BLG 527E - Machine Learning Homework 3

Name: Hasan F. Durkaya

ID: 504241526

Date: 11.01.2025

		Question	Total
Grade	Max	10	10
	Expected	9.5	9.5

Architecture and Hyperparameters:

```
class Encoder(nn.Module):
    def __init__(self, input_dim, latent_dim):
        super(Encoder, self).__init__()
        self.encoder = nn.Sequential(
            nn.Linear(input_dim, 512),
            nn.ReLU(),
            nn.Linear(512, 256),
            nn.ReLU(),
            nn.Linear(256, latent_dim),
            nn.ReLU()
        )

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

class Decoder(nn.Module):
    def __init__(self, latent_dim, output_dim):
        super(Decoder, self).__init__()
        self.decoder = nn.Sequential(
            nn.Linear(latent_dim, 256),
            nn.ReLU(),
            nn.Linear(256, 512),
            nn.ReLU(),
            nn.Linear(512, output_dim),
            nn.Sigmoid()
        )

    def forward(self, x):
        return self.decoder(x)

class Autoencoder(nn.Module):
    def __init__(self, input_dim, latent_dim):
        super(Autoencoder, self).__init__()
        self.encoder = Encoder(input_dim, latent_dim)
        self.decoder = Decoder(latent_dim, input_dim)

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

The **Encoder** compresses the input data into a smaller, more meaningful representation “the latent space”. It starts with the input dimension (input_dim) and progressively reduces its size through a series of fully connected layers. Each layer is followed by a ReLU activation function, which introduces non-linearity, enabling the model to learn complex patterns. The final output of the Encoder is a latent representation of size latent_dim.

The **Decoder**, reconstructs the original input from the compressed latent representation. It mirrors the Encoder's structure n reverse, expanding the data back to its original formn. Decoder also uses fully connected layers and ReLU activations, but its final layer applies a Sigmoid activation function. This ensures that the reconstructed output values are between 0 and 1.

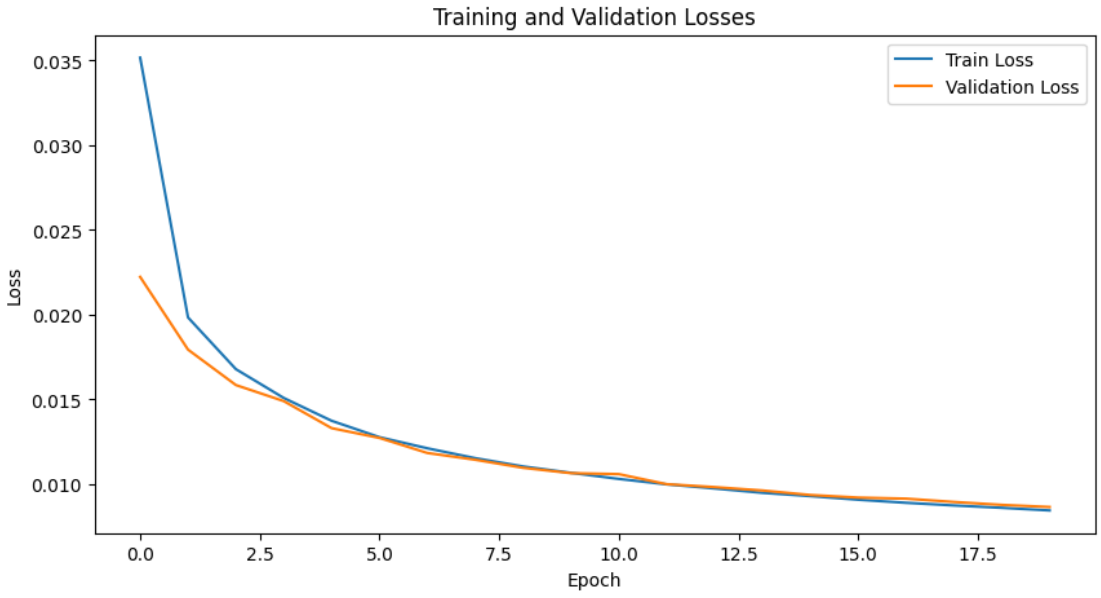
The **Autoencoder** combines the Encoder and Decoder into a single model. When data is passed through the Autoencoder, it first flows through the Encoder, which compresses it into the latent space. This compressed representation is then passed to the Decoder, which reconstructs the input.

```
# Set random seed for reproducibility.
torch.manual_seed(42)

# Hyperparameters.
EPOCHS = 20
BATCH_SIZE = 128
LEARNING_RATE = 0.001
LATENT_DIM = 64
INPUT_DIM = 28 * 28
```

Hyperparameters are set as shown and seed for reproducibility.

Training process and Final Reconsturcitons Loss:

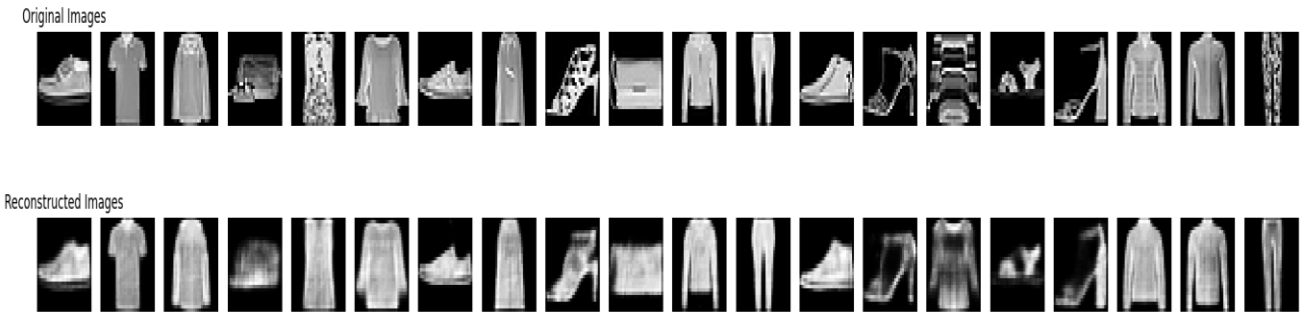
Epoch: 20, Train Loss: 0.008457, Validation Loss: 0.008660



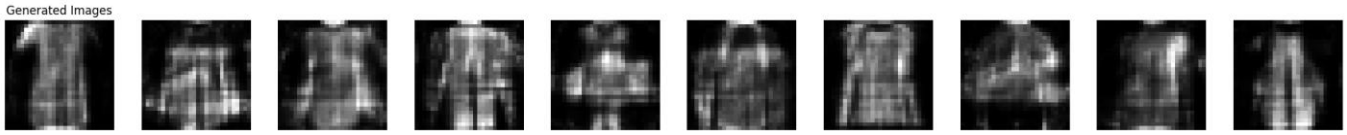
```python
#Function to train the model.
def train_model(model, train_loader, test_loader, criterion, optimizer, epochs):

    #Set the model to training mode. Device is set to GPU if available.
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    model = model.to(device)

    train_losses = []
    test_losses = []

    # Training.
    for epoch in range(epochs):

        model.train()
        train_loss = 0

        for batch_idx, (data, _) in enumerate(train_loader):
            data = data.view(data.size(0), -1).to(device)
            optimizer.zero_grad()
            reconstruction = model(data)
            loss = criterion(reconstruction, data)
            loss.backward()
            optimizer.step()
            train_loss += loss.item()

        train_loss /= len(train_loader)
        train_losses.append(train_loss)

        #Validation.
        model.eval()
        test_loss = 0
        with torch.no_grad():
            for data, _ in test_loader:
                data = data.view(data.size(0), -1).to(device)
                reconstruction = model(data)
                loss = criterion(reconstruction, data)
                test_loss += loss.item()

        test_loss /= len(test_loader)
        test_losses.append(test_loss)

        print(f'Epoch: {epoch+1}, Train Loss: {train_loss:.6f}, Validation Loss: {test_loss:.6f}')

    return train_losses, test_losses
```

The train_model function trains autoencoder over multiple epochs, tracking training and validation losses. It first moves the model to the appropriate device (GPU). During each epoch, it processes batches of data, computes reconsturction losses, and updates the model during training. In the validation phase, the model's performance is evaluated without weight updates. The function returns the average training and validation recınstruction losses for each epoch.

Examples :

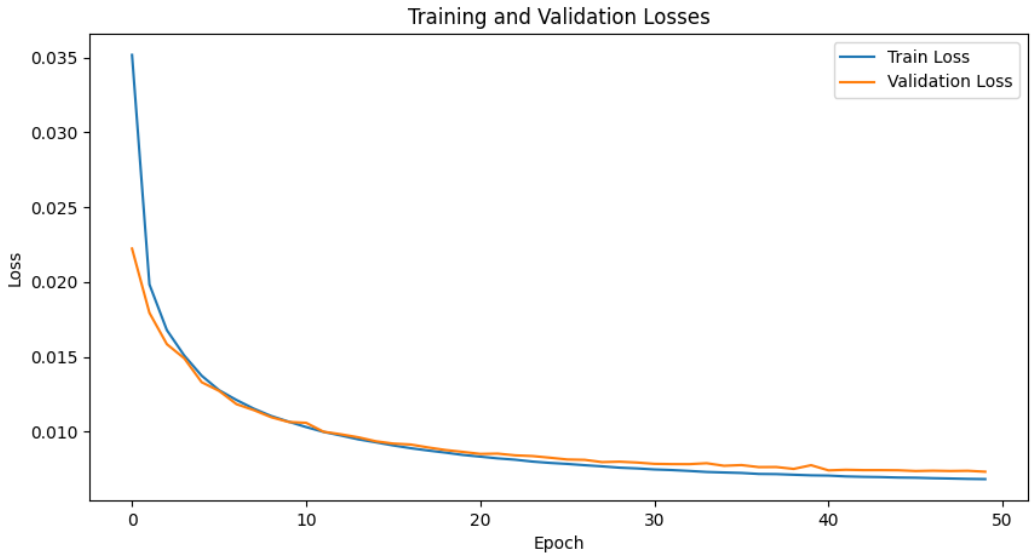


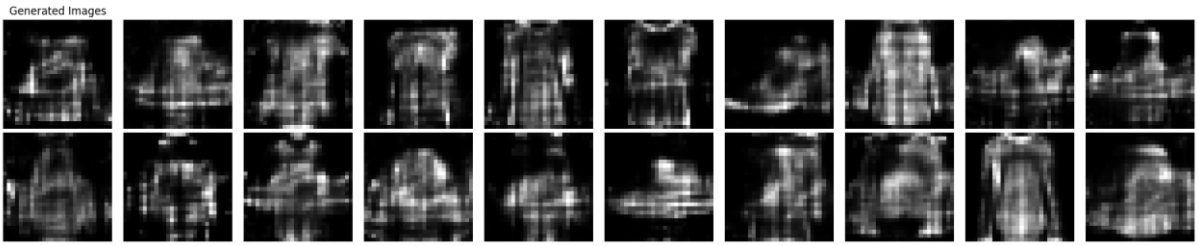
Samples:



Generated images are cloth images it is easy to observe but details are not very good. Contrast is gray rather than clear black and whites. There are only upper body clothes it can be because of fashion mnist dataset. It is mainly focusing on upperbody clothes. There are lack fo details in the generated images. This can be because of the models capacity, maybe the model is not complex enough. Training more epoch may result in better performance. To understand effect of the training more I also trained more epochs. I have trained for 20 epoch at first. Above images are generated from the 20 epoch train. Also below results are from 50 epch train. As a result model is doing what it supposed to do but it can be improved.

Epoch: 50, Train Loss: 0.006841, Validation Loss: 0.007336





With 50 epoch train, Train results seem quite better. Loss is very low on the other hand, generated images from random vectors are not better than 20 epoch train. Therefore we can say model can be improved with a more complex architecture and some normalization operations for solving overfitting issues.

The code block:

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

# Set random seed for reproducibility.
torch.manual_seed(42)

# Hyperparameters.
EPOCHS = 50
BATCH_SIZE = 128
LEARNING_RATE = 0.001
LATENT_DIM = 64
INPUT_DIM = 28 * 28


class Encoder(nn.Module):
    def __init__(self, input_dim, latent_dim):
        super(Encoder, self).__init__()
        self.encoder = nn.Sequential(
            nn.Linear(input_dim, 512),
            nn.ReLU(),
            nn.Linear(512, 256),
            nn.ReLU(),
            nn.Linear(256, latent_dim),
            nn.ReLU()
        )

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

```
class Decoder(nn.Module):
    def __init__(self, latent_dim, output_dim):
        super(Decoder, self).__init__()
        self.decoder = nn.Sequential(
            nn.Linear(latent_dim, 256),
            nn.ReLU(),
            nn.Linear(256, 512),
            nn.ReLU(),
            nn.Linear(512, output_dim),
            nn.Sigmoid()
        )

    def forward(self, x):
        return self.decoder(x)

class Autoencoder(nn.Module):
    def __init__(self, input_dim, latent_dim):
        super(Autoencoder, self).__init__()
        self.encoder = Encoder(input_dim, latent_dim)
        self.decoder = Decoder(latent_dim, input_dim)

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


#Function to get the data.
def load_data():
    transform = transforms.Compose([
        transforms.ToTensor()
    ])

    # Training and test datasets.
    train_dataset = datasets.FashionMNIST(
        root='./data',
        train=True,
        download=True,
        transform=transform
    )

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

    #Data loaders.
```

```
    train_loader = DataLoader(
        train_dataset,
        batch_size=BATCH_SIZE,
        shuffle=True
    )

    test_loader = DataLoader(
        test_dataset,
        batch_size=BATCH_SIZE,
        shuffle=True
    )

    return train_loader, test_loader


#Function to train the model.
def train_model(model, train_loader, test_loader, criterion, optimizer,
epochs):

    #Set the model to training mode. Device is set to GPU if available.
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    model = model.to(device)

    train_losses = []
    test_losses = []

    # Training.
    for epoch in range(epochs):

        model.train()
        train_loss = 0

        for batch_idx, (data, _) in enumerate(train_loader):
            data = data.view(data.size(0), -1).to(device)
            optimizer.zero_grad()
            reconstruction = model(data)
            loss = criterion(reconstruction, data)
            loss.backward()
            optimizer.step()
            train_loss += loss.item()

        train_loss /= len(train_loader)
        train_losses.append(train_loss)

        #Validation.
        model.eval()
        test_loss = 0
        with torch.no_grad():
            for data, _ in test_loader:
```

```
                data = data.view(data.size(0), -1).to(device)
                reconstruction = model(data)
                loss = criterion(reconstruction, data)
                test_loss += loss.item()

        test_loss /= len(test_loader)
        test_losses.append(test_loss)

        print(f'Epoch: {epoch+1}, Train Loss: {train_loss:.6f}, Validation
Loss: {test_loss:.6f}')

    return train_losses, test_losses



def visualize_reconstruction(model, test_loader, num_images=20):

    #Set the model to evaluation mode. Device is set to GPU if available.
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    model.eval()

    with torch.no_grad():
        data, _ = next(iter(test_loader))
        data = data[:num_images]
        data = data.view(data.size(0), -1).to(device)
        reconstruction = model(data)

        #Convert to numpy for visualization.
        data = data.cpu().view(-1, 28, 28)
        reconstruction = reconstruction.cpu().view(-1, 28, 28)

        plt.figure(figsize=(20, 4))
        for i in range(num_images):
            #riginal images.
            plt.subplot(2, num_images, i + 1)
            plt.imshow(data[i], cmap='gray')
            plt.axis('off')
            if i == 0:
                plt.title('Original Images')

            #Reconstructed images.
            plt.subplot(2, num_images, i + num_images + 1)
            plt.imshow(reconstruction[i], cmap='gray')
            plt.axis('off')
            if i == 0:
                plt.title('Reconstructed Images')

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

```python
def generate_samples(model, num_samples=20):
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    model.eval()

    with torch.no_grad():
        # Generate random latent vectors
        latent_samples = torch.randn(num_samples, LATENT_DIM).to(device)

        # Generate images using the decoder
        generated = model.decoder(latent_samples)
        generated = generated.cpu().view(-1, 28, 28)

        plt.figure(figsize=(20, 4))
        for i in range(num_samples):
            plt.subplot(2, num_samples // 2, i + 1)
            plt.imshow(generated[i], cmap='gray')
            plt.axis('off')
            if i == 0:
                plt.title('Generated Images')

        plt.tight_layout()
        plt.show()


def main():
    #Data Loading.
    train_loader, test_loader = load_data()

    #Model, criterion and optimizer.
    model = Autoencoder(INPUT_DIM, LATENT_DIM)
    criterion = nn.MSELoss()
    optimizer = optim.Adam(model.parameters(), lr=LEARNING_RATE)

    #Training the model.
    train_losses, test_losses = train_model(
        model, train_loader, test_loader, criterion, optimizer, EPOCHS
    )

    #Plotting training and validation losses
    plt.figure(figsize=(10, 5))
    plt.plot(train_losses, label='Train Loss')
    plt.plot(test_losses, label='Validation Loss')
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.title('Training and Validation Losses')
    plt.legend()
    plt.show()
```

```
    #Visualizing reconstructions
    visualize_reconstruction(model, test_loader)

    #Generating new samples
    generate_samples(model)


if __name__ == "__main__":
    main()
```