EX NO: 10	GAN for generating hand-written digits

Aim:

To implement a **Generative Adversarial Network (GAN)** for generating handwritten digits, we can use the **MNIST dataset**.

Algorithm:

1. Import Libraries: We'll use `TensorFlow` and `Keras` for model building.
2. Load MNIST Dataset: The dataset consists of 28x28 grayscale images of handwritten digits (0-9).
3. Define the Generator: The generator takes random noise as input and produces a 28x28 image.
4. Define the Discriminator: The discriminator classifies images as real (from the dataset) or fake (generated by the generator).
5. Define the GAN: The GAN combines the generator and discriminator, and it is trained to generate realistic images.
6. Training Loop: During each iteration, the discriminator and generator are updated alternately.

Code:

```
import torch

import torch.nn as nn

import torch.optim as optim
```

```
import torchvision

import torchvision.transforms as transforms

import matplotlib.pyplot as plt

import numpy as np

from torchvision.utils import make_grid

latent_dim = 100

hidden_dim = 256

image_dim = 784  # 28x28 images

num_epochs = 20

batch_size = 64

lr = 0.0002

beta1 = 0.5

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

transform = transforms.Compose([

    transforms.ToTensor(),

    transforms.Normalize([0.5], [0.5])  # Normalize to [-1, 1]

])

train_dataset = torchvision.datasets.MNIST(root='./data', train=True, transform=transform, download=True)

train_loader = torch.utils.data.DataLoader(dataset=train_dataset, batch_size=batch_size, shuffle=True)

class Generator(nn.Module):

    def __init__(self):

        super(Generator, self).__init__()
```

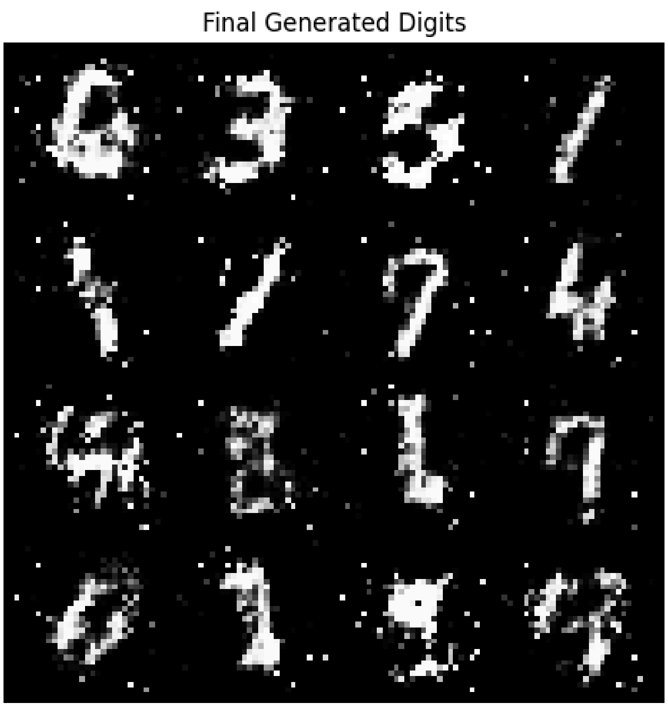
```
        self.model = nn.Sequential(
            nn.Linear(latent_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, image_dim),
            nn.Tanh()  # Output range [-1, 1]
        )

    def forward(self, z):
        img = self.model(z)
        return img.view(img.size(0), 1, 28, 28)
class Discriminator(nn.Module):
    def __init__(self):
        super(Discriminator, self).__init__()
        self.model = nn.Sequential(
            nn.Linear(image_dim, hidden_dim),
            nn.LeakyReLU(0.2),
            nn.Linear(hidden_dim, hidden_dim),
            nn.LeakyReLU(0.2),
            nn.Linear(hidden_dim, 1),
            nn.Sigmoid()  # Output probability
        )
```

```python
    def forward(self, img):
        img_flat = img.view(img.size(0), -1)
        validity = self.model(img_flat)
        return validity
generator = Generator().to(device)
discriminator = Discriminator().to(device)
adversarial_loss = nn.BCELoss()
g_optimizer = optim.Adam(generator.parameters(), lr=lr, betas=(beta1, 0.999))
d_optimizer = optim.Adam(discriminator.parameters(), lr=lr, betas=(beta1, 0.999))

for epoch in range(num_epochs):
    for i, (imgs, _) in enumerate(train_loader):
        batch_size = imgs.size(0)
        real_label = torch.ones(batch_size, 1).to(device)
        fake_label = torch.zeros(batch_size, 1).to(device)
        d_optimizer.zero_grad()

        # Train with real images
        real_imgs = imgs.to(device)
        real_validity = discriminator(real_imgs)
        d_real_loss = adversarial_loss(real_validity, real_label)
        z = torch.randn(batch_size, latent_dim).to(device)
        fake_imgs = generator(z)
```

```
    fake_validity = discriminator(fake_imgs.detach())
    d_fake_loss = adversarial_loss(fake_validity, fake_label)

    # Total discriminator loss
    d_loss = (d_real_loss + d_fake_loss) / 2

    g_optimizer.zero_grad()

    fake_validity = discriminator(fake_imgs)
    g_loss = adversarial_loss(fake_validity, real_label)  # Trick discriminator

    g_loss.backward()
    g_optimizer.step()

    if i % 200 == 0:
        print(f"[Epoch {epoch}/{num_epochs}] [Batch {i}/{len(train_loader)}] "
              f"D_loss: {d_loss.item():.4f}, G_loss: {g_loss.item():.4f}")

if epoch % 10 == 0:
    with torch.no_grad():
        fake_imgs = generator(torch.randn(16, latent_dim).to(device))
        fake_imgs = fake_imgs.cpu()
        grid = make_grid(fake_imgs, nrow=4, normalize=True)
```

```
            plt.figure(figsize=(6, 6))
            plt.imshow(np.transpose(grid, (1, 2, 0)))
            plt.axis('off')
            plt.title(f'Generated Digits at Epoch {epoch}')
            plt.savefig(f'generated_digits_epoch_{epoch}.png')
            plt.close()

torch.save(generator.state_dict(), 'generator.pth')
with torch.no_grad():
    final_samples = generator(torch.randn(16, latent_dim).to(device))
    final_samples = final_samples.cpu()
    grid = make_grid(final_samples, nrow=4, normalize=True)
    plt.figure(figsize=(6, 6))
    plt.imshow(np.transpose(grid, (1, 2, 0)))
    plt.axis('off')
    plt.title('Final Generated Digits')
    plt.savefig('final_generated_digits.png')
    plt.show()
```

Output:



Result:

After training the GAN, the generator starts producing realistic handwritten digits that resemble the MNIST dataset. Initially, the images may appear random, but as training progresses, they become more recognizable as handwritten digits.