Experiment 10

Aim Design and implement a Generative Adversarial Network (GAN) to generate handwritten digits using the MNIST dataset.

Apparatus

- **Hardware:**
  - CPU or GPU-enabled system (Google Colab with GPU recommended)
- **Software and Libraries:**
  - Google Colab or Jupyter Notebook
  - PyTorch o Torchvision o Matplotlib o NumPy
  - IPython.display

Theory

A Generative Adversarial Network (GAN) is a deep learning model introduced by *Ian Goodfellow et al. (2014)* that consists of two neural networks — a Generator (G) and a Discriminator (D) — trained in opposition to one another.

1. **Generator (G):**
   The Generator takes random noise as input and produces fake data resembling real data (in this case, handwritten digits).
2. **Discriminator (D):**
   The Discriminator receives both real and fake data and tries to distinguish between them by outputting a probability value (real or fake).
3. **Adversarial Training:**
   The two networks compete in a *minimax game*:
   - The Generator tries to produce realistic data to fool the Discriminator. o The Discriminator tries to correctly classify real vs. fake data.

   Over time, both networks improve — the Generator learns to produce more realistic images, and the Discriminator becomes a better classifier.

4. **Loss Function:**
   The Binary Cross-Entropy (BCE) loss is used for both G and D to measure how well they perform in distinguishing/generating images.
5. **Mathematical Representation:** $\min_{G} \max_{D} V(D,G) = E_{x\sim p_{\text{dt}}(x)}[\log D(x)] + E_{z\sim p_z(z)}[\log(1 - D(G(z)))]$
6. **Applications of GANs:**
   - Image generation o Super-resolution o Data augmentation o Style transfer

Procedure

1. **Import Required Libraries:**

Import `torch`, `torchvision`, `matplotlib`, and other supporting modules.

2. **Set Hyperparameters:**
   Define latent dimension (100), batch size (128), learning rate (0.0002), and number of epochs (50).
3. **Build the Generator:**
   - Input: Random noise vector (100 dimensions). o Several linear layers with LeakyReLU activation and Batch Normalization.
   - Output: Flattened 28×28 image with Tanh activation.
4. **Build the Discriminator:**
   - Input: Flattened image (28×28 = 784 values). o Several linear layers with LeakyReLU activation and Dropout.
   - Output: Single neuron with Sigmoid activation (real/fake probability).
5. **Load the MNIST Dataset:**
   - Normalize pixel values to range ([-1, 1]).
   - Use DataLoader for batch processing.
6. **Define Optimizers and Loss Function:**
   - Use Binary Cross-Entropy (BCE) loss.
   - Optimizer: Adam (learning rate 0.0002, betas=(0.5, 0.999)).
7. **Training Loop:**
   - For each epoch:
     - Train Discriminator using real and generated fake images.
     - Train Generator to fool the Discriminator.
     - Compute and display losses for both networks.
     - Generate and display sample images periodically.
8. **Plot Loss Curves:**
   Visualize Generator and Discriminator loss vs. epoch.
9. **Generate Final Images:**
   After training, generate a grid of fake handwritten digits to evaluate performance.

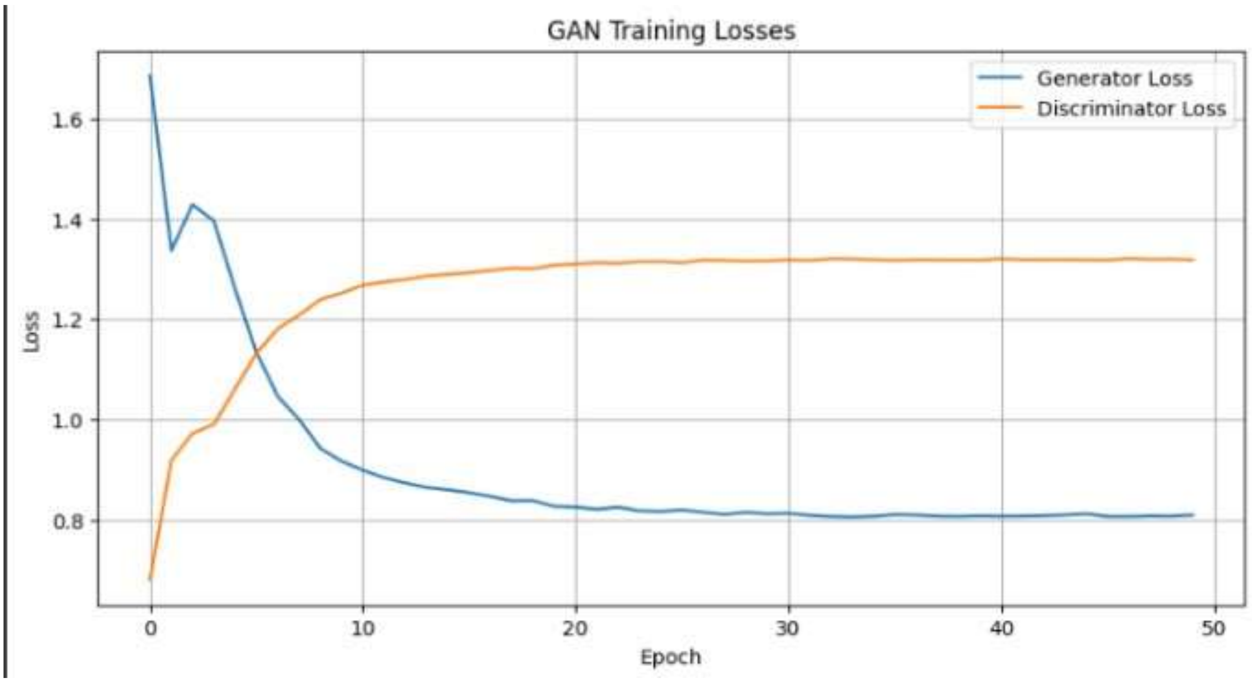
CODE

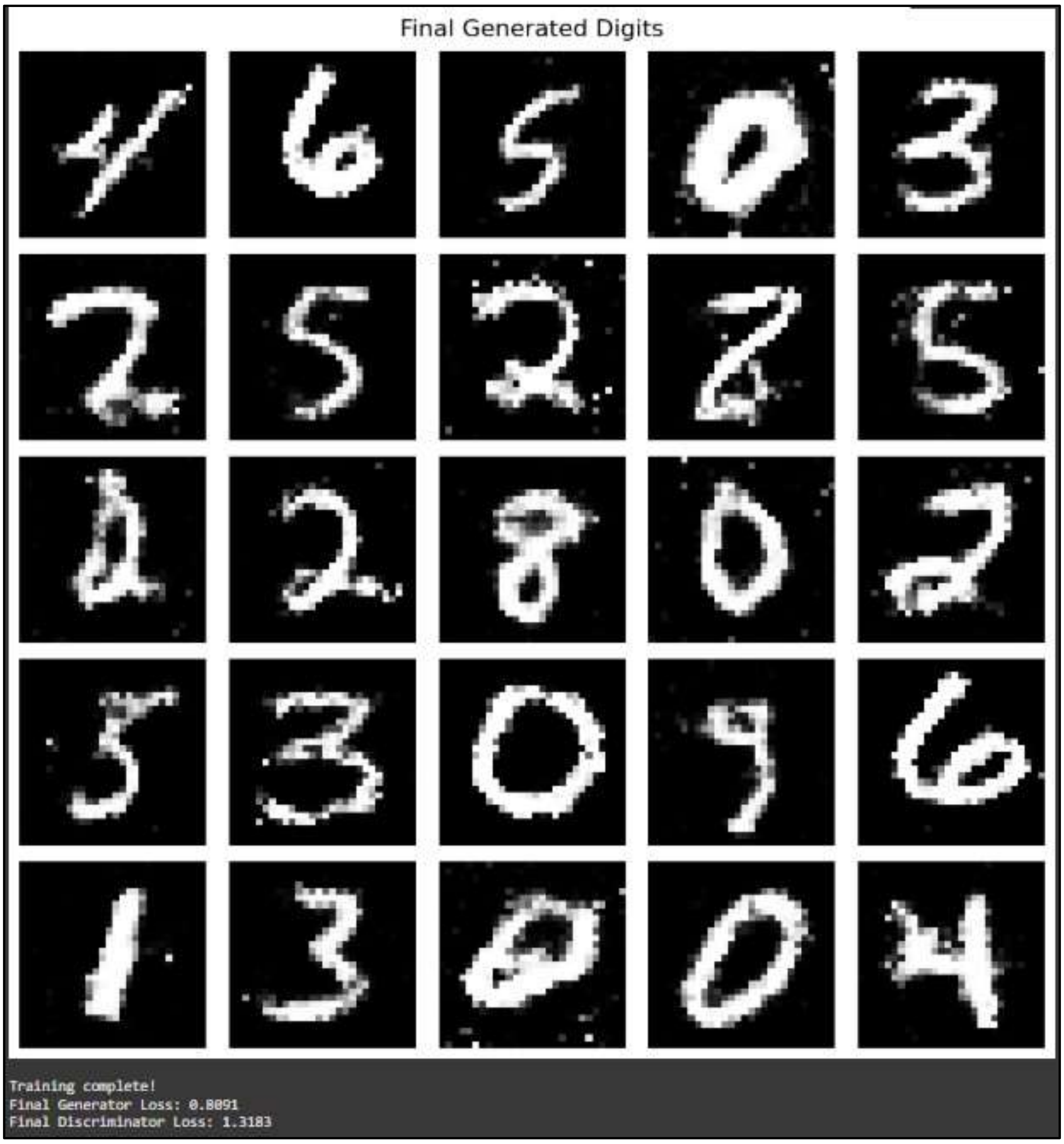
```
# Generative Adversarial Network (GAN) Implementation
# Training on MNIST dataset to generate handwritten digits
# Ready for Google Colab
 import torch import torch.nn as nn import
torch.optim as optim from torch.utils.data
import DataLoader from torchvision import
datasets, transforms import
matplotlib.pyplot as plt import numpy as np
from IPython.display import clear_output

torch.manual_seed(42) np.random.seed(42)

LATENT_DIM = 100
IMG_SIZE = 28 * 28
BATCH_SIZE = 128
LEARNING_RATE = 0.0002
EPOCHS = 50
DEVICE = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

print(f"Using device: {DEVICE}")
```

```
# Generator Network class
Generator(nn.Module):
def __init__(self):
        super(Generator, self).__init__()
self.model = nn.Sequential(
nn.Linear(LATENT_DIM, 256),
nn.LeakyReLU(0.2),
nn.BatchNorm1d(256),
nn.Linear(256, 512),
nn.LeakyReLU(0.2),
nn.BatchNorm1d(512),
nn.Linear(512, 1024),
nn.LeakyReLU(0.2),
nn.BatchNorm1d(1024),
nn.Linear(1024, IMG_SIZE),
nn.Tanh()         )     def forward(self,
z):         img = self.model(z)
return img

# Discriminator Network class
Discriminator(nn.Module):
def __init__(self):
        super(Discriminator, self).__init__()
self.model = nn.Sequential(
nn.Linear(IMG_SIZE, 1024),
nn.LeakyReLU(0.2),
nn.Dropout(0.3),             nn.Linear(1024,
512),             nn.LeakyReLU(0.2),
nn.Dropout(0.3),             nn.Linear(512,
256),             nn.LeakyReLU(0.2),
nn.Dropout(0.3),             nn.Linear(256,
1),             nn.Sigmoid()
        )     def forward(self,
img):         validity =
self.model(img)         return
validity

# Data Loading
transform = transforms.Compose([
transforms.ToTensor(),
    transforms.Normalize([0.5], [0.5])
])
mnist_data = datasets.MNIST(root='./data', train=True, download=True,
transform=transform)
dataloader = DataLoader(mnist_data, batch_size=BATCH_SIZE, shuffle=True,
drop_last=True)

# Model initialization generator =
Generator().to(DEVICE) discriminator =
Discriminator().to(DEVICE) criterion =
nn.BCELoss()
optimizer_G = optim.Adam(generator.parameters(), lr=LEARNING_RATE,
betas=(0.5, 0.999))
optimizer_D = optim.Adam(discriminator.parameters(), lr=LEARNING_RATE,
betas=(0.5, 0.999))
 def
train_gan():
    generator.train()
discriminator.train()     g_losses,
d_losses = [], []
         for epoch in
range(EPOCHS):
        epoch_g_loss, epoch_d_loss = 0, 0         for
i, (real_imgs, _) in enumerate(dataloader):
            batch_size = real_imgs.size(0)
            real_imgs = real_imgs.view(batch_size, -1).to(DEVICE)
real_labels = torch.ones(batch_size, 1).to(DEVICE)
fake_labels = torch.zeros(batch_size, 1).to(DEVICE)

            optimizer_D.zero_grad()
            real_output = discriminator(real_imgs)
            d_loss_real = criterion(real_output, real_labels)

            z = torch.randn(batch_size, LATENT_DIM).to(DEVICE)
fake_imgs = generator(z)
            fake_output = discriminator(fake_imgs.detach())
d_loss_fake = criterion(fake_output, fake_labels)

            d_loss = d_loss_real + d_loss_fake
d_loss.backward()             optimizer_D.step()

            optimizer_G.zero_grad()
            z = torch.randn(batch_size, LATENT_DIM).to(DEVICE)
gen_imgs = generator(z)             g_output =
discriminator(gen_imgs)             g_loss =
criterion(g_output, real_labels)             g_loss.backward()
optimizer_G.step()

            epoch_g_loss += g_loss.item()
epoch_d_loss += d_loss.item()

        avg_g_loss = epoch_g_loss / len(dataloader)
avg_d_loss = epoch_d_loss / len(dataloader)
g_losses.append(avg_g_loss)
d_losses.append(avg_d_loss)
         if (epoch + 1) % 5 ==
0:
            print(f"Epoch [{epoch+1}/{EPOCHS}] | D Loss: {avg_d_loss:.4f} |
G Loss: {avg_g_loss:.4f}")             with torch.no_grad():
                z = torch.randn(16, LATENT_DIM).to(DEVICE)
sample_imgs = generator(z).cpu().view(-1, 28, 28)
clear_output(wait=True)
                fig, axes = plt.subplots(4, 4, figsize=(8, 8))
for idx, ax in enumerate(axes.flat):
                    ax.imshow(sample_imgs[idx], cmap='gray')
ax.axis('off')
                plt.suptitle(f'Generated Images - Epoch {epoch+1}')
plt.tight_layout()
                plt.show()
return g_losses, d_losses

print("Starting GAN training...") g_losses,
d_losses = train_gan()

plt.figure(figsize=(10, 5))
plt.plot(g_losses, label='Generator Loss')
plt.plot(d_losses, label='Discriminator Loss')
plt.xlabel('Epoch') plt.ylabel('Loss')
plt.title('GAN Training Losses') plt.legend()
plt.grid(True) plt.show()
generator.eval() with
torch.no_grad():
    z = torch.randn(25, LATENT_DIM).to(DEVICE)
final_imgs = generator(z).cpu().view(-1, 28, 28)
fig, axes = plt.subplots(5, 5, figsize=(10, 10))     for
idx, ax in enumerate(axes.flat):
        ax.imshow(final_imgs[idx], cmap='gray')
ax.axis('off')
    plt.suptitle('Final Generated Digits', fontsize=16)
plt.tight_layout()     plt.show()

print("\nTraining complete!")
print(f"Final Generator Loss: {g_losses[-1]:.4f}") print(f"Final
Discriminator Loss: {d_losses[-1]:.4f}")
```

Result

- The model successfully generated realistic **handwritten digit images** after training for 50 epochs.
- The **Generator Loss** decreased steadily, showing improved generation quality.
- The **Discriminator Loss** stabilized, indicating balanced adversarial training.E





```
Training complete!
Final Generator Loss: 0.8091
Final Discriminator Loss: 1.3183
```

Conclusion

The **Generative Adversarial Network (GAN)** was successfully implemented and trained on the **MNIST dataset** to synthesize realistic handwritten digits.
The results demonstrate the adversarial learning mechanism where the Generator learns to create convincing fake images, and the Discriminator learns to differentiate between real and fake ones.
This experiment validates the efficiency of GANs in image generation and representation learning tasks.