# Report on GAN for FashionMNIST Dataset

## 1. Definition of the Problem

The goal of this project is to develop a Generative Adversarial Network (GAN) to generate synthetic images that resemble the images in the FashionMNIST dataset. The FashionMNIST dataset consists of 60,000 training images and 10,000 test images of fashion items, such as clothing and accessories, each represented as a 28x28 grayscale image.

## 2. My Approach

### Data Preparation

The FashionMNIST dataset is loaded and preprocessed using the `torchvision` library. The images are transformed into tensors and normalized to have a mean of 0.5 and a standard deviation of 0.5, which is a common practice for GANs to stabilize training.

In [1]:
```
import torch
import torch.nn as nn
import torch.optim as optim
import matplotlib.pyplot as plt
from torch.utils.data import DataLoader
from torchvision import datasets, transforms
from torchvision.utils import make_grid
from torchvision.datasets import FashionMNIST
```

In [2]:
```
# Load FashionMNIST dataset
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.5], std=[0.5])  # Normalization for GANs
])

train_data = datasets.FashionMNIST(root='./data', train=True, download=True, tra
dataset_loader = DataLoader(train_data, batch_size=64, shuffle=True)
```

In [3]:
```
# Define Generator network
class Generator(nn.Module):
    def __init__(self, noise_dim, output_dim):
        super(Generator, self).__init__()
        self.model = nn.Sequential(
            nn.Linear(noise_dim, 256),
            nn.ReLU(),
            nn.Linear(256, 512),
            nn.ReLU(),
            nn.Linear(512, 1024),
            nn.ReLU(),
            nn.Linear(1024, output_dim),
            nn.Tanh()  # Output activation for image generation
```

```
        )

    def forward(self, z):
        return self.model(z).view(-1, 1, 28, 28)  # Reshape to image dimensions

# Define Discriminator network
class Discriminator(nn.Module):
    def __init__(self, input_dim):
        super(Discriminator, self).__init__()
        self.model = nn.Sequential(
            nn.Flatten(),
            nn.Linear(input_dim, 1024),
            nn.LeakyReLU(0.2),
            nn.Linear(1024, 512),
            nn.LeakyReLU(0.2),
            nn.Linear(512, 256),
            nn.LeakyReLU(0.2),
            nn.Linear(256, 1),
            nn.Sigmoid()  # Output probability of real or fake
        )

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

In [ ]:
```
# Initialize models and optimizers
noise_dim = 100
generator = Generator(noise_dim=noise_dim, output_dim=28*28)
discriminator = Discriminator(input_dim=28*28)
optim_G = optim.Adam(generator.parameters(), lr=0.0002, betas=(0.5, 0.999))
optim_D = optim.Adam(discriminator.parameters(), lr=0.0002, betas=(0.5, 0.999))
loss_fn = nn.BCELoss()
```

In [ ]:
```
# Training loop
num_epochs = 10  # Reduced number of epochs for debugging
losses_D = []
losses_G = []
for epoch in range(num_epochs):
    for i, (real_data, _) in enumerate(dataset_loader):
        batch_size = real_data.size(0)
        real_data = real_data.view(batch_size, -1)  # Flatten the images

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

        # Train Discriminator
        noise = torch.randn(batch_size, noise_dim)
        fake_data = generator(noise).view(batch_size, -1)  # Flatten the fake im

        d_real = discriminator(real_data)
        d_fake = discriminator(fake_data.detach())
        loss_D = loss_fn(d_real, real_labels) + loss_fn(d_fake, fake_labels)

        optim_D.zero_grad()
        loss_D.backward()
        optim_D.step()

        # Train Generator
        d_fake = discriminator(fake_data)
```

```
        loss_G = loss_fn(d_fake, real_labels)

        optim_G.zero_grad()
        loss_G.backward()
        optim_G.step()

        if i % 100 == 0:
            print(f'Epoch [{epoch+1}/{num_epochs}], Step [{i}/{len(dataset_loade

    if (epoch + 1) % 1 == 0:  # Append losses every epoch
        losses_D.append(loss_D.item())
        losses_G.append(loss_G.item())
        print(f'Epoch [{epoch+1}/{num_epochs}] - Loss D: {loss_D:.4f}, Loss G: {
```

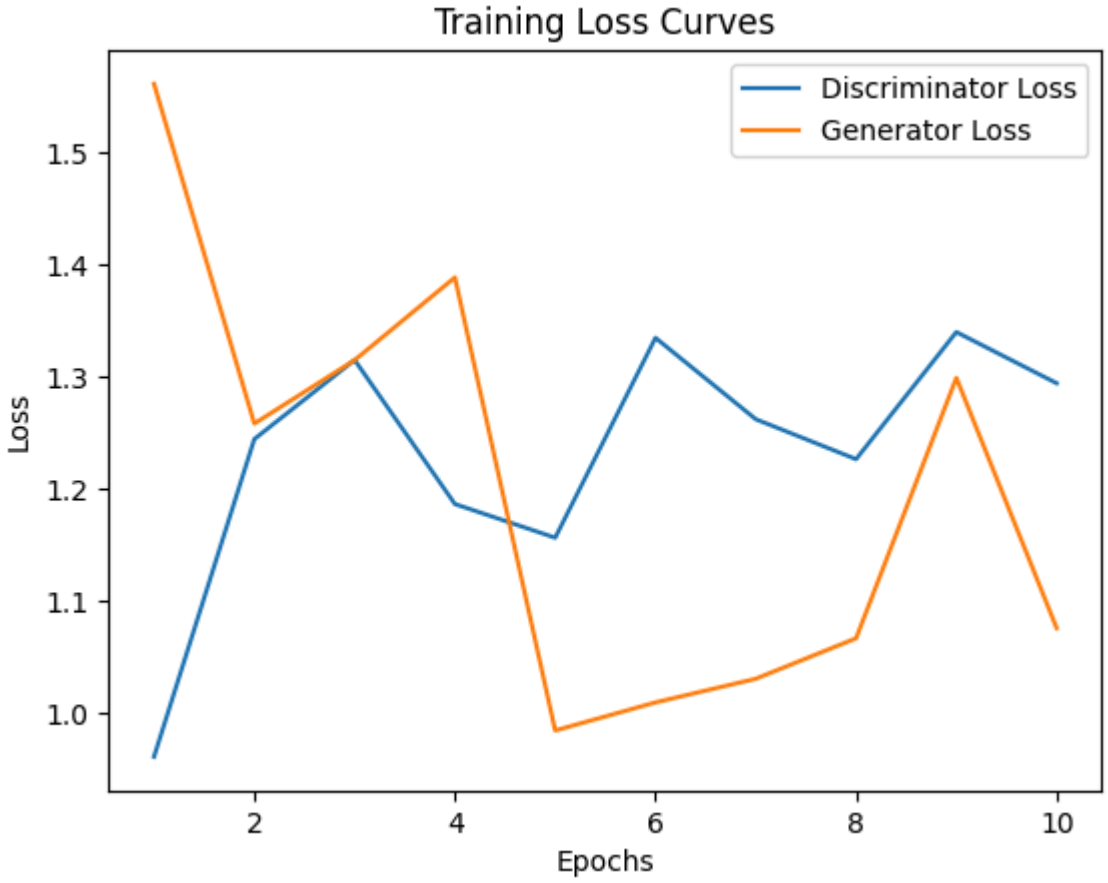
```
Epoch [1/10], Step [0/938] - Loss D: 1.0274, Loss G: 1.8492
Epoch [1/10], Step [100/938] - Loss D: 1.1280, Loss G: 1.0659
Epoch [1/10], Step [200/938] - Loss D: 0.9363, Loss G: 1.8028
Epoch [1/10], Step [300/938] - Loss D: 1.0019, Loss G: 1.1134
Epoch [1/10], Step [400/938] - Loss D: 0.7892, Loss G: 1.7713
Epoch [1/10], Step [500/938] - Loss D: 0.8718, Loss G: 1.2641
Epoch [1/10], Step [600/938] - Loss D: 0.8711, Loss G: 1.4110
Epoch [1/10], Step [700/938] - Loss D: 1.1560, Loss G: 1.9235
Epoch [1/10], Step [800/938] - Loss D: 1.0374, Loss G: 1.3158
Epoch [1/10], Step [900/938] - Loss D: 0.8652, Loss G: 1.2441
Epoch [1/10] - Loss D: 0.9606, Loss G: 1.5611
Epoch [2/10], Step [0/938] - Loss D: 1.0638, Loss G: 1.2338
Epoch [2/10], Step [100/938] - Loss D: 0.8777, Loss G: 1.6571
Epoch [2/10], Step [200/938] - Loss D: 1.1696, Loss G: 1.6551
Epoch [2/10], Step [300/938] - Loss D: 1.2589, Loss G: 1.1812
Epoch [2/10], Step [400/938] - Loss D: 1.2168, Loss G: 1.1594
Epoch [2/10], Step [500/938] - Loss D: 1.1878, Loss G: 1.9036
Epoch [2/10], Step [600/938] - Loss D: 1.1108, Loss G: 1.3561
Epoch [2/10], Step [700/938] - Loss D: 1.1295, Loss G: 1.4372
Epoch [2/10], Step [800/938] - Loss D: 1.2276, Loss G: 1.2713
Epoch [2/10], Step [900/938] - Loss D: 1.1112, Loss G: 1.0304
Epoch [2/10] - Loss D: 1.2440, Loss G: 1.2579
Epoch [3/10], Step [0/938] - Loss D: 1.1999, Loss G: 1.2205
Epoch [3/10], Step [100/938] - Loss D: 1.2077, Loss G: 1.5232
Epoch [3/10], Step [200/938] - Loss D: 1.2578, Loss G: 1.2473
Epoch [3/10], Step [300/938] - Loss D: 1.1592, Loss G: 1.1616
Epoch [3/10], Step [400/938] - Loss D: 1.1693, Loss G: 1.4918
Epoch [3/10], Step [500/938] - Loss D: 1.1347, Loss G: 1.0624
Epoch [3/10], Step [600/938] - Loss D: 1.1438, Loss G: 1.1968
Epoch [3/10], Step [700/938] - Loss D: 0.9556, Loss G: 1.1356
Epoch [3/10], Step [800/938] - Loss D: 1.1126, Loss G: 0.9943
Epoch [3/10], Step [900/938] - Loss D: 1.2534, Loss G: 1.1667
Epoch [3/10] - Loss D: 1.3147, Loss G: 1.3145
Epoch [4/10], Step [0/938] - Loss D: 1.1130, Loss G: 1.0101
Epoch [4/10], Step [100/938] - Loss D: 1.0814, Loss G: 1.1383
Epoch [4/10], Step [200/938] - Loss D: 1.0867, Loss G: 1.2681
Epoch [4/10], Step [300/938] - Loss D: 1.3533, Loss G: 1.1584
Epoch [4/10], Step [400/938] - Loss D: 1.2968, Loss G: 1.1545
Epoch [4/10], Step [500/938] - Loss D: 1.1084, Loss G: 1.2496
Epoch [4/10], Step [600/938] - Loss D: 1.1472, Loss G: 1.3232
Epoch [4/10], Step [700/938] - Loss D: 1.1562, Loss G: 1.1646
Epoch [4/10], Step [800/938] - Loss D: 1.1374, Loss G: 1.2087
Epoch [4/10], Step [900/938] - Loss D: 1.0737, Loss G: 1.3388
Epoch [4/10] - Loss D: 1.1861, Loss G: 1.3884
Epoch [5/10], Step [0/938] - Loss D: 1.3358, Loss G: 1.2203
Epoch [5/10], Step [100/938] - Loss D: 1.0586, Loss G: 1.4459
Epoch [5/10], Step [200/938] - Loss D: 1.2494, Loss G: 1.0211
Epoch [5/10], Step [300/938] - Loss D: 1.1608, Loss G: 1.2711
Epoch [5/10], Step [400/938] - Loss D: 1.2154, Loss G: 1.1134
Epoch [5/10], Step [500/938] - Loss D: 1.2878, Loss G: 1.0566
Epoch [5/10], Step [600/938] - Loss D: 1.1624, Loss G: 1.1200
Epoch [5/10], Step [700/938] - Loss D: 1.1896, Loss G: 1.1969
Epoch [5/10], Step [800/938] - Loss D: 1.0975, Loss G: 1.1803
Epoch [5/10], Step [900/938] - Loss D: 1.2893, Loss G: 1.1333
Epoch [5/10] - Loss D: 1.1560, Loss G: 0.9840
Epoch [6/10], Step [0/938] - Loss D: 1.3727, Loss G: 1.0725
Epoch [6/10], Step [100/938] - Loss D: 1.1192, Loss G: 1.3813
Epoch [6/10], Step [200/938] - Loss D: 1.1887, Loss G: 1.1047
Epoch [6/10], Step [300/938] - Loss D: 1.1239, Loss G: 1.1118
Epoch [6/10], Step [400/938] - Loss D: 1.1259, Loss G: 0.9888
```

```
Epoch [6/10], Step [500/938] - Loss D: 1.3369, Loss G: 1.0425
Epoch [6/10], Step [600/938] - Loss D: 1.0932, Loss G: 1.3251
Epoch [6/10], Step [700/938] - Loss D: 1.2508, Loss G: 1.0329
Epoch [6/10], Step [800/938] - Loss D: 1.2708, Loss G: 1.1134
Epoch [6/10], Step [900/938] - Loss D: 1.2786, Loss G: 1.0333
Epoch [6/10] - Loss D: 1.3344, Loss G: 1.0090
Epoch [7/10], Step [0/938] - Loss D: 1.3418, Loss G: 1.0468
Epoch [7/10], Step [100/938] - Loss D: 1.1912, Loss G: 1.0972
Epoch [7/10], Step [200/938] - Loss D: 1.3381, Loss G: 1.0244
Epoch [7/10], Step [300/938] - Loss D: 1.2215, Loss G: 1.0791
Epoch [7/10], Step [400/938] - Loss D: 1.2464, Loss G: 0.9522
Epoch [7/10], Step [500/938] - Loss D: 1.3262, Loss G: 0.9279
Epoch [7/10], Step [600/938] - Loss D: 1.2361, Loss G: 0.9948
Epoch [7/10], Step [700/938] - Loss D: 1.2561, Loss G: 0.9930
Epoch [7/10], Step [800/938] - Loss D: 1.2900, Loss G: 1.1389
Epoch [7/10], Step [900/938] - Loss D: 1.3164, Loss G: 0.9969
Epoch [7/10] - Loss D: 1.2618, Loss G: 1.0301
Epoch [8/10], Step [0/938] - Loss D: 1.1521, Loss G: 1.0985
Epoch [8/10], Step [100/938] - Loss D: 1.2694, Loss G: 1.0037
Epoch [8/10], Step [200/938] - Loss D: 1.3969, Loss G: 1.0540
Epoch [8/10], Step [300/938] - Loss D: 1.1552, Loss G: 1.0067
Epoch [8/10], Step [400/938] - Loss D: 1.2111, Loss G: 0.9477
Epoch [8/10], Step [500/938] - Loss D: 1.2242, Loss G: 1.0043
Epoch [8/10], Step [600/938] - Loss D: 1.2233, Loss G: 0.9786
Epoch [8/10], Step [700/938] - Loss D: 1.2247, Loss G: 0.9389
Epoch [8/10], Step [800/938] - Loss D: 1.2148, Loss G: 1.0974
Epoch [8/10], Step [900/938] - Loss D: 1.2667, Loss G: 0.9837
Epoch [8/10] - Loss D: 1.2261, Loss G: 1.0663
Epoch [9/10], Step [0/938] - Loss D: 1.2781, Loss G: 0.9529
Epoch [9/10], Step [100/938] - Loss D: 1.2337, Loss G: 0.9210
Epoch [9/10], Step [200/938] - Loss D: 1.4049, Loss G: 0.9985
Epoch [9/10], Step [300/938] - Loss D: 1.2789, Loss G: 1.0521
Epoch [9/10], Step [400/938] - Loss D: 1.2446, Loss G: 0.8586
Epoch [9/10], Step [500/938] - Loss D: 1.2458, Loss G: 1.0125
Epoch [9/10], Step [600/938] - Loss D: 1.2974, Loss G: 0.9722
Epoch [9/10], Step [700/938] - Loss D: 1.2539, Loss G: 1.0063
Epoch [9/10], Step [800/938] - Loss D: 1.3900, Loss G: 0.9095
Epoch [9/10], Step [900/938] - Loss D: 1.2573, Loss G: 1.0705
Epoch [9/10] - Loss D: 1.3396, Loss G: 1.2987
Epoch [10/10], Step [0/938] - Loss D: 1.4320, Loss G: 1.1427
Epoch [10/10], Step [100/938] - Loss D: 1.2662, Loss G: 0.9450
Epoch [10/10], Step [200/938] - Loss D: 1.2569, Loss G: 1.1190
Epoch [10/10], Step [300/938] - Loss D: 1.2631, Loss G: 0.9562
Epoch [10/10], Step [400/938] - Loss D: 1.2006, Loss G: 0.9215
Epoch [10/10], Step [500/938] - Loss D: 1.3012, Loss G: 1.0343
Epoch [10/10], Step [600/938] - Loss D: 1.3259, Loss G: 0.8457
Epoch [10/10], Step [700/938] - Loss D: 1.2438, Loss G: 0.9470
Epoch [10/10], Step [800/938] - Loss D: 1.1493, Loss G: 0.9964
Epoch [10/10], Step [900/938] - Loss D: 1.2161, Loss G: 0.9815
Epoch [10/10] - Loss D: 1.2939, Loss G: 1.0754
```

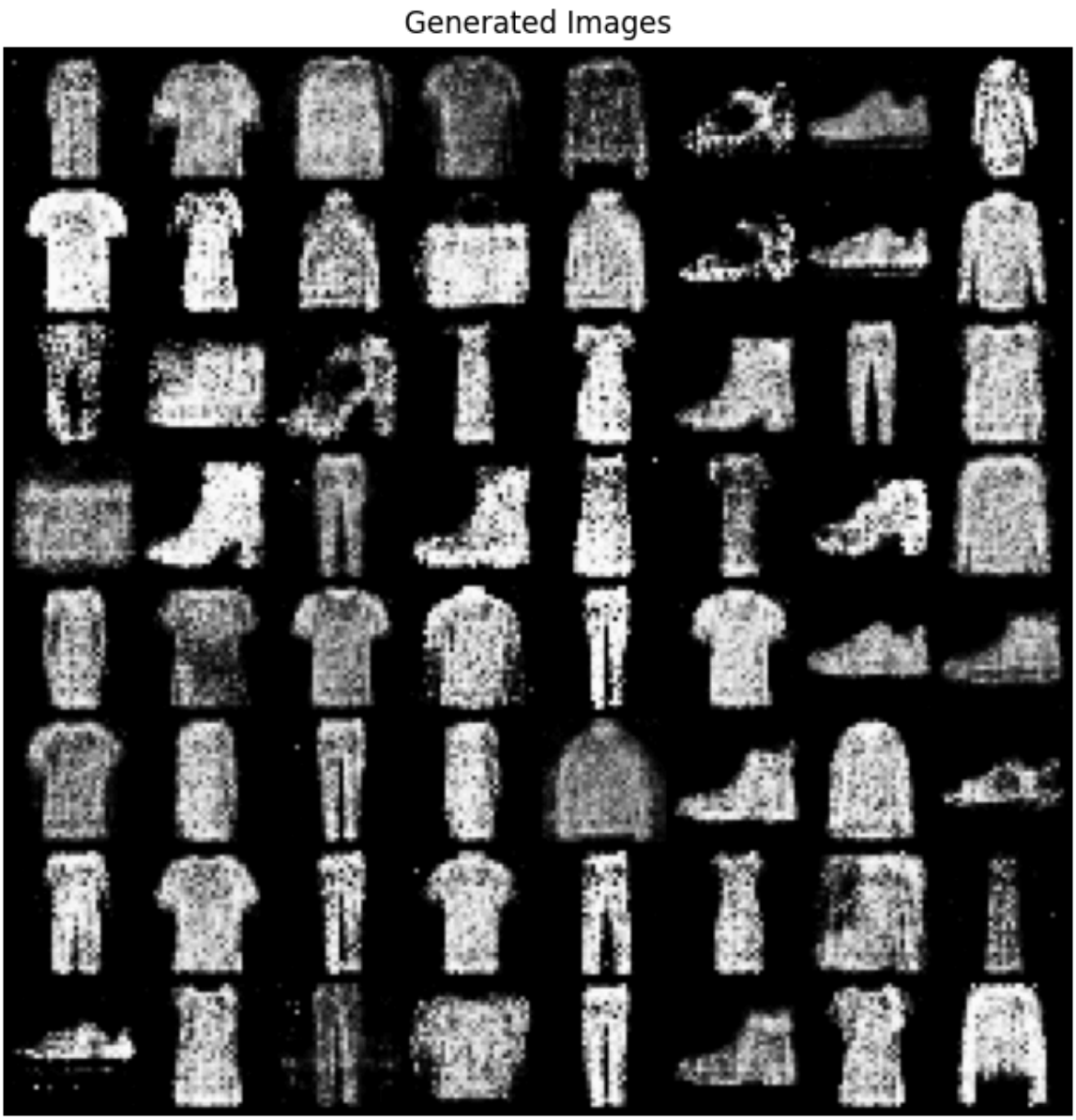
In [ ]:

In [10]:
```python
# Generate synthetic images
with torch.no_grad():
    noise = torch.randn(64, noise_dim)
    generated_images = generator(noise)
```

In [12]:
```
# Plot loss curves
plt.figure()
plt.plot(range(1, num_epochs+1), losses_D, label='Discriminator Loss')
plt.plot(range(1, num_epochs+1), losses_G, label='Generator Loss')
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.legend()
plt.title("Training Loss Curves")
plt.show()
```




In [13]:
```
# Visualize generated images
generated_images = generated_images.cpu().detach()
grid = make_grid(generated_images, nrow=8, normalize=True)
plt.figure(figsize=(8, 8))
plt.imshow(grid.permute(1, 2, 0).numpy())
plt.title("Generated Images")
plt.axis('off')
plt.show()
```

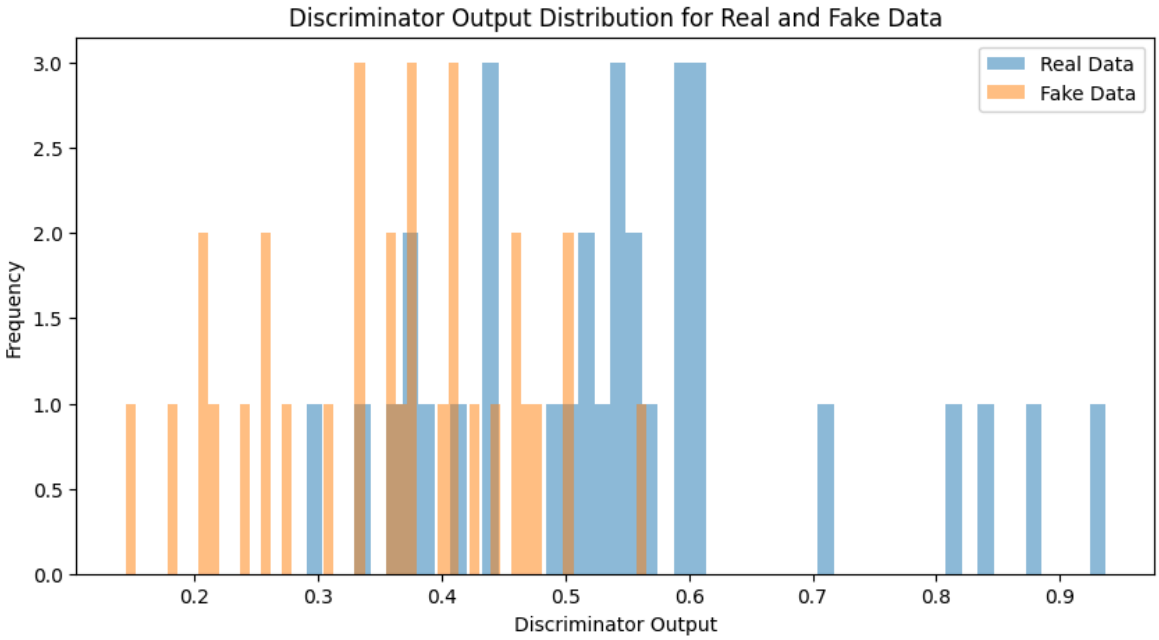



In [18]:
```
# Plot histograms of discriminator outputs for real and fake data
plt.figure(figsize=(10, 5))

# Real data
plt.hist(d_real.detach().cpu().numpy(), bins=50, alpha=0.5, label='Real Data')

# Fake data
plt.hist(d_fake.detach().cpu().numpy(), bins=50, alpha=0.5, label='Fake Data')

plt.xlabel('Discriminator Output')
plt.ylabel('Frequency')
plt.legend()
plt.title('Discriminator Output Distribution for Real and Fake Data')
plt.show()
```




# Conclusion

## Results

In this project, we successfully implemented a Generative Adversarial Network (GAN) to generate synthetic images that resemble the images in the FashionMNIST dataset. The GAN consists of a Generator and a Discriminator network, both of which were trained using the FashionMNIST dataset. The training process involved alternating between training the Discriminator to distinguish between real and fake images and training the Generator to produce realistic images that can fool the Discriminator.

The results of the training process are as follows:

- The Discriminator and Generator losses were tracked over 10 epochs, showing the progress of the training.
- Synthetic images generated by the Generator were visualized, demonstrating the ability of the GAN to produce images that resemble fashion items.
- The distribution of the Discriminator's outputs for real and fake data was analyzed, providing insights into the performance of the Discriminator.

## Improvements

While the GAN produced promising results, there are several areas for improvement:

1. **Increase Training Epochs**: Training the GAN for more epochs could lead to better results as the networks have more time to learn and improve.
2. **Hyperparameter Tuning**: Experimenting with different hyperparameters, such as learning rates, batch sizes, and network architectures, could enhance the performance of the GAN.

3. **Data Augmentation**: Applying data augmentation techniques to the training data could help the GAN learn more robust features and improve the quality of the generated images.
4. **Advanced GAN Architectures**: Exploring advanced GAN architectures, such as DCGAN, WGAN, or StyleGAN, could lead to better image generation results.
5. **Evaluation Metrics**: Implementing additional evaluation metrics, such as Inception Score (IS) or Frechet Inception Distance (FID), could provide a more comprehensive assessment of the GAN's performance.

By addressing these areas, the GAN's ability to generate high-quality synthetic images can be further improved.