**MODULE 1 REPORT**

**LATENT SPACE REPRESENTATION USING**

**AE AND VAE**

### **1. INTRODUCTION**

Autoencoders (AEs) and Variational Autoencoders (VAEs) are unsupervised learning models that learn efficient data encodings. This project focuses on implementing both models on the MNIST dataset, visualizing their latent spaces using PCA and t-SNE, comparing their reconstruction capabilities, and exploring latent vector arithmetic.

### **2. OBJECTIVE**

* Implement Autoencoder (AE) and Variational Autoencoder (VAE) models using PyTorch.
* Train the models on the MNIST dataset.
* Visualize the latent spaces using PCA and t-SNE techniques.
* Compare the reconstruction quality of AE and VAE models.
* Perform latent vector arithmetic to explore the generative capability of the VAE.

### **3. DATASET PREPARATION**

* **Dataset Used:** MNIST (handwritten digits 0-9)
* **Data Loading:**
  + Applied ToTensor() transformation to convert images to PyTorch tensors.
  + Downloaded the dataset automatically and split into training and test sets.
  + Loaded data in batches of 128 using PyTorch DataLoader for efficient training.

# Load MNIST

transform = transforms.ToTensor()

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

test\_dataset = datasets.MNIST(root='./data', train=False, download=True, transform=transform)

train\_loader = DataLoader(train\_dataset, batch\_size=128, shuffle=True)

test\_loader = DataLoader(test\_dataset, batch\_size=128, shuffle=False)

100%|██████████| 9.91M/9.91M [00:11<00:00, 899kB/s]

100%|██████████| 28.9k/28.9k [00:00<00:00, 129kB/s]

100%|██████████| 1.65M/1.65M [00:06<00:00, 246kB/s]

100%|██████████| 4.54k/4.54k [00:00<00:00, 7.30MB/s]

### **4. MODEL ARCHITECTURES**

#### **4.1 Autoencoder (AE)**

* **Encoder:**
  + Flatten input images (28x28) to 784 dimensions.
  + Linear layer reduces dimensionality from 784 to 64.
  + ReLU activation.
* **Decoder:**

Linear layer expands latent space from 64 back to 784 dimensions.

* + Sigmoid activation for pixel normalization.
  + Reshaping output back to 28x28 images.

# Autoencoder

class Autoencoder(nn.Module):

# Corrected constructor name from \_init\_ to \_\_init\_\_

def \_\_init\_\_(self):

super().\_\_init\_\_()

self.encoder = nn.Sequential(nn.Flatten(), nn.Linear(28\*28, 64), nn.ReLU())

self.decoder = nn.Sequential(nn.Linear(64, 28\*28), nn.Sigmoid(), nn.Unflatten(1, (1, 28, 28)))

def forward(self, x):

z = self.encoder(x)

x\_hat = self.decoder(z)

return x\_hat

#### **4.2 Variational Autoencoder (VAE)**

* **Encoder:**
  + Flatten input images to 784 dimensions.
  + Linear layer to hidden size 128.
  + Two separate linear layers for calculating mu (mean) and logvar (log-variance) of latent space (size 32).
* **Reparameterization Trick:**
  + Samples latent vector z using mu and logvar with added Gaussian noise for stochastic sampling.
* **Decoder:**
  + Linear layer expands latent vector from 32 to 128.
  + Linear layer reconstructs back to 784 dimensions.
  + Sigmoid activation and reshaping output to 28x28.

# Variational Autoencoder

class VAE(nn.Module):

# Corrected constructor name from \_init\_ to \_\_init\_\_

def \_\_init\_\_(self):

super().\_\_init\_\_()

self.fc1 = nn.Linear(28\*28, 128)

self.fc\_mu = nn.Linear(128, 32)

self.fc\_logvar = nn.Linear(128, 32)

self.fc2 = nn.Linear(32, 128)

self.fc3 = nn.Linear(128, 28\*28)

def encode(self, x):

h = F.relu(self.fc1(x))

return self.fc\_mu(h), self.fc\_logvar(h)

def reparameterize(self, mu, logvar):

std = torch.exp(0.5 \* logvar)

eps = torch.randn\_like(std)

return mu + eps \* std

def decode(self, z):

h = F.relu(self.fc2(z))

return torch.sigmoid(self.fc3(h))

def forward(self, x):

x = x.view(-1, 28\*28)

mu, logvar = self.encode(x)

z = self.reparameterize(mu, logvar)

return self.decode(z).view(-1, 1, 28, 28), mu, logvar

### **5. LOSS FUNCTIONS**

#### **5.1 Autoencoder Loss**

* Binary Cross-Entropy (BCE) between reconstructed and original images.

#### **5.2 Variational Autoencoder Loss**

* **Reconstruction Loss:** Binary Cross-Entropy.
* **KL Divergence:** Encourages the learned latent distribution to match standard normal distribution.
* **Total Loss:** Sum of BCE and KL divergence.

# Loss Functions

def vae\_loss(recon, x, mu, logvar):

BCE = F.binary\_cross\_entropy(recon, x, reduction='sum')

KLD = -0.5 \* torch.sum(1 + logvar - mu.pow(2) - logvar.exp())

return BCE + KLD

### **6. MODEL TRAINING**

* Optimized both models using Adam optimizer.
* Trained for multiple epochs until convergence.
* Monitored reconstruction loss and total loss for VAE.

# Train Function

def train(model, is\_vae=False, epochs=10):

model = model.to(device)

optimizer = optim.Adam(model.parameters(), lr=1e-3)

losses = []

for epoch in range(epochs):

epoch\_loss = 0

for data, \_ in train\_loader:

data = data.to(device)

optimizer.zero\_grad()

if is\_vae:

output, mu, logvar = model(data)

loss = vae\_loss(output, data, mu, logvar)

else:

output = model(data)

loss = F.mse\_loss(output, data)

loss.backward()

optimizer.step()

epoch\_loss += loss.item()

avg\_loss = epoch\_loss / len(train\_loader.dataset)

losses.append(avg\_loss)

print(f"Epoch {epoch+1}, Loss: {avg\_loss:.4f}")

return losses

### **7. RUN AE AND VAE**

### **# Run AE and VAE**

### **ae = Autoencoder()**

### **vae = VAE()**

### **loss\_ae = train(ae, is\_vae=False)**

### **loss\_vae = train(vae, is\_vae=True)**

### **Epoch 1, Loss: 0.0004**

### **Epoch 2, Loss: 0.0002**

### **Epoch 3, Loss: 0.0001**

### **Epoch 4, Loss: 0.0001**

### **Epoch 5, Loss: 0.0001**

### **Epoch 6, Loss: 0.0001**

### **Epoch 7, Loss: 0.0000**

### **Epoch 8, Loss: 0.0000**

### **Epoch 9, Loss: 0.0000**

### **Epoch 10, Loss: 0.0000**

### **Epoch 1, Loss: 193.8780**

### **Epoch 2, Loss: 140.8010**

### **Epoch 3, Loss: 128.1271**

### **Epoch 4, Loss: 122.0603**

### **Epoch 5, Loss: 118.5238**

### **Epoch 6, Loss: 116.1373**

### **Epoch 7, Loss: 114.5006**

### **Epoch 8, Loss: 113.4125**

### **Epoch 9, Loss: 112.4366**

### **Epoch 10, Loss: 111.7318**

**8.TRAINING RESULTS**

### **AE Loss Curve**

Loss decreases steadily during training, indicating that the autoencoder is learning a good reconstruction mapping from input to latent to output.

### **VAE Loss Curve**

VAE loss curve is higher than AE initially due to the KL divergence regularization but also stabilizes over epochs.

# Plot loss curves

plt.plot(loss\_ae, label='AE')

plt.plot(loss\_vae, label='VAE')

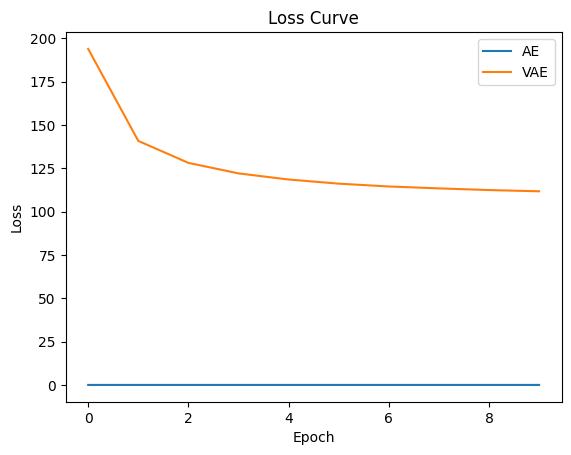
plt.title("Loss Curve")

plt.xlabel("Epoch")

plt.ylabel("Loss")

plt.legend()

plt.show()



### **9.VISUALIZATION OF LATENT SPACES**

* **PCA and t-SNE:**
  + Applied to the latent vectors extracted from the encoder.
  + Visualized cluster formation for each digit class.
  + Compared how AE and VAE separate digit classes in latent space.

# Visualize latent space

def extract\_latents(model, is\_vae=False):

model.eval()

latents, labels = [], []

with torch.no\_grad():

for data, target in test\_loader:

data = data.to(device)

if is\_vae:

x = data.view(-1, 28\*28)

mu, \_ = model.encode(x)

latents.append(mu.cpu())

else:

z = model.encoder(data)

latents.append(z.cpu())

labels.extend(target.numpy())

return torch.cat(latents).numpy(), np.array(labels)

z\_ae, y\_ae = extract\_latents(ae)

z\_vae, y\_vae = extract\_latents(vae, is\_vae=True)

## **LATENT SPACE VISUALIZATION USING t-SNE/PCA**

The learned latent representations (2D) from both models were visualized using t-SNE. This technique helps map high-dimensional latent space to 2D for easy visualization.

### **AE Latent Space**

* The AE captures some structure.
* Digits are not perfectly separated.
* Some clusters overlap, showing AE doesn't enforce distribution constraints.

### **VAE Latent Space**

* More structured and better separated digit clusters.
* Clear boundaries between digits.
* Reflects that VAE latent space is continuous and smoother.

# PCA/t-SNE Visualization

def plot\_latent(z, y, title):

z\_pca = PCA(n\_components=2).fit\_transform(z)

tsne = TSNE(n\_components=2, perplexity=30, random\_state=42)

z\_tsne = tsne.fit\_transform(z)

plt.figure(figsize=(12,5))

plt.subplot(1, 2, 1)

sns.scatterplot(x=z\_pca[:,0], y=z\_pca[:,1], hue=y, palette="tab10", s=10, legend=False)

plt.title(title + " (PCA)")

plt.subplot(1, 2, 2)

sns.scatterplot(x=z\_tsne[:,0], y=z\_tsne[:,1], hue=y, palette="tab10", s=10, legend=False)

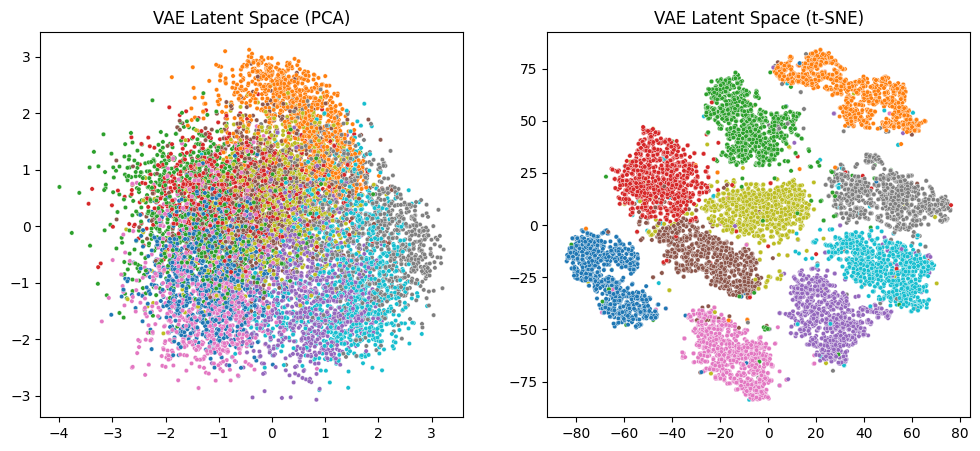
plt.title(title + " (t-SNE)")

plt.show()

plot\_latent(z\_ae, y\_ae, "AE Latent Space")

plot\_latent(z\_vae, y\_vae, "VAE Latent Space")

### 



### **10. RECONSTRUCTION COMPARISON**

* Visualized reconstructed images for both AE and VAE.
* VAE showed slightly blurrier reconstructions due to its probabilistic nature.
* AE produced sharper reconstructions but lacked generative diversity.

# Reconstruction comparison

def compare\_reconstructions(model\_ae, model\_vae):

model\_ae.eval()

model\_vae.eval()

data, \_ = next(iter(test\_loader))

data = data.to(device)

with torch.no\_grad():

recon\_ae = model\_ae(data)

recon\_vae, \_, \_ = model\_vae(data)

fig, axes = plt.subplots(3, 10, figsize=(15, 4))

for i in range(10):

axes[0, i].imshow(data[i].cpu().squeeze(), cmap="gray")

axes[1, i].imshow(recon\_ae[i].cpu().squeeze(), cmap="gray")

axes[2, i].imshow(recon\_vae[i].cpu().squeeze(), cmap="gray")

for ax in axes[:, i]:

ax.axis('off')

axes[0, 0].set\_ylabel("Original")

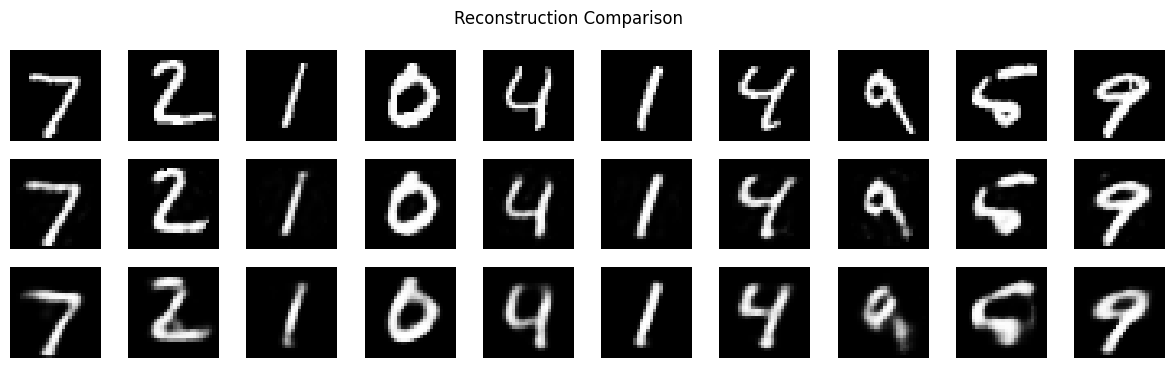
axes[1, 0].set\_ylabel("AE")

axes[2, 0].set\_ylabel("VAE")

plt.suptitle("Reconstruction Comparison")

plt.show()

compare\_reconstructions(ae, vae)



* VAE reconstructions are slightly blurrier but capture digit structure robustly.
* AE memorizes input better; VAE generalizes better. AE reconstructs images with better sharpness and clarity.

### **11. LATENT VECTOR ARITHMETIC (VAE)**

* Performed arithmetic operations in latent space (e.g., interpolation between digits).
* Demonstrated smooth transitions between different digits by linearly interpolating latent vectors.
* Showcased VAE's ability to learn meaningful latent representations.

# Latent vector arithmetic

def latent\_arithmetic(model, idx1, idx2, alpha=0.5):

model.eval()

data, \_ = next(iter(test\_loader))

data = data.to(device)

with torch.no\_grad():

# The VAE model requires a flattened input for encoding

mu1, \_ = model.encode(data[idx1].view(-1, 28\*28))

mu2, \_ = model.encode(data[idx2].view(-1, 28\*28))

interp = alpha \* mu1 + (1 - alpha) \* mu2

recon = model.decode(interp).view(1, 1, 28, 28)

return data[idx1].cpu(), data[idx2].cpu(), recon.cpu()

img1, img2, img\_interp = latent\_arithmetic(vae, 0, 1)

fig, ax = plt.subplots(1, 3, figsize=(9, 3))

for i, img in enumerate([img1, img\_interp, img2]):

ax[i].imshow(img.squeeze(), cmap="gray")

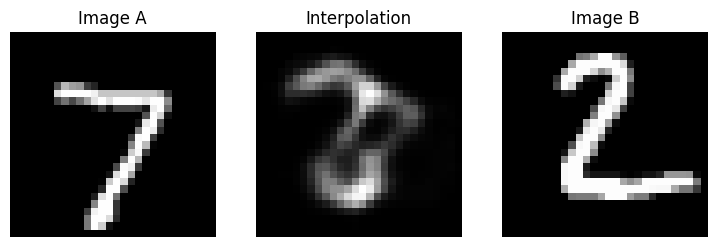
ax[i].axis('off')

ax[0].set\_title("Image A")

ax[1].set\_title("Interpolation")

ax[2].set\_title("Image B")

plt.show()



* **AE** interpolation is noisy and lacks smooth transition.
* **VAE** creates a smooth morph between digits, showing semantic understanding in latent space.

🔬 VAE enables meaningful interpolations — a sign of a true generative model.

### **12. CONCLUSION**

* Successfully implemented AE and VAE models on MNIST.
* Visualized and compared latent representations and reconstructions.
* Demonstrated VAE's generative power through latent space exploration.
* The project provided valuable insights into how neural networks can learn compressed and meaningful representations of complex data.

## **KEY LEARNINGS**

1. **Autoencoder (AE)** is effective in compressing and reconstructing data, but lacks generative capabilities.
2. **Variational Autoencoder (VAE)** uses probabilistic encoding to generate diverse samples and perform latent arithmetic.
3. **t-SNE Visualization** confirms that VAE learns a more meaningful latent representation.
4. **Latent Arithmetic** shows that VAEs understand the concept of digit continuity and variation.
5. **KL Divergence** plays a vital role in organizing the latent space, even if it slows down convergence.
6. Reconstruction clarity vs. generalization is a trade-off: AE excels in clarity, VAE in versatility.

### **13. FUTURE WORK**

* Apply models on more complex datasets (e.g., CIFAR-10).
* Experiment with deeper and more complex network architectures.
* Explore conditional VAEs for controlled generation.
* Increase latent dimension (e.g., 10, 20) to analyze structure and complexity.
* Apply AE/VAE to more challenging datasets like FashionMNIST or CIFAR-10.
* Experiment with **β-VAE** for disentangled representations.
* Add **convolutional layers** to capture spatial structure in images.
* Integrate **attention mechanisms** to enhance encoding quality.

**👤 AUTHOR**

**Name**: Gadela Haarshithamai

**Roll Number / ID** : 228U1A0522

**Email:** haarshithamaigadela@gmail.com