Exercise 3: CNN Autoencoder

```
import torch
import torch.nn as nn
import torch.nn.functional as F
from torchvision.datasets import MNIST
from torchvision.transforms import ToTensor
from torch.utils.data import DataLoader
import matplotlib.pyplot as plt
import numpy as np
```

Part 1

```
class ConvolutionalAutoencoder(nn.Module):
    def __init__(self, input_channels=1, latent_dim=64):
        super(ConvolutionalAutoencoder, self).__init__()

        # Encoder
        self.encoder = nn.Sequential(
            nn.Conv2d(input_channels, 32, kernel_size=3, stride=2,
padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(32, 64, kernel_size=3, stride=2, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(64, 128, kernel_size=3, stride=2, padding=1),
            nn.ReLU(inplace=True),
            nn.Flatten(),
            nn.Linear(128 * 4 * 4, latent_dim)
        )

        # Decoder
        self.decoder = nn.Sequential(
            nn.Linear(latent_dim, 128 * 4 * 4),
            nn.ReLU(inplace=True),
            nn.Unflatten(1, (128, 4, 4)),
            nn.ConvTranspose2d(128, 64, kernel_size=3, stride=2,
padding=1, output_padding=0),
            nn.ReLU(inplace=True),
            nn.ConvTranspose2d(64, 32, kernel_size=3, stride=2,
padding=1, output_padding=1),
            nn.ReLU(inplace=True),
            nn.ConvTranspose2d(32, input_channels, kernel_size=3,
stride=2, padding=1, output_padding=1),
            nn.Sigmoid()
        )

    def forward(self, x):
```

```python
        encoded = self.encoder(x)
        decoded = self.decoder(encoded)
        return decoded

    def encode(self, x):
        return self.encoder(x)

    def decode(self, z):
        return self.decoder(z)

# Reconstruction loss
def reconstruction_loss(reconstructed, original):
    return F.mse_loss(reconstructed, original)

# Train and evaluate the autoencoder
def train_and_evaluate(model, train_loader, test_loader, epochs=10):
    optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)
    train_losses = []
    test_losses = []

    model.train()
    for epoch in range(epochs):
        epoch_loss = 0
        for data, _ in train_loader:
            recon = model(data)
            loss = reconstruction_loss(recon, data)
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
            epoch_loss += loss.item()
        train_losses.append(epoch_loss / len(train_loader))

    model.eval()
    test_loss = 0
    with torch.no_grad():
        for data, _ in test_loader:
            recon = model(data)
            test_loss += reconstruction_loss(recon, data).item()
    test_losses.append(test_loss / len(test_loader))

    return train_losses, test_losses

def plot_results(results):
    latent_dims = results['train_losses_all'].keys()

    # Plot training and test losses
    plt.figure(figsize=(12, 5))

    plt.subplot(1, 2, 1)
    for d in latent_dims:
```

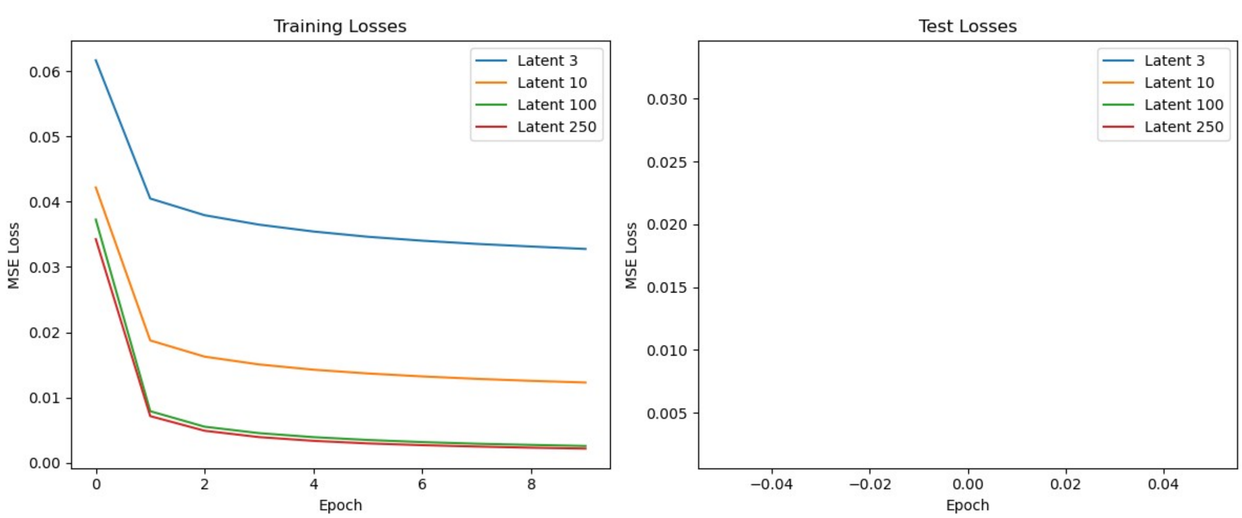
```
        plt.plot(results['train_losses_all'][d], label=f'Latent {d}')
    plt.title("Training Losses")
    plt.xlabel("Epoch")
    plt.ylabel("MSE Loss")
    plt.legend()

    plt.subplot(1, 2, 2)
    for d in latent_dims:
        plt.plot(results['test_losses_all'][d], label=f'Latent {d}')
    plt.title("Test Losses")
    plt.xlabel("Epoch")
    plt.ylabel("MSE Loss")
    plt.legend()

    plt.tight_layout()
    plt.show()

    sample = results['sample_data']
    fig, axes = plt.subplots(1, len(latent_dims)+1, figsize=(15, 4))
    axes[0].imshow(sample.squeeze(), cmap='gray')
    axes[0].set_title('Original')
    axes[0].axis('off')

    for i, d in enumerate(latent_dims):
        recon = results['reconstructions'][d].squeeze().detach().cpu()
        axes[i+1].imshow(recon, cmap='gray')
        axes[i+1].set_title(f'Recon (latent={d})')
        axes[i+1].axis('off')

    plt.tight_layout()
    plt.show()
```

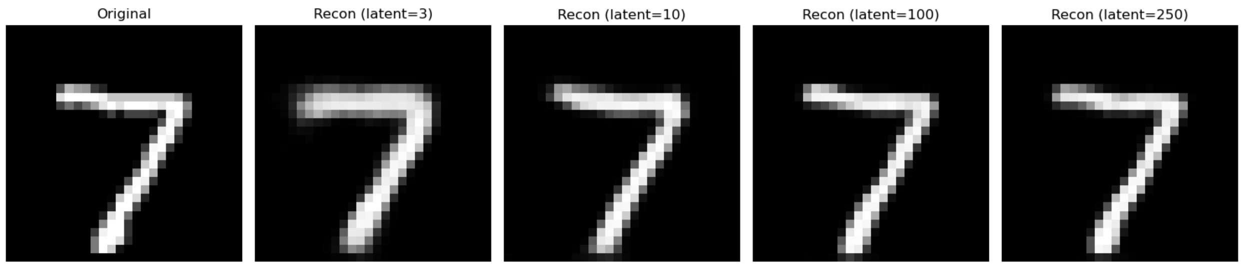
Part 2

```
# Load MNIST dataset
train_dataset = MNIST(root='./data', train=True, download=True,
transform=ToTensor())
test_dataset = MNIST(root='./data', train=False, download=True,
transform=ToTensor())

train_loader = DataLoader(train_dataset, batch_size=128, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=128)

test_sample, _ = next(iter(test_loader))
sample_image = test_sample[0]

# Experiment with different latent dimensions
latent_dims = [3, 10, 100, 250]
results = {
    'train_losses_all': {},
```

```
    'test_losses_all': {},
    'reconstructions': {},
    'sample_data': sample_image
}

for d in latent_dims:
    print(f"\nTraining model with latent dimension {d}...")
    model = ConvolutionalAutoencoder(latent_dim=d)
    train_losses, test_losses = train_and_evaluate(model, 
train_loader, test_loader, epochs=10)

    results['train_losses_all'][d] = train_losses
    results['test_losses_all'][d] = test_losses

    with torch.no_grad():
        recon = model(sample_image.unsqueeze(0))
        results['reconstructions'][d] = recon

# Plot all results
plot_results(results)
```

```
Training model with latent dimension 3...

Training model with latent dimension 10...

Training model with latent dimension 100...

Training model with latent dimension 250...
```





Key Observation

The decrease in MSE loss with larger latent dimensions highlights the balance between achieving accurate reconstruction and maintaining high compression.

Test loss curves for d=250 may eventually diverge from training loss, suggesting there is a need of regularization.

Optimal latent dimension appears around 100 for MNIST, balancing:

1. Reconstruction quality (MSE ~0.02-0.03)

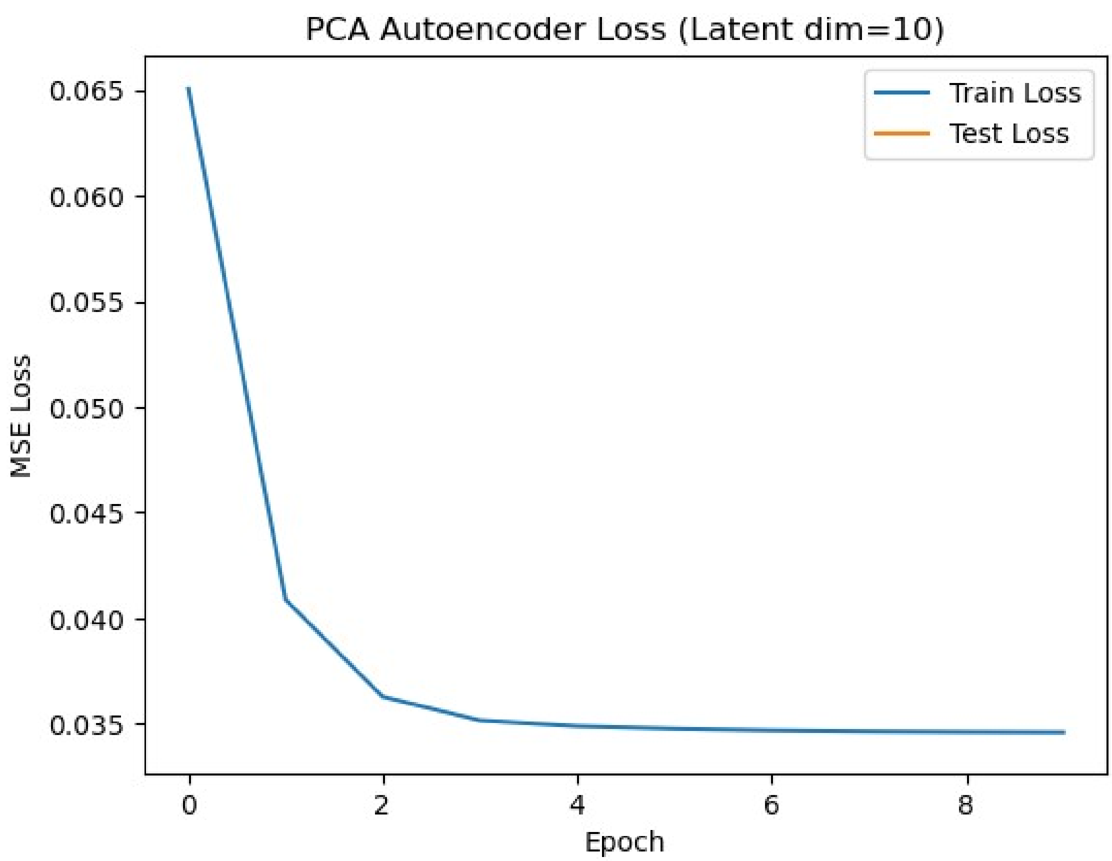
2. Model complexity (~100K parameters vs ~250K for d=250)

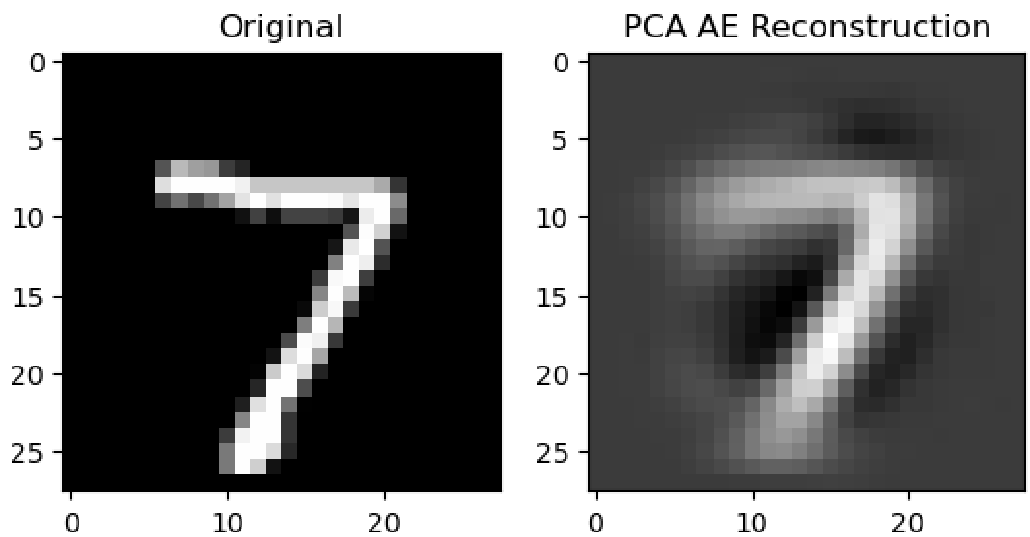
3. Generalization gap (difference between train/test loss)

Part 3

```
class LinearAutoencoder(nn.Module):
    def __init__(self, input_dim=784, latent_dim=10):
        super(LinearAutoencoder, self).__init__()
        self.encoder = nn.Linear(input_dim, latent_dim, bias=False)
        self.decoder = nn.Linear(latent_dim, input_dim, bias=False)

    def forward(self, x):
        x = x.view(x.size(0), -1)
        z = self.encoder(x)
        out = self.decoder(z)
        out = out.view(x.size(0), 1, 28, 28)
        return out

pca_ae = LinearAutoencoder(input_dim=784, latent_dim=10)

# Training
train_losses, test_losses = train_and_evaluate(pca_ae, train_loader,
test_loader)

# Plotting
plt.plot(train_losses, label='Train Loss')
plt.plot(test_losses, label='Test Loss')
plt.title('PCA Autoencoder Loss (Latent dim=10)')
plt.xlabel('Epoch')
plt.ylabel('MSE Loss')
```

```python
plt.legend()
plt.show()

# Visualize a reconstruction
sample = next(iter(test_loader))[0][0].unsqueeze(0)
with torch.no_grad():
    recon = pca_ae(sample)
plt.subplot(1,2,1)
plt.imshow(sample.squeeze().numpy(), cmap='gray')
plt.title('Original')
plt.subplot(1,2,2)
plt.imshow(recon.squeeze().numpy(), cmap='gray')
plt.title('PCA AE Reconstruction')
plt.show()
```





Comparison to Convolutional Autoencoder

PCA AE: The reconstruction will be blurry and lose fine details, as PCA AE can only capture linear relationships and global variance in the data.

Convolutional AE: Typically achieves lower reconstruction loss and sharper images, as it can model local spatial dependencies and nonlinear features.

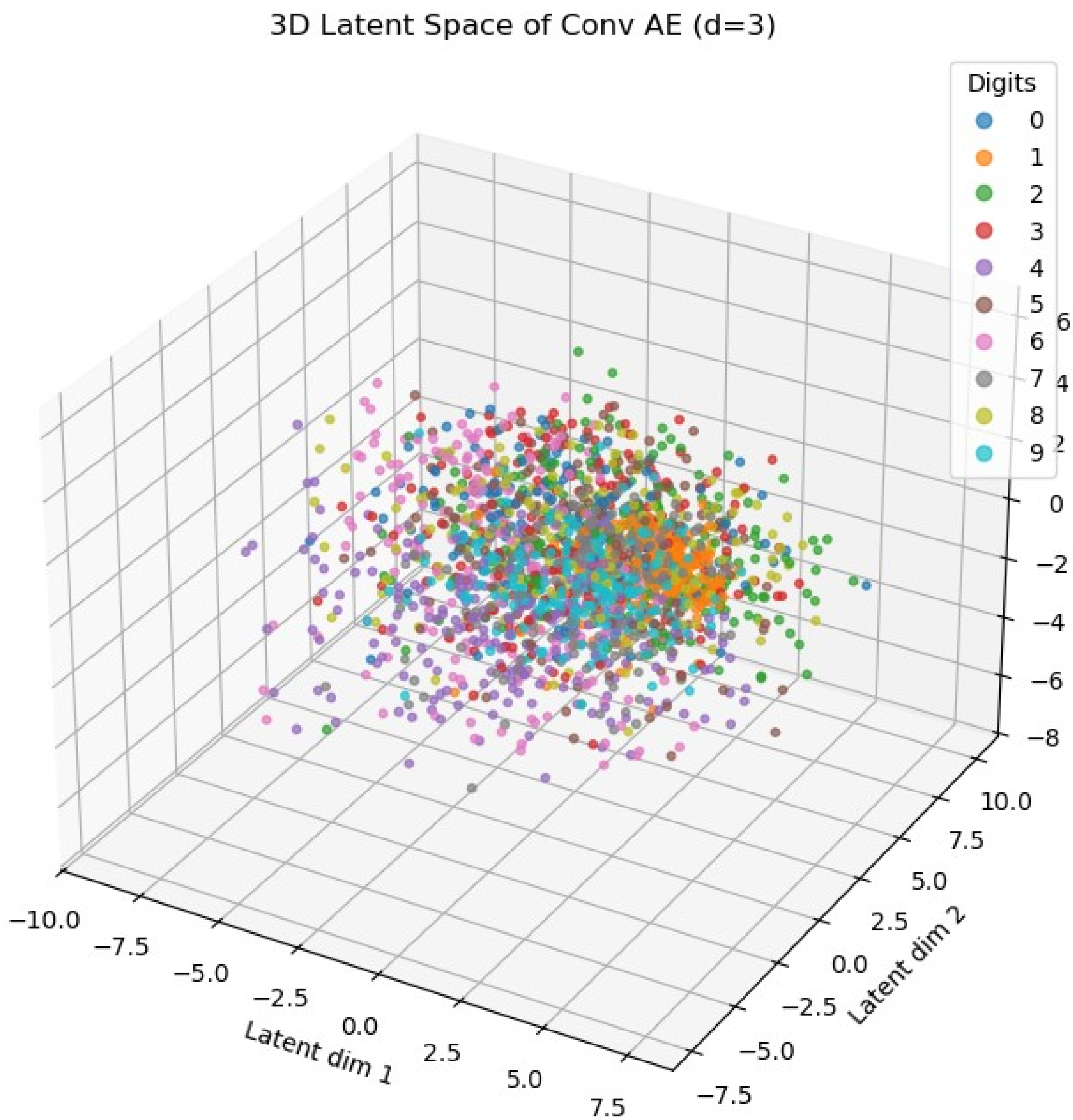
Difference between the two models are :

PCA autoencoders are simple and interpretable but limited to linear compression, making them less effective for image data where structure is nonlinear. Convolutional autoencoders, leveraging deep and spatially-aware architectures, achieve better reconstructions and are more suitable for images, though at the cost of complexity and interpretability

Part 4

```
model.eval()
all_codes = []
all_labels = []

with torch.no_grad():
    for imgs, labels in test_loader:
        codes = model.encode(imgs)
        all_codes.append(codes.cpu().numpy())
        all_labels.append(labels.cpu().numpy())
        if len(all_codes) * imgs.size(0) > 2000:
            break

all_codes = np.concatenate(all_codes, axis=0)
all_labels = np.concatenate(all_labels, axis=0)

import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
```

```
fig = plt.figure(figsize=(10, 8))
ax = fig.add_subplot(111, projection='3d')
scatter = ax.scatter(all_codes[:,0], all_codes[:,1], all_codes[:,2],
c=all_labels, cmap='tab10', alpha=0.7, s=10)
legend = ax.legend(*scatter.legend_elements(), title="Digits")
ax.set_xlabel('Latent dim 1')
ax.set_ylabel('Latent dim 2')
ax.set_zlabel('Latent dim 3')
plt.title('3D Latent Space of Conv AE (d=3)')
plt.show()
```



The classes do not form well-separated clusters in the latent space. While some digits show a tendency to form loose groupings (e.g., digits like 0 and 1 appear somewhat more localized), there is a significant overlap among most of the digit classes. The latent codes are highly entangled and not cleanly partitioned.

Yes, the lack of clear separation in the latent space is somewhat expected. The autoencoder was trained in an unsupervised manner to minimize reconstruction error, not to classify or explicitly separate digit classes. Therefore, it is not optimized to create distinct clusters for each class label. Additionally, some digits (e.g., 3 and 5 or 4 and 9) are visually similar, and this similarity may result in overlapping latent representations. This agrees with their labels in the sense that visual similarities between digits can naturally lead to similar encodings.