Practical no: 9

Aim: Set up a generator network to produce samples and a discriminator network to distinguish between real and generated data. (Use a simple dataset)

Description: This code implements a simple Generative Adversarial Network (GAN) using PyTorch to learn a 1D data distribution. The Generator takes random noise as input and produces fake samples aiming to mimic the real data, which in this case are samples drawn from a standard normal distribution. The Discriminator is trained to distinguish between real samples and generated fake samples, outputting a probability score. Both networks are simple fully connected neural networks.

During training, the Discriminator learns to better identify real vs. fake data, while the Generator improves to fool the Discriminator. Every 500 epochs, the code visualizes the distributions of real and generated samples using histograms, showing how the Generator’s output gradually aligns with the real data distribution. This visual feedback helps to understand the adversarial learning process, demonstrating the Generator’s progress in mimicking the true data distribution over time.

Working:

- Every 500 epochs, the script will display a histogram comparing real samples (from normal distribution) and generated samples.
- You should see the fake data distribution getting closer to the real one over time.
- At the end, it prints 10 generated samples.

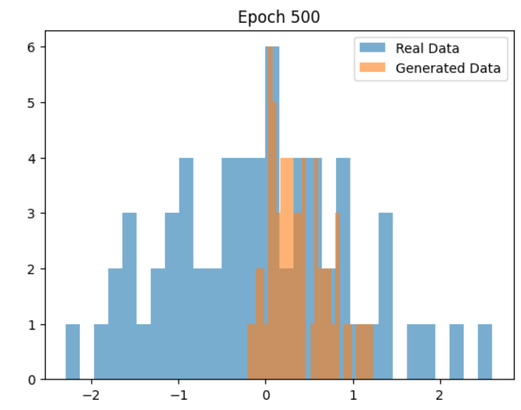
Code:

```
import torch
import torch.nn as nn
import torch.optim as optim
import matplotlib.pyplot as plt
import numpy as np

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

# Generator network
class Generator(nn.Module):
    def __init__(self, noise_dim=5, output_dim=1):
        super(Generator, self).__init__()
        self.net = nn.Sequential(
            nn.Linear(noise_dim, 16),
            nn.ReLU(),
            nn.Linear(16, 32),
            nn.ReLU(),
            nn.Linear(32, output_dim)
        )
    def forward(self, x):
        return self.net(x)

# Discriminator network
class Discriminator(nn.Module):
    def __init__(self, input_dim=1):
        super(Discriminator, self).__init__()
        self.net = nn.Sequential(
            nn.Linear(input_dim, 32),
            nn.LeakyReLU(0.2),
            nn.Linear(32, 16),
            nn.LeakyReLU(0.2),
            nn.Linear(16, 1),
            nn.Sigmoid()
        )

    def forward(self, x):
        return self.net(x)

# Hyperparameters
noise_dim = 5
batch_size = 64
lr = 0.001
num_epochs = 5000
plot_interval = 500

# Instantiate networks
G = Generator(noise_dim).to(device)
D = Discriminator().to(device)

# Optimizers
optimizer_G = optim.Adam(G.parameters(), lr=lr)
optimizer_D = optim.Adam(D.parameters(), lr=lr)

# Loss function
criterion = nn.BCELoss()

# For plotting
plt.ion()
fig, ax = plt.subplots()

for epoch in range(1, num_epochs + 1):
    # 1. Sample real data (from normal distribution)
    real_samples = torch.randn(batch_size, 1).to(device)
    real_labels = torch.ones(batch_size, 1).to(device)

    # 2. Sample noise and generate fake data
    noise = torch.randn(batch_size, noise_dim).to(device)
    fake_samples = G(noise)
    fake_labels = torch.zeros(batch_size, 1).to(device)

    # --- Train Discriminator ---
    optimizer_D.zero_grad()
    outputs_real = D(real_samples)
    loss_real = criterion(outputs_real, real_labels)

    outputs_fake = D(fake_samples.detach())
    loss_fake = criterion(outputs_fake, fake_labels)

    loss_D = loss_real + loss_fake
    loss_D.backward()
    optimizer_D.step()

    # --- Train Generator ---
    optimizer_G.zero_grad()
    fake_samples = G(noise)
    outputs = D(fake_samples)
    loss_G = criterion(outputs, real_labels)  # want fake samples to be classified as real
    loss_G.backward()
    optimizer_G.step()

    # Print losses
    if epoch % plot_interval == 0:
        print(f"Epoch [{epoch}/{num_epochs}] Loss D: {loss_D.item():.4f}, Loss G:
{loss_G.item():.4f}")

        # Plot real vs generated distributions
        ax.clear()
        real_np = real_samples.cpu().detach().numpy()
        gen_np = fake_samples.cpu().detach().numpy()

        ax.hist(real_np, bins=30, alpha=0.6, label='Real Data')
        ax.hist(gen_np, bins=30, alpha=0.6, label='Generated Data')
        ax.legend()
        ax.set_title(f"Epoch {epoch}")
        plt.pause(0.1)

plt.ioff()
plt.show()

# Final generated samples output
with torch.no_grad():
    test_noise = torch.randn(10, noise_dim).to(device)
    generated_samples = G(test_noise).cpu().numpy()
```

```
print("Generated samples:", generated_samples.flatten())
```

Output:

```
Epoch [500/5000] Loss D: 1.3082, Loss G: 0.5938
```



```
Epoch [1000/5000] Loss D: 1.3955, Loss G: 0.6868
Epoch [1500/5000] Loss D: 1.3857, Loss G: 0.6820
Epoch [2000/5000] Loss D: 1.3880, Loss G: 0.6767
Epoch [2500/5000] Loss D: 1.3851, Loss G: 0.6818
Epoch [3000/5000] Loss D: 1.3552, Loss G: 0.7459
Epoch [3500/5000] Loss D: 1.3707, Loss G: 0.6556
Epoch [4000/5000] Loss D: 1.3754, Loss G: 0.6547
Epoch [4500/5000] Loss D: 1.3771, Loss G: 0.6992
Epoch [5000/5000] Loss D: 1.3818, Loss G: 0.6930
Generated samples: [ 0.8334529  -0.62251115 -0.08735475 -1.0544331  -0.39718705 -0.1324631
  0.8607729  -0.10508311 -0.6055295  -0.8825881 ]
```

Conclusion & Learning:

1. **Basic GAN Architecture Works:** Even a simple Generator and Discriminator with fully connected layers can learn to approximate a simple data distribution like a 1D normal distribution.
2. **Adversarial Training Dynamics:** The Generator and Discriminator improve together — the Discriminator gets better at spotting fakes, while the Generator improves to fool the Discriminator, showing the essence of adversarial learning.
3. **Visualization is Key:** Plotting the real vs. generated data distributions during training provides valuable insight into how well the Generator is learning, making it easier to diagnose training progress or issues like mode collapse.
4. **Training Stability:** GANs can be unstable and require careful tuning of hyperparameters like learning rates and batch sizes. In this simple example, stable training is easier but still benefits from such tuning.
5. **Limitations of Simple Models:** While this toy example works well for 1D data, real-world applications typically require more complex networks (e.g., CNNs for images) and training tricks for good results.
6. **Importance of Noise Input:** The noise vector fed to the Generator is essential for producing diverse outputs and learning the underlying data distribution.
7. **GANs Learn Distributions, Not Just Points:** Over time, the Generator captures the overall shape of the data distribution, not just memorizing specific samples.