        for images, _ in train_loader:
            images = images.to(device)
            recon, mu, logvar = model(images)
            loss = vae_loss(recon, images, mu, logvar)

            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
            total_loss += loss.item()

        avg_loss = total_loss / len(train_loader.dataset)
        print(f"Epoch [{epoch+1}/{epochs}], Loss: {avg_loss:.4f}")
        losses.append(avg_loss)
    # Plot the loss curve
    plt.plot(losses)
    plt.title("VAE Training Loss")
    plt.xlabel("Epoch")
    plt.ylabel("Loss")
    plt.savefig("losses_curves.png")
    plt.show()
    plt.close()
# Run VAE training
vae = VAE().to(device)
train_vae(vae)
```

Output:

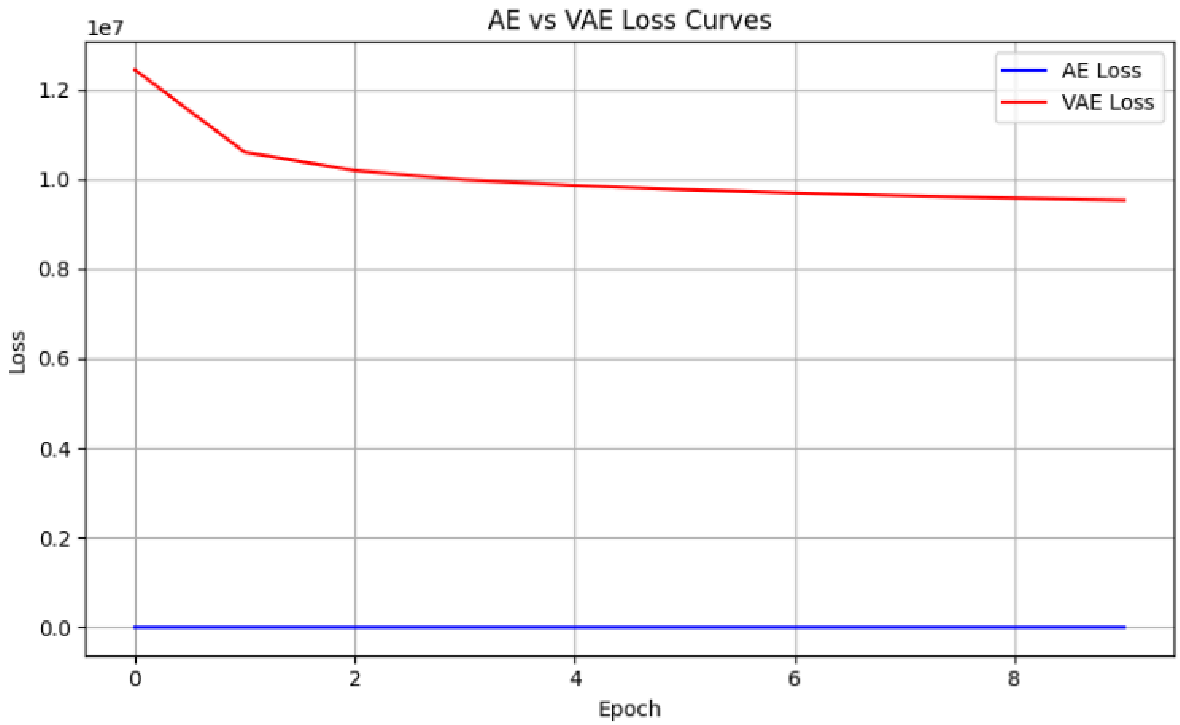


Note: VAE loss includes KL divergence, making total loss initially higher compared to AE.

Comparison Between AE and VAE Loss Curves

The loss curves of Autoencoders (AE) and Variational Autoencoders (VAE) show distinct behaviors due to their underlying loss functions. An AE minimizes only the reconstruction loss, such as Mean Squared Error, which leads to a smooth and faster decline in the loss curve with lower final loss values. In contrast, a VAE includes an additional KL Divergence term in its loss function to enforce a structured latent space. This results in a higher initial loss and slower convergence. While AEs typically achieve sharper reconstructions, VAEs offer better generalization and a well-organized latent space due to this regularization.

Visual Representation:



Latent Space Visualization

To understand how the models encode information, we visualized the 2D latent representations of the MNIST dataset using t-SNE, a non-linear dimensionality reduction technique. Each point represents an input image projected into latent space, and colors indicate digit labels (0–9). This helps assess how well the model separates different classes in its compressed representation.

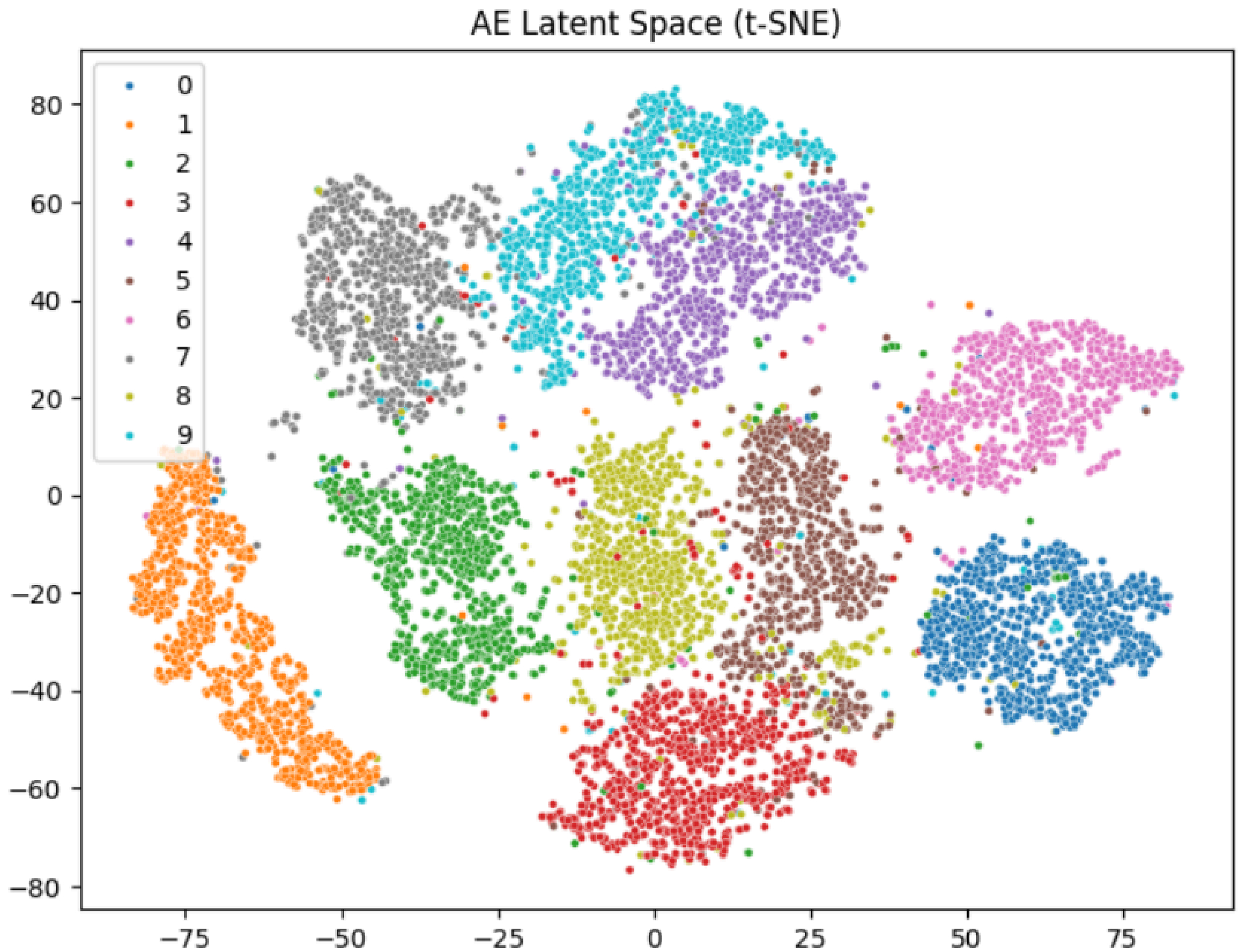
AE Latent Space

The AE latent space appears scattered and unstructured, with overlapping regions among digit classes. Without any constraint like KL divergence, the latent representation lacks clear separation, leading to less meaningful clustering.

```
import torch
from torchvision import datasets, transforms
from torch.utils.data import DataLoader
from sklearn.manifold import TSNE
from sklearn.decomposition import PCA
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
transform = transforms.ToTensor()
test_dataset = datasets.MNIST(root='./data', train=False, transform=transform,
                              download=True)
test_loader = DataLoader(test_dataset, batch_size=128, shuffle=False)
```

```
def visualize_latent_space(encoder,data_loader, use_tsne=True, model_type="AE"):
    encoder.eval()
    all_z = []
    all_labels = []
    with torch.no_grad():
        for images, labels in data_loader:
            images = images.to(device)
            if model_type == "AE":
                z = encoder(images).cpu().numpy()
            elif model_type == "VAE":
                _, mu, _ = encoder(images)
                z = mu.cpu().numpy()
            else:
                raise ValueError("Invalid model_type. Use 'AE' or 'VAE'.")
            all_z.append(z)
            all_labels.append(labels.numpy())
    all_z = np.concatenate(all_z)
    all_labels = np.concatenate(all_labels)
    if use_tsne:
        z_2d = TSNE(n_components=2, init='random', learning_rate='auto').
        fit_transform(all_z)
    else:
        z_2d = PCA(n_components=2).fit_transform(all_z)
    plt.figure(figsize=(8, 6))
    sns.scatterplot(x=z_2d[:, 0], y=z_2d[:, 1], hue=all_labels, palette="tab10",
                    s=10, legend="full")
    plt.title(f"{model_type} Latent Space ({'t-SNE' if use_tsne else 'PCA'})")
    plt.show()
    plt.close()
visualize_latent_space(ae.encoder, test_loader, use_tsne=True, model_type="AE")
```

Output:



VAE Latent Space

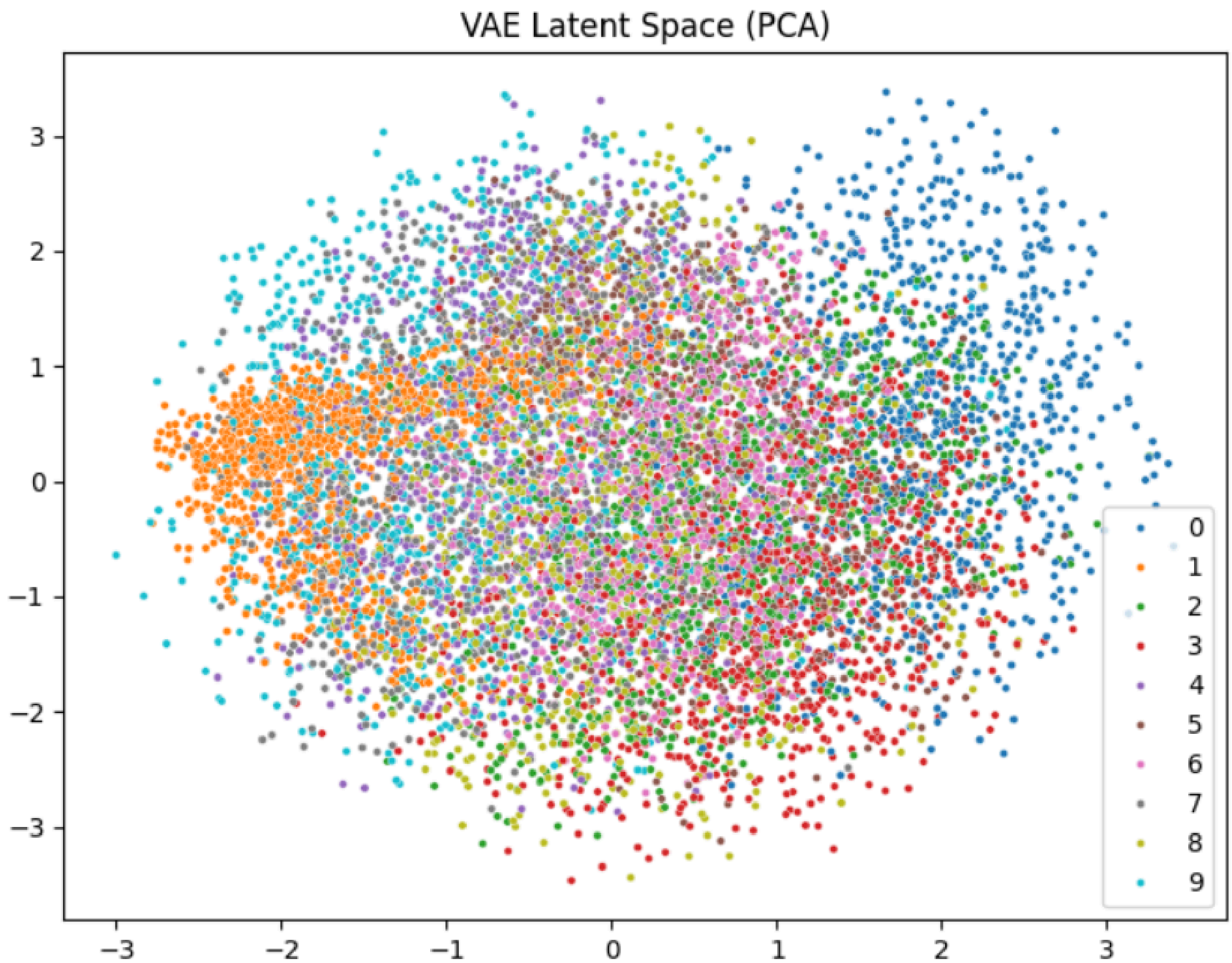
VAE latent space forms more structured and clustered regions, showing better separation between digit classes due to the regularization imposed by KL divergence. This encourages a continuous and smooth latent representation.

Code:

```python
import torch
from torchvision import datasets, transforms
from torch.utils.data import DataLoader
from sklearn.manifold import TSNE
from sklearn.decomposition import PCA
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
transform = transforms.ToTensor()
test_dataset = datasets.MNIST(root='./data', train=False, transform=transform,
                              download=True)
test_loader = DataLoader(test_dataset, batch_size=128, shuffle=False)
```

```python
def visualize_latent_space(encoder,data_loader,use_tsne=True, model_type="AE")
    encoder.eval()
    all_z = []
    all_labels = []
    with torch.no_grad():
        for images, labels in data_loader:
            images = images.to(device)
            if model_type == "AE":
                z = encoder(images).cpu().numpy()
            elif model_type == "VAE":
                _, mu, _ = encoder(images)
                z = mu.cpu().numpy()
            else:
                raise ValueError("Invalid model_type. Use 'AE' or 'VAE'.")
            all_z.append(z)
            all_labels.append(labels.numpy())
    all_z = np.concatenate(all_z)
    all_labels = np.concatenate(all_labels)
    if use_tsne:
        z_2d = TSNE(n_components=2, init='random', learning_rate='auto').
        fit_transform(all_z)
    else:
        z_2d = PCA(n_components=2).fit_transform(all_z)
    plt.figure(figsize=(8, 6))
    sns.scatterplot(x=z_2d[:, 0], y=z_2d[:, 1], hue=all_labels,palette="tab10"
                    s=10, legend="full")
    plt.title(f"{model_type} Latent Space ({'t-SNE' if use_tsne else 'PCA'})")
    plt.show()
    plt.close()
visualize_latent_space(vae, test_loader, use_tsne=False, model_type="VAE")
```

Output:



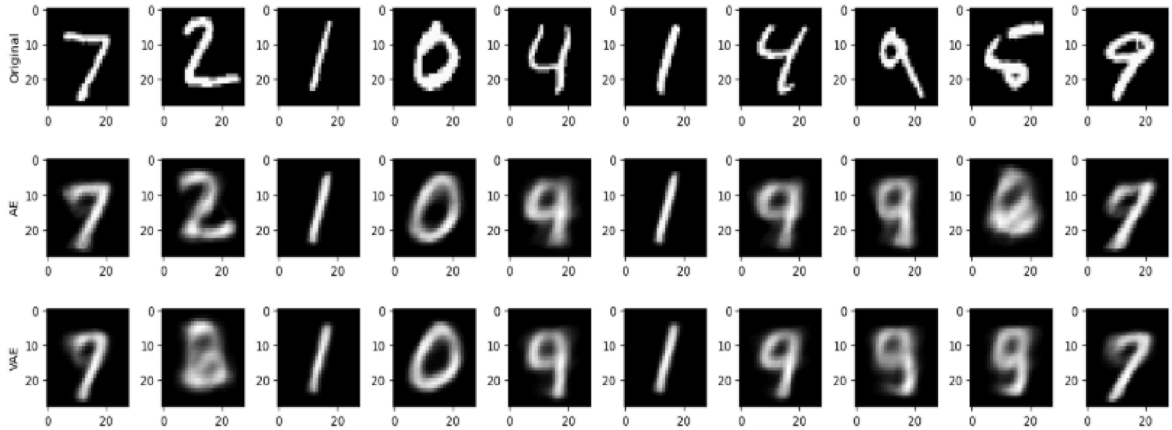
Observation: VAE latent space is better organized and more interpretable than AE.

Reconstruction Comparison

Code:

```
import torch
import matplotlib.pyplot as plt
def compare_reconstructions(ae, vae, data_loader, device):
    """Parameters:
    - ae: Trained Autoencoder model
    - vae: Trained Variational Autoencoder model
    - data_loader: DataLoader for test data
    - device: 'cuda' or 'cpu'
    """
    ae.eval()
    vae.eval()
    images, _ = next(iter(data_loader))
    images = images.to(device)
    with torch.no_grad():
        ae_recon = ae(images).view(-1, 1, 28, 28).cpu()
        vae_recon, _, _ = vae(images)
        vae_recon = vae_recon.view(-1, 1, 28, 28).cpu()
    fig, axes = plt.subplots(3, 10, figsize=(15, 5))
    for i in range(10):
        axes[0, i].imshow(images[i].cpu().squeeze(), cmap='gray')
        axes[0, i].axis('off')
        axes[1, i].imshow(ae_recon[i].squeeze(), cmap='gray')
        axes[1, i].axis('off')
        axes[2, i].imshow(vae_recon[i].squeeze(), cmap='gray')
        axes[2, i].axis('off')
    axes[0, 0].set_ylabel('Original', fontsize=12)
    axes[1, 0].set_ylabel('AE', fontsize=12)
    axes[2, 0].set_ylabel('VAE', fontsize=12)
    plt.suptitle("Original vs AE vs VAE Reconstruction", fontsize=14)
    plt.show()
compare_reconstructions(ae, vae, test_loader, device)
```

Output:



AE tends to reconstruct sharper digits. VAE reconstructions are blurrier but more diverse due to its generative nature.

Latent Vector Arithmetic (Bonus)

Interpolation between Digit “3” and “8”

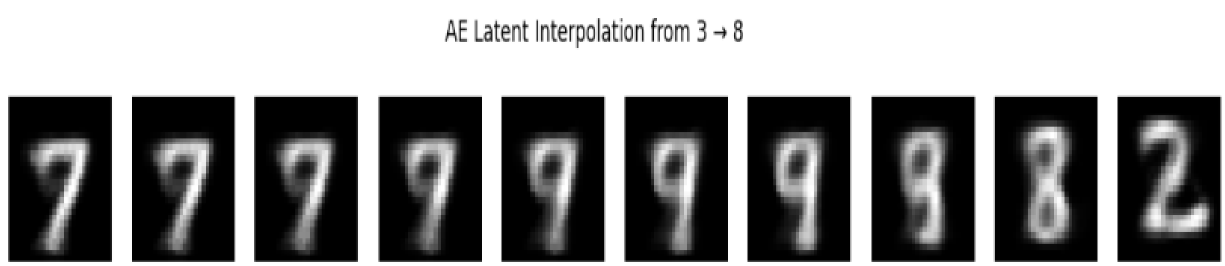
- AE interpolation: Produces mixed or collapsed digits
- VAE interpolation: Smooth morphing between digit shape

AE Interpolation

Code:

```
model.eval()
img3 = test_data[0][0].unsqueeze(0).to(device)  # Digit 3
img8 = test_data[1][0].unsqueeze(0).to(device)  # Digit 8


with torch.no_grad():
    z1 = model.encoder(img3)
    z2 = model.encoder(img8)


    steps = 10
    fig, axs = plt.subplots(1, steps, figsize=(15, 2))
    for i, alpha in enumerate(torch.linspace(0, 1, steps)):
        z_interp = (1 - alpha) * z1 + alpha * z2
        gen_img = model.decoder(z_interp).view(28, 28).cpu().numpy()
        axs[i].imshow(gen_img, cmap='gray')
        axs[i].axis('off')
plt.suptitle("AE Latent Interpolation from 3 → 8")
plt.show()
```

Output:

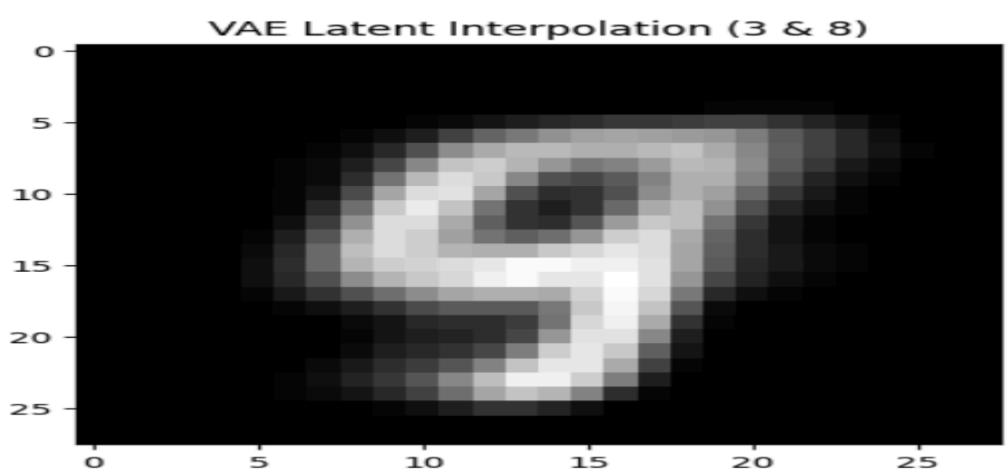


VAE Interpolation

Code:

```
vae.eval()
img3 = test_data[0][0].unsqueeze(0).to(device)  # Digit 3
img8 = test_data[1][0].unsqueeze(0).to(device)  # Digit 8


with torch.no_grad():
    h3 = torch.relu(vae.encoder_fc(img3.view(-1, 784)))
    mu3 = vae.mu(h3)


    h8 = torch.relu(vae.encoder_fc(img8.view(-1, 784)))
    mu8 = vae.mu(h8)


    # Interpolate in latent space
    z_interp_vae = 0.5 * mu3 + 0.5 * mu8
    gen_img_vae = vae.decoder(z_interp_vae).view(28, 28).cpu().numpy()


plt.imshow(gen_img_vae, cmap='gray')
plt.title("VAE Latent Interpolation (3 & 8)")
plt.show()
```

Output:



Key Learnings

- **AE** compresses data into a lower-dimensional space but lacks the ability to generate new samples, as it uses deterministic encoding.

- **VAE** uses probabilistic encoding, allowing it to generate new data, perform smooth interpolations, and learn a continuous latent space.

- **PCA** and **t-SNE** are helpful tools to visualize the latent space and understand how the model clusters and separates data.

- **Latent vector arithmetic** in VAEs shows that the model learns meaningful, semantic relationships in the latent space (e.g., transforming one digit into another).

Future Work

- **Increase Latent Dimensions:**
  Test higher latent sizes (e.g., 10, 20) to improve representation and class separation.

- **Use Complex Datasets:**
  Apply VAEs to datasets like CIFAR-10 and CelebA for richer image learning.

- **Try VAE Variants:**
  Explore β-VAE, Conditional VAE, and Denoising Autoencoders for better control and robustness.

- **Add Attention Mechanisms:**
  Use attention or transformers in the encoder to boost feature learning.

Author Info

Name: Raga Manasa

Roll Number: 228U1A3324

Email: manasaraga2005@gmail.com