Experiment #	<TO BE FILLED BY STUDENT>	Student ID	<TO BE FILLED BY STUDENT>
Date	<TO BE FILLED BY STUDENT>	Student Name	[@KLWKS_bot] THANOS

Experiment 16 : Generative Adversarial Networks (GANs) for Image Generation on MNIST Data

Aim/Objective:

To implement a Generative Adversarial Network (GAN) on the MNIST dataset for image generation, focusing on understanding the adversarial training process and evaluating the quality of generated images.

Description:

GANs consist of two neural networks: a generator and a discriminator. The generator learns to create realistic data samples, while the discriminator distinguishes real data from generated data. This experiment explores the GAN architecture and its application to generating handwritten digit images.

Pre-Requisites:

Understanding of GAN architecture and adversarial training.
Basics of PyTorch and neural network design.
Familiarity with the MNIST dataset.
Basic knowledge of loss functions (e.g., binary cross-entropy).

Pre-Lab:

1. What is the primary purpose of using GANs for image generation?

GANs generate realistic images by learning patterns from real data and synthesizing new samples that resemble them.

2. Explain the roles of the generator and discriminator in a GAN. How do they interact during training?

The generator creates fake images, while the discriminator distinguishes real from fake. They compete, improving each other over time—the generator tries to fool the discriminator, and the discriminator learns to detect fakes.

3. What challenges might arise during GAN training, such as mode collapse or vanishing gradients?

Challenges include mode collapse (generator produces limited variations), vanishing gradients (weak updates slow learning), and instability (difficulty in balancing both networks).

4. How does the choice of loss function affect the training of GANs?

The loss function influences training stability. Common choices include Binary Cross-Entropy and Wasserstein loss, which impact convergence and quality of generated images.

In-Lab:
Program 1: Build a GAN for Image Generation on MNIST Dataset.
Procedure/Program:

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

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

latent_dim = 100
image_size = 28
batch_size = 128
num_epochs = 50
learning_rate = 0.0002

dataloader = DataLoader(
   torchvision.datasets.MNIST(
      root="./data", train=True, download=True,
      transform=transforms.Compose([
         transforms.ToTensor(),
         transforms.Normalize((0.5,), (0.5,))
      ])
   ),
   batch_size=batch_size, shuffle=True
)

class Generator(nn.Module):
   def __init__(self):
      super(Generator, self).__init__()
      self.model = nn.Sequential(
         nn.Linear(latent_dim, 128), nn.ReLU(),
         nn.Linear(128, 256), nn.ReLU(),
         nn.Linear(256, 512), nn.ReLU(),
```

```
        nn.Linear(512, image_size * image_size), nn.Tanh()
      )

    def forward(self, z):
      return self.model(z).view(-1, 1, image_size, image_size)

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

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

G = Generator().to(device)
D = Discriminator().to(device)

criterion = nn.BCELoss()
optimizer_G = optim.Adam(G.parameters(), lr=learning_rate, betas=(0.5, 0.999))
optimizer_D = optim.Adam(D.parameters(), lr=learning_rate, betas=(0.5, 0.999))

fixed_noise = torch.randn(16, latent_dim).to(device)

for epoch in range(num_epochs):
    for real_imgs, _ in dataloader:
      real_imgs = real_imgs.to(device)
      batch_size = real_imgs.size(0)

      real_labels = torch.ones(batch_size, 1).to(device)
      fake_labels = torch.zeros(batch_size, 1).to(device)

      optimizer_D.zero_grad()
      real_loss = criterion(D(real_imgs), real_labels)
```

```
    z = torch.randn(batch_size, latent_dim).to(device)
        fake_imgs = G(z)
        fake_loss = criterion(D(fake_imgs.detach()), fake_labels)

        loss_D = real_loss + fake_loss
        loss_D.backward()
        optimizer_D.step()

        optimizer_G.zero_grad()
        loss_G = criterion(D(fake_imgs), real_labels)
        loss_G.backward()
        optimizer_G.step()

    print(f"Epoch [{epoch+1}/{num_epochs}] | D Loss: {loss_D.item():.4f} | G Loss:
{loss_G.item():.4f}")

    with torch.no_grad():
        fake_images = (G(fixed_noise).cpu() + 1) / 2

        fig, axs = plt.subplots(2, 8, figsize=(10, 2))
        for j, img in enumerate(fake_images):
            axs[j // 8, j % 8].imshow(img.squeeze(), cmap="gray")
            axs[j // 8, j % 8].axis("off")
        plt.show()
```

- **Data and Results:**

Data

The MNIST dataset consists of 28x28 grayscale handwritten digit images.

Result

The GAN successfully generates realistic handwritten digits after training iterations.

- **Analysis and Inferences:**

Analysis

The generator improves over epochs, producing clearer and sharper images.

Inferences

GANs can effectively learn digit patterns and generate synthetic images.

Sample VIVA-VOCE Questions (In-Lab):

1. What is the role of random noise in the GAN's generator?

It introduces diversity, preventing the generator from memorizing data and enabling varied outputs.

2. Why is it important to alternate training between the generator and discriminator?

Keeps generator and discriminator balanced; prevents one from overpowering the other, ensuring stable learning.

3. What are some evaluation metrics for the quality of GAN-generated images?

Inception Score (IS), Fréchet Inception Distance (FID), Precision & Recall, and Human Evaluation.

4. How can GANs be modified for conditional image generation?

Use Conditional GANs (cGANs) by adding extra inputs (e.g., labels) to both generator and discriminator for controlled output.

5. What are some real-world applications of GANs beyond image generation?

Data augmentation, style transfer, super-resolution, medical imaging, video synthesis, and drug discovery.

Post-Lab:

Program 2: Explore the impact of changing hyperparameters, such as the learning rate, batch size, generator architecture, and discriminator architecture, on the quality of generated images and the stability of training.

Procedure/Program:

- **Learning Rate**: Too high causes instability; too low slows learning. Optimal: **0.0001–0.0002**.
- **Batch Size**: Small (16–32) adds variance; large (128–256) smooths training but risks mode collapse. Balanced around **64–128**.
- **Generator**: Deeper networks capture details but may overfit; use **batch norm, upsampling, and skip connections**.
- **Discriminator**: Strong ones improve quality but can overpower the generator; use **spectral norm, dropout** for stability.
- **Other Factors**: **Wasserstein loss, gradient penalty, and feature matching** enhance training robustness.

- **Data and Results:**

Data

GAN trained with varying learning rates, batch sizes, and architectures.

Result

Image quality and training stability depend on optimal hyperparameter tuning.

- **Analysis and Inferences:**

Analysis

Higher learning rates destabilize training; deeper networks improve image details.

Inferences

Balanced hyperparameters enhance GAN performance and prevent mode collapse.

Evaluator Remark (if Any):	**Marks Secured _____ out of 50**
	Signature of the Evaluator with Date

Evaluator MUST ask Viva-voce prior to signing and posting marks for each experiment.