GENERATIVE AI
ASSIGNMENT # 1
REPORT

Name: Alaiba Nawaz
Roll Number: 21L-5650
Section: BDS-8A

1.Handmade Architecture of Models (Neat and well explained)

(I don't know what you meant by handmade architecture so just in case i have attached code as well as the drawing of the architecture)

1.1 GAN (CODE)

Generator

```
latent_dim = 100

class Generator(nn.Module):
    def __init__(self):
        super(Generator, self).__init__()
        self.model = nn.Sequential(
            nn.Linear(latent_dim, 256),
            nn.ReLU(),
            nn.Linear(256, 512),
            nn.ReLU(),
            nn.Linear(512, 28 * 28),
            nn.Tanh()
        )

    def forward(self, z):
        img = self.model(z)
        return img.view(-1, 1, 28, 28)
```

Explanation of Generator:

This Generator class in PyTorch defines a neural network that takes a 100-dimensional latent vector (z) as input and transforms it through fully connected layers into a 28×28 image, applying ReLU activations and a final Tanh activation for pixel normalization. The output is reshaped to match the grayscale image format used in datasets like MNIST.

Discriminator

```
class Discriminator(nn.Module):
    def __init__(self):
        super(Discriminator, self).__init__()
        self.model = nn.Sequential(
            nn.Linear(28 * 28, 512),
            nn.LeakyReLU(0.2),
            nn.Linear(512, 256),
            nn.LeakyReLU(0.2),
            nn.Linear(256, 1),
            nn.Sigmoid()
        )

    def forward(self, img):
        img_flat = img.view(img.size(0), -1)
        return self.model(img_flat)
```

Explanation of Discriminator: The Discriminator class in PyTorch defines a neural network that takes a flattened 28×28 image as input, passes it through fully connected layers with LeakyReLU activations, and outputs a single probability (0 to 1) using a Sigmoid activation to classify the image as real or fake.Leaky ReLU is used in the Discriminator to prevent the dying neuron problem, which can occur with standard ReLU when gradients become zero, stopping learning. The small negative slope (0.2 in this case) allows a small gradient for negative inputs, improving gradient flow and training stability, especially in GANs.

Adversarial Training

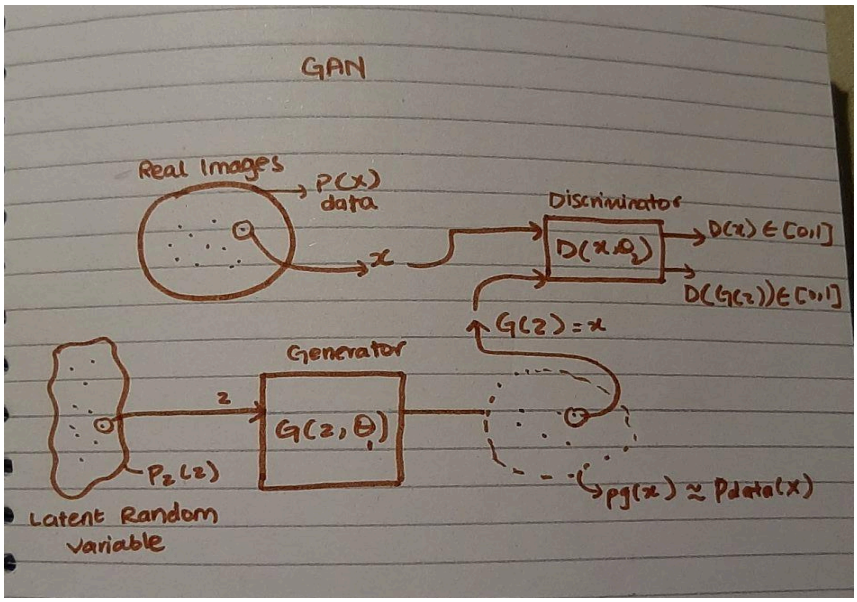
```
def discriminator_loss(real_output, fake_output):
    return -torch.mean(torch.log(real_output) + torch.log(1 - fake_output))

def generator_loss(fake_output):
    return -torch.mean(torch.log(fake_output))
```

Explanation of Adversarial:

discriminator_loss(real_output, fake_output): Encourages the discriminator to maximize the probability of classifying real images correctly (log(real_output)) while minimizing the probability of classifying fake images as real (log(1 - fake_output)).

generator_loss(fake_output): Encourages the generator to maximize the probability that the discriminator classifies fake images as real (log(fake_output)).

1.2 Drawing



A GAN consists of two neural networks:

- **Generator G(z,θ):** Takes a latent random variable z (sampled from a prior distribution P(z)) as input and generates synthetic data samples that resemble real data.
- **Discriminator D(x,ϕ):** Takes an input sample and outputs a probability indicating whether the sample is real (from dataset) or fake (from generator).

Workflow of GAN

1. **Real Data Distribution P(x):** The model starts with a dataset containing real images (or any data distribution).
2. **Latent Space Sampling P(z):** A random variable z is sampled from a known probability distribution (e.g., Gaussian or uniform).
3. **Generator G(z):** The generator network takes z and produces a fake data sample G(z).
4. **Discriminator D(x):**Receives both real samples x from P(x) and fake samples)G(z) and outputs a probability score
5. **Adversarial Training:**The Discriminator tries to maximize its accuracy by distinguishing real from fake.The Generator tries to fool the discriminator by producing more realistic samples.

Objective Function (Loss Function)

- The discriminator is trained to maximize
- The generator is trained to minimize

1.2 VAE(CODE)

Encoder

```
class Encoder(nn.Module):
    def __init__(self, latent_dim=20):
        super(Encoder, self).__init__()
        self.conv = nn.Sequential(
            nn.Conv2d(1, 32, kernel_size=4, stride=2, padding=1),
            nn.ReLU(),
            nn.Conv2d(32, 64, kernel_size=4, stride=2, padding=1),
            nn.ReLU()
        )
        self.flatten = nn.Flatten()
        self.fc_mu = nn.Linear(64 * 7 * 7, latent_dim)
        self.fc_logvar = nn.Linear(64 * 7 * 7, latent_dim)

    def forward(self, x):
        h = self.conv(x)
        h = self.flatten(h)
        mu = self.fc_mu(h)
        logvar = self.fc_logvar(h)
        return mu, logvar
```

Explanation of Encoder:

This Encoder class is part of a Variational Autoencoder (VAE) and maps an input image to a latent space distribution:

- Uses convolutional layers to extract hierarchical features from the input image.
- Flattens the feature maps and passes them through two separate fully connected layers to generate the mean (mu) and log-variance (logvar) of a latent space distribution.
- These outputs define a Gaussian distribution, from which latent vectors are sampled for reconstruction.

Decoder

```
class Decoder(nn.Module):
    def __init__(self, latent_dim=20):
        super(Decoder, self).__init__()
        self.fc = nn.Linear(latent_dim, 64 * 7 * 7)
        self.deconv = nn.Sequential(
            nn.ConvTranspose2d(64, 32, kernel_size=4, stride=2, padding=1),
            nn.ReLU(),
            nn.ConvTranspose2d(32, 1, kernel_size=4, stride=2, padding=1),
            nn.Sigmoid()
        )

    def forward(self, z):
        h = self.fc(z)
        h = h.view(-1, 64, 7, 7)
        x_recon = self.deconv(h)
        return x_recon
```

Explanation of Decoder: This Decoder class reconstructs an image from a latent vector in a Variational Autoencoder (VAE):

- A fully connected layer expands the latent vector into a 7×7×64 feature map.
- Transpose convolution layers progressively upsample the feature map to a 28×28 image.
- A Sigmoid activation ensures pixel values are between 0 and 1, suitable for grayscale images like MNIST.

VAE Model ¶

```
class VAE(nn.Module):
    def __init__(self, latent_dim=20):
        super(VAE, self).__init__()
        self.encoder = Encoder(latent_dim)
        self.decoder = Decoder(latent_dim)

    def reparameterize(self, mu, logvar):
        std = torch.exp(0.5 * logvar)
        eps = torch.randn_like(std)
        return mu + eps * std

    def forward(self, x):
        mu, logvar = self.encoder(x)
        z = self.reparameterize(mu, logvar)
        x_recon = self.decoder(z)
        return x_recon, mu, logvar
```

Explanation of VAE Model: This VAE class implements a Variational Autoencoder (VAE) by combining an encoder, a decoder, and a reparameterization trick:

- The encoder maps input images to a latent space distribution (mu, logvar).
- The reparameterization trick samples z from this distribution to enable backpropagation.
- The decoder reconstructs the image from z, learning a meaningful latent representation.

Loss Function

```
def loss_function(x_recon, x, mu, logvar):
    BCE = F.binary_cross_entropy(x_recon, x, reduction='sum')
    KL = -0.5 * torch.sum(1 + logvar - mu.pow(2) - logvar.exp())
    return BCE + KL
```

Explanation of Loss Function:

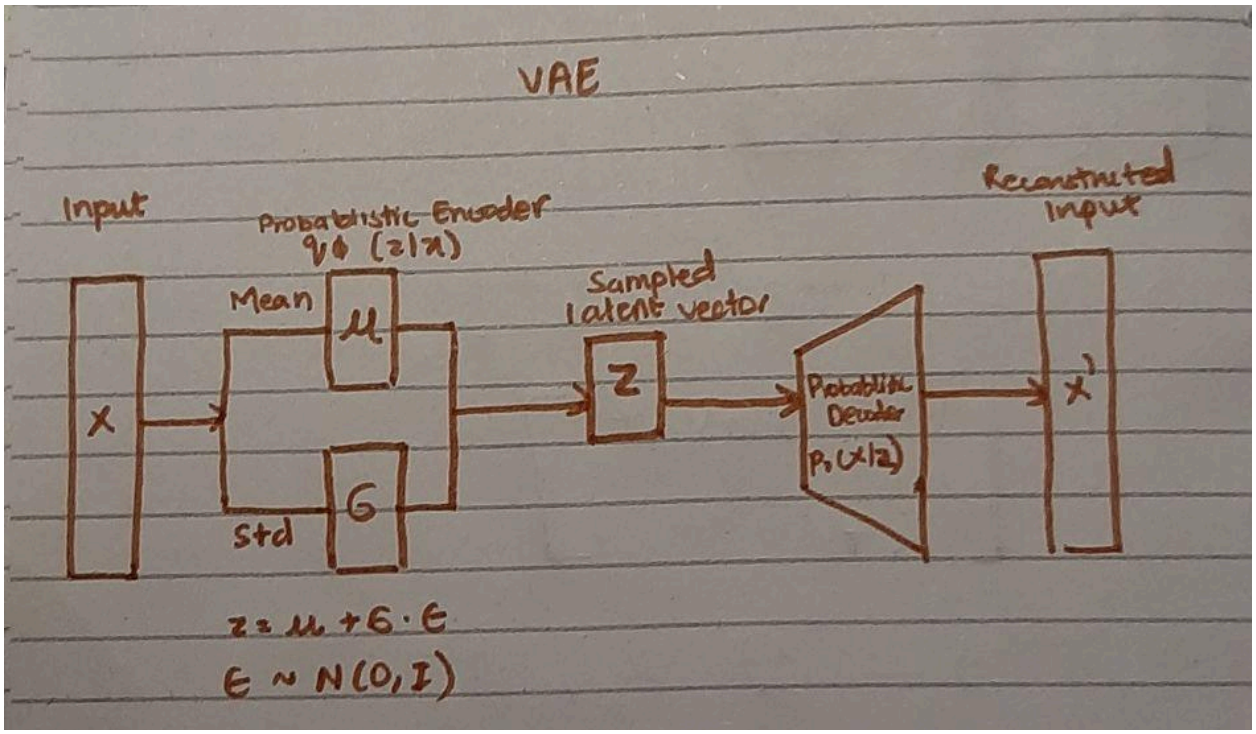
This VAE loss function combines two key terms:

Reconstruction Loss (BCE): Measures how well the reconstructed image (x_recon) matches the original image (x) using binary cross-entropy, encouraging accurate reconstruction.

Kullback-Leibler (KL) Divergence: Regularizes the latent space by forcing the learned distribution (mu, logvar) to be close to a standard normal distribution (N(0,1)), ensuring smoothness and meaningful latent representations.

The total loss (BCE + KL) balances reconstruction quality and latent space regularization.

1.3 VAE Drawing



The Variational Autoencoder (VAE) is a generative model that learns efficient latent representations of data by introducing probabilistic encoding and decoding. Its architecture consists of three main components:

1. **Encoder (Probabilistic Inference Network):** The encoder maps an input X to a latent space representation by estimating two parameters: the mean (μ\mu) and standard deviation (σ\sigma) of a Gaussian distribution. Instead of directly encoding X into a single point, VAE encodes it into a distribution, ensuring smooth transitions in the latent space.

2. **Latent Space (Reparameterization Trick):** To enable gradient-based optimization, VAE samples a latent vector Z from the estimated Gaussian distribution using the reparameterization trick:$Z=\mu+\sigma \cdot \epsilon, \epsilon \sim N(0,I)$.This trick allows backpropagation while maintaining stochasticity in sampling.
3. **Decoder (Generative Model):** The decoder reconstructs the input X' from the sampled latent vector Z by learning the conditional probability distribution $P(X|Z)$. This ensures that the reconstructed output resembles the original input while maintaining variability in generated outputs.

The VAE is trained using a loss function that combines reconstruction loss (to ensure accurate output reconstruction) and KL-divergence loss (to enforce the latent space distribution to be close to a standard normal distribution). This architecture makes VAEs powerful for applications like image generation, anomaly detection, and data compression.

2. Generated images

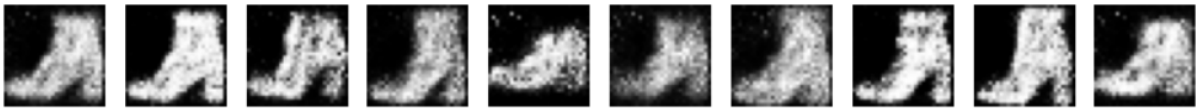
2.1 Generated images by GANS

2.1.1 MNIST DIGITS

```
Generating 10 random digit images
MNIST Models loaded.
```

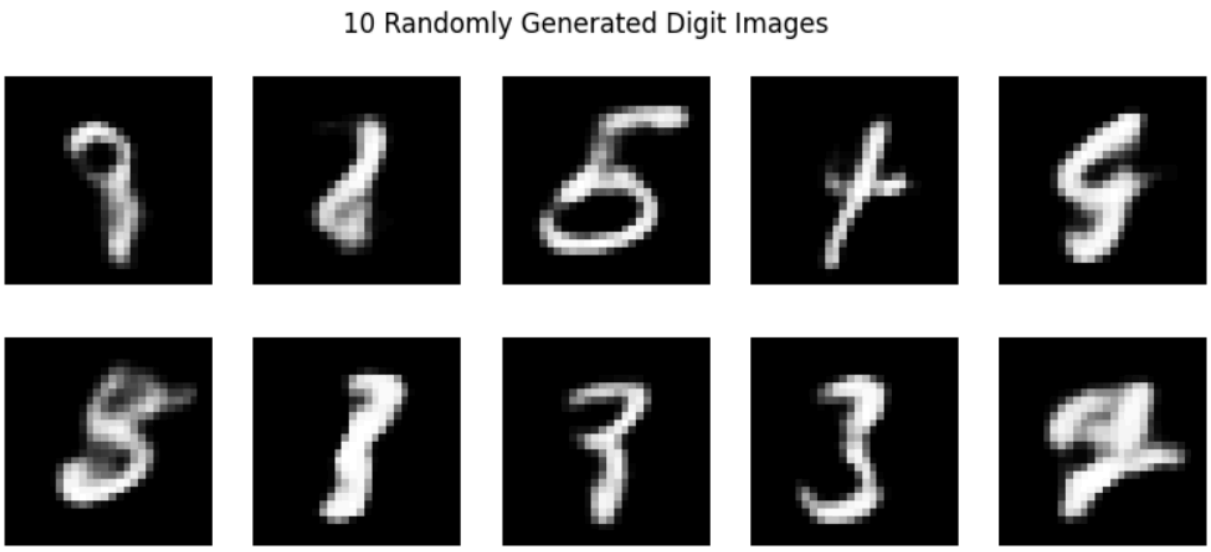
2.1.1 MNIST FASHION

```
Generating 10 images of shoes
FASHION Models loaded.
```

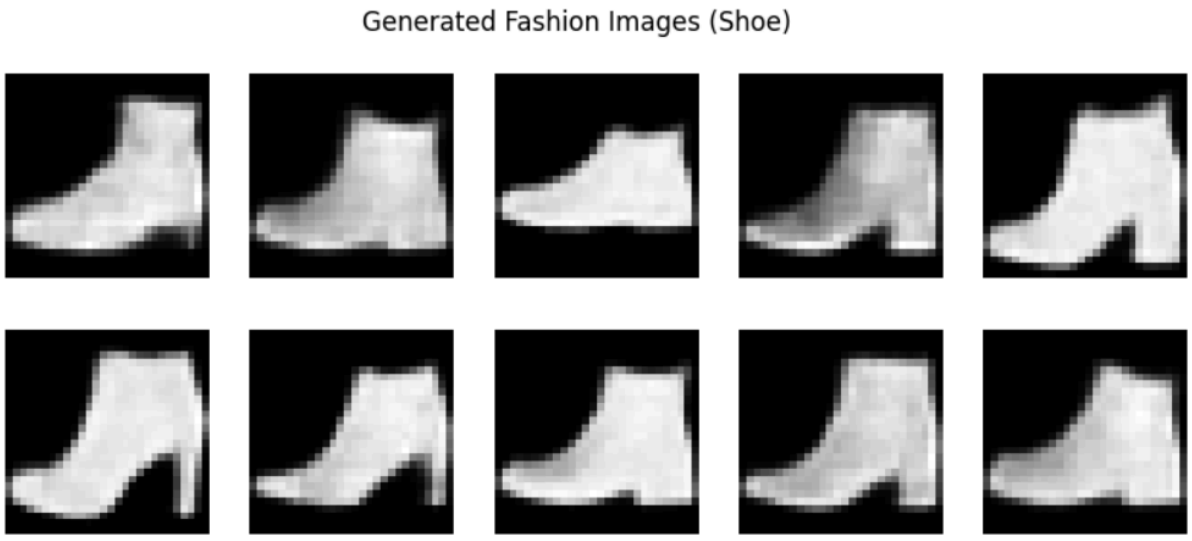


2.1 Generated images by VAES

2.2.1 MNIST DIGITS

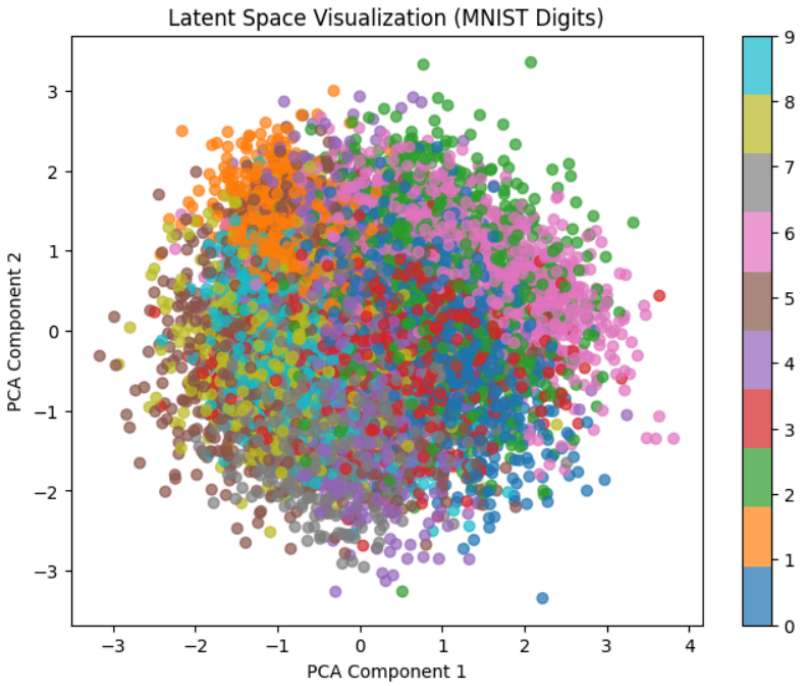


2.2.1 MNIST FASHION

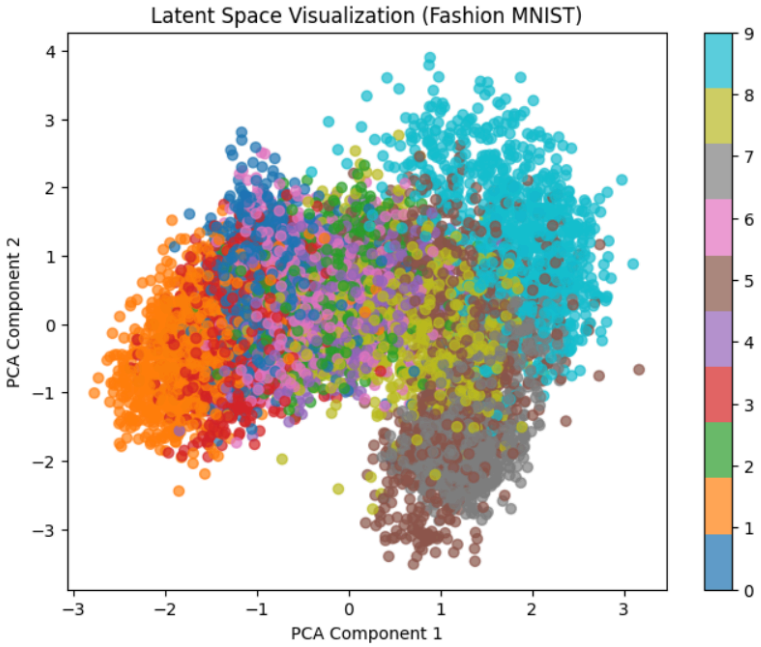


3. Plots for latent space representation

3.1 Latent Space Visualization MNIST DIGITS



3.1 Latent Space Visualization MNIST FASHION



4. Discussions

4.1 Latent Space Discussion

4.1.1 Latent Space MNIST Digits

- The points are spread out in a dense, overlapping manner, indicating significant similarity between digit classes.
- Since handwritten digits often have structural similarities (e.g., 3 and 8, 1 and 7), the lack of clear clustering suggests that distinguishing certain digits may be challenging in a lower-dimensional space.
- The distribution appears more circular and uniform, meaning that feature extraction might not lead to well-separated clusters

4.1.2 Latent Space Fashion MNIST

- Unlike digits, Fashion MNIST shows distinct clusters, meaning different clothing items (shoes, shirts, bags) have more unique feature representations.
- The spread is more structured, with regions that seem to be dominated by particular classes, suggesting that fashion items have more identifiable patterns.
- This clear separation indicates that even simple models can classify Fashion MNIST better than MNIST Digits, as clothing categories have more distinguishable characteristics than handwritten digits.

4.2 Image Quality Discussion

The images produced by gans are sharper than the images produced by vaes.

1. Why are GANs sharper?

- GANs use a generator and a discriminator in a competitive game.
- The discriminator pushes the generator to create images that look as realistic as possible.
- This adversarial loss function encourages high-frequency details, leading to sharper images.
- However, GANs can suffer from mode collapse, where they generate only a limited variety of images.

2. Why are VAEs blurry?

- VAEs learn a probabilistic latent space where images are encoded and then reconstructed.
- The reconstruction involves sampling from a learned distribution, which introduces noise.
- The loss function includes a KL divergence term, forcing the latent space to be smooth and continuous.

- This smoothness leads to blurry outputs, as the model learns to average multiple plausible reconstructions instead of fine-tuning sharp details.