Generative Adversarial Network (GAN)

A Generative Adversarial Network (GAN) is a type of machine learning model used for generating new data that resembles a given dataset. It consists of two neural networks that compete against each other in a process called adversarial training.

Components of a GAN

1. Generator (G): *T akes in random noise as input.
    - Tries to generate realistic-looking data (e.g., images).
    - Learns to trick the Discriminator into believing that generated data is real.
2. Discriminator (D):
    - Takes in both real and generated data as input.
    - Learns to distinguish between real and fake data.
    - Provides feedback to the Generator so it can improve.

How GANs Work

1. Random noise is fed into the Generator, which produces fake images.
2. Real images (from the dataset) and fake images (from the Generator) are given to the Discriminator.
3. The Discriminator tries to correctly classify images as real or fake.
4. The Generator learns to produce more realistic images by improving based on the Discriminator’s feedback.
5. Over time, both networks improve, and the Generator starts creating images that look real.

Loss Functions

1. The Discriminator aims to maximize its ability to differentiate real from fake data.
2. The Generator tries to minimize the Discriminator’s ability to detect fake data.
3. This results in a zero-sum game, where one model's improvement means the other must improve as well.

Applications of GANs

1. Image Generation (e.g., DeepFake, AI art)
2. Data Augmentation (creating more training samples)
3. Super-Resolution (enhancing low-quality images)
4. Style Transfer (applying artistic styles to photos)
5. 3D Model Generation (for gaming and simulation)

```
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
import numpy as np
```

```
# Hyperparameters
latent_dim = 100
batch_size = 64
epochs = 50
lr = 0.0002
image_size = 28*28  # Flattened MNIST images
```

```
# Generator
class Generator(nn.Module):
    def __init__(self):
        super(Generator, self).__init__()
        self.model = nn.Sequential(
```

```python
            nn.Linear(latent_dim, 256),
            nn.ReLU(),
            nn.Linear(256, 512),
            nn.ReLU(),
            nn.Linear(512, image_size),
            nn.Tanh()
        )

    def forward(self, z):
        return self.model(z)
```

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

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

```python
# Initialize models
generator = Generator()
discriminator = Discriminator()
```

```python
# Loss and optimizers
criterion = nn.BCELoss()
g_optimizer = optim.Adam(generator.parameters(), lr=lr)
d_optimizer = optim.Adam(discriminator.parameters(), lr=lr)
```

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

```python
# Training loop
for epoch in range(epochs):
    for i, (real_images, _) in enumerate(dataloader):
        batch_size = real_images.shape[0]
        real_images = real_images.view(batch_size, -1)

        # Train Discriminator
        real_labels = torch.ones(batch_size, 1)
        fake_labels = torch.zeros(batch_size, 1)

        real_output = discriminator(real_images)
        real_loss = criterion(real_output, real_labels)

        z = torch.randn(batch_size, latent_dim)
        fake_images = generator(z)
        fake_output = discriminator(fake_images.detach())
        fake_loss = criterion(fake_output, fake_labels)

        d_loss = real_loss + fake_loss
        d_optimizer.zero_grad()
        d_loss.backward()
        d_optimizer.step()

        # Train Generator
        fake_output = discriminator(fake_images)
        g_loss = criterion(fake_output, real_labels)

        g_optimizer.zero_grad()
        g_loss.backward()
```

```
        g_optimizer.step()

    print(f"Epoch [{epoch+1}/{epochs}]  D Loss: {d_loss.item():.4f}  G Loss: {g_loss.item():.4f}")
```

```
# Generate and display some fake images
z = torch.randn(16, latent_dim)
fake_images = generator(z).detach().numpy()
fake_images = fake_images.reshape(-1, 28, 28)

fig, axes = plt.subplots(4, 4, figsize=(6,6))
for i, ax in enumerate(axes.flat):
    ax.imshow(fake_images[i], cmap='gray')
    ax.axis('off')
plt.show()
```