Deep Learning - ECE-GY 7123
Homework 3
Giorgi Merabishvili, gm3386 May 6

Problem 1

Part A: Gradient Calculation and Update Rule

The probability of selecting action a_i is defined by the softmax function:

$$\pi(a_i) = \frac{e^{\theta_i}}{\sum_{j=1}^{k} e^{\theta_j}}$$

The gradient of the policy with respect to its parameters θ_i is derived as follows:

$$\frac{\partial \pi(a_i)}{\partial \theta_i} = \frac{e^{\theta_i}\left(\sum_{j=1}^{k} e^{\theta_j} - e^{\theta_i}\right)}{\left(\sum_{j=1}^{k} e^{\theta_j}\right)^2} = \pi(a_i)(1 - \pi(a_i))$$

The update rule for the parameters, when action a_i is selected and a reward R_i is observed, is given by:

$$\Delta\theta_i = \eta R_i(1 - \pi(a_i))$$

This rule means that the adjustment to θ_i is directly proportional to the reward R_i and the potential for increase in the probability of choosing a_i, reflecting an incentive to increase actions that yield higher rewards.

Part B: Ensuring Convergence of Parameters

Challenges with Constant Step Sizes

Constant step sizes can lead to significant training instability. This instability arises because a constant step size does not adapt to the variability in reward scales and gradient magnitudes, potentially causing the training process to oscillate or even diverge, particularly when η is too large.

Adaptive Step Size Methods

Adaptive step size methods such as RMSProp and Adam are essential to stabilize training in environments with high variability in rewards and gradients.

RMSProp RMSProp adjusts the learning rate by using an exponential moving average of the squared gradients. This adjustment helps in reducing the step size when the gradients are large, thus preventing large swings in parameter updates and aiding convergence:

$$\eta_t = \frac{\eta}{\sqrt{EMA(\nabla\theta^2) + \epsilon}}$$

Adam Adam builds on RMSProp by incorporating momentum, considering both the average and the variance of past gradients to determine the optimal learning rate, facilitating smoother and more stable updates:

$$\eta_t = \frac{\eta}{\sqrt{EMA(\nabla\theta^2) + \epsilon}} \cdot m_t$$

where m_t represents the momentum term, calculated as a moving average of past gradients.

Decaying Learning Rate

Scheduled Decay Implementing a decaying learning rate that reduces over time ensures that the learning rate does not remain excessively high as the training progresses. This can be a simple time-based decay or a performance-based decay, where the reduction happens when the improvement in policy performance plateaus.

Convergence Criteria Establishing convergence criteria based on the stability of policy performance or the smallness of policy updates can also assist in ensuring convergence. Monitoring the change in policy or the reward variance, and reducing the learning rate if changes fall below a specific threshold, provides a practical mechanism for managing the convergence of parameters effectively.

Problem 2

Minimax Optimization

Function:

$$f(x, y) = 4x^2 - 4y^2$$

This quadratic function is symmetric with respect to the x and y axes, but with opposite signs for their quadratic terms, indicating opposite behaviors along each axis.

a. Saddle Point Determination

A saddle point is defined as a point where the function transitions from a minimum to a maximum along orthogonal directions. For the function f, compute the gradient:

$$\nabla f(x, y) = (8x, -8y)$$

Setting this gradient equal to zero provides the equations:

$$8x = 0, \quad -8y = 0$$

Solving these, we find:

$$x = 0, \quad y = 0$$

Thus, the saddle point is at $(0, 0)$. At this point, the function has a minimum along the x-axis and a maximum along the y-axis due to the respective positive and negative coefficients of x^2 and y^2.

b. Gradient Descent/Ascent Equations

The minimization and maximization processes can be modeled using gradient descent and ascent, respectively. Starting from an initial point (x_0, y_0), the iterative updates are:

$$x_{n+1} = x_n - \eta \frac{\partial f}{\partial x} = x_n - \eta(8x_n)$$

$$y_{n+1} = y_n + \eta \frac{\partial f}{\partial y} = y_n + \eta(-8y_n)$$

Here, η is the step size, and the negative and positive signs represent descent and ascent, respectively, reflecting the efforts to minimize and maximize f.

c. Allowable Step Sizes for Convergence

To ensure the stability and convergence of the iterative process, the step size η must be carefully selected. Considering the gradient's Lipschitz constant $L = 8$, the step size condition for convergence is derived from the stability criterion in numerical methods:

$$0 < \eta < \frac{2}{L}$$

This translates to:

$$0 < \eta < \frac{1}{4}$$

Choosing a step size within this range helps avoid overshooting the saddle point and ensures smooth convergence.

d. Regular Gradient Descent Dynamics

Applying regular gradient descent to both x and y would typically aim to find a local minimum of f, but because of the mixed quadratic terms, such a method would converge to either the global minimum or maximum, depending on the initialization and absence of balanced minimization and maximization strategies:

$$x_{n+1} = x_n - \eta \cdot 8x_n$$

$$y_{n+1} = y_n - \eta \cdot (-8y_n)$$

Under certain initial conditions and specific values of η, this might accidentally converge to the saddle point, but generally, it seeks a minimum along both axes, which is not present except at infinity.

HW3

Giorgi Merabishvili - gm3386

untitled0-7

May 7, 2024

[1]:
```
# imports for PyTorch and other necessary modules
import torch
from torch import nn, optim
import torchvision
from torchvision.datasets import FashionMNIST
from torchvision import transforms
from torch.utils.data import DataLoader
import matplotlib.pyplot as plt
from tqdm import tqdm
import numpy as np
```

Use the following generator architecture (which is essentially the reverse of a standard discriminative architecture). You can use the same kernel size. Construct: • a dense layer that takes a unit Gaussian noise vector of length 100 and mapsittoavectorofsize7∗7∗256. Nobiasterms. • severaltranspose2Dconvolutions($256\times7\times7 \rightarrow 128\times7\times7 \rightarrow 64 \times 14 \times 14 \rightarrow 1 \times 28 \times 28$). No bias terms. • each convolutional layer (except the last one) is equipped with Batch Normalization (batch norm), followed by Leaky ReLU with slope 0.3. The last (output) layer is equipped with tanh activation (no batch norm).

[2]:
```
# Generator class for producing image from noise
# ConvTranspose2d for upscaling, BatchNorm for stabilization, and LeakyReLU for␣
 ↪non-linearity
class Generator(nn.Module):
    def __init__(self, noise_channels, image_channels, features):
        super(Generator, self).__init__()
        self.features = features
        self.linear = nn.Linear(noise_channels, self.features * 4 * 7 * 7,␣
 ↪bias=False)
        self.model = nn.Sequential(
            nn.ConvTranspose2d(self.features * 4, self.features * 2, 5, 1, 2,␣
 ↪bias=False),
            nn.BatchNorm2d(self.features * 2),
            nn.LeakyReLU(0.3),
            nn.ConvTranspose2d(self.features * 2, self.features, 5, 2, 2, 1,␣
 ↪bias=False),
            nn.BatchNorm2d(self.features),
            nn.LeakyReLU(0.3),
```

```
            nn.ConvTranspose2d(self.features, image_channels, 5, 2, 2, 1,␣
 ↪bias=False),
            nn.Tanh()
        )

    def forward(self, x):
        x = self.linear(x)
        x = x.view(-1, self.features * 4, 7, 7)
        return self.model(x)
```

Use the following discriminator architecture (kernel size = 5×5 with stride = 2 in both directions): • 2Dconvolutions($1\times28\times28 \to 64\times14\times14 \to 128\times7\times7$) • each convolutional layer is equipped with a Leaky ReLU with slope 0.3, followed by Dropout with parameter 0.3. • a dense layer that takes the flattened output of the last convolution and maps it to a scalar.

[3]:
```
# Discriminator class to classify real vs fake images
# Conv2d for feature extraction, Dropout for regularization, and Sigmoid for␣
 ↪output activation
class Discriminator(nn.Module):
    def __init__(self, image_channels, features=64):
        super(Discriminator, self).__init__()
        self.model = nn.Sequential(
            nn.Conv2d(image_channels, features, 5, 2, 2),
            nn.LeakyReLU(0.3),
            nn.Dropout(0.3),
            nn.Conv2d(features, features * 2, 5, 2, 2),
            nn.LeakyReLU(0.3),
            nn.Dropout(0.3),
            nn.Flatten(),
            nn.Linear(features * 2 * 7 * 7, 1),
            nn.Sigmoid()
        )

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

[4]:
```
# Setting up the device, learning rate, batch size, and number of epochs
# Device setup to use GPU if available
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
LEARNING_RATE = 1e-4
BATCH_SIZE = 64
EPOCHS = 50
```

Use the FashionMNIST training dataset (which we used in previous assign- ments) to train the DCGAN. Images are grayscale and size 28×28.

[5]:
```
# Prepare transformations, load FashionMNIST dataset
# DataLoader does batch-wise iteration over dataset
transforms = transforms.Compose([
    transforms.Resize(28),
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))
])

dataset = FashionMNIST(root="./data", train=True, transform=transforms,␣
 ↪download=True)
dataloader = DataLoader(dataset, batch_size=BATCH_SIZE, shuffle=True)
```

Use the binary cross-entropy loss for training both the generator and the discriminator. Use the Adam optimizer with learning rate 10^−4.

[6]:
```
# Set up generator and discriminator models, optimizers, and loss function
gen_model = Generator(100, 1, 64).to(device)
disc_model = Discriminator(1, 64).to(device)
gen_optimizer = optim.Adam(gen_model.parameters(), lr=LEARNING_RATE, betas=(0.
 ↪5, 0.999))
disc_optimizer = optim.Adam(disc_model.parameters(), lr=LEARNING_RATE, betas=(0.
 ↪5, 0.999))
criterion = nn.BCELoss()
```

[7]:
```
print("Discriminator parameters:", sum(p.numel() for p in disc_model.
 ↪parameters()))
print("Generator parameters:", sum(p.numel() for p in gen_model.parameters()))
```

```
Discriminator parameters: 212865
Generator parameters: 2280384
```

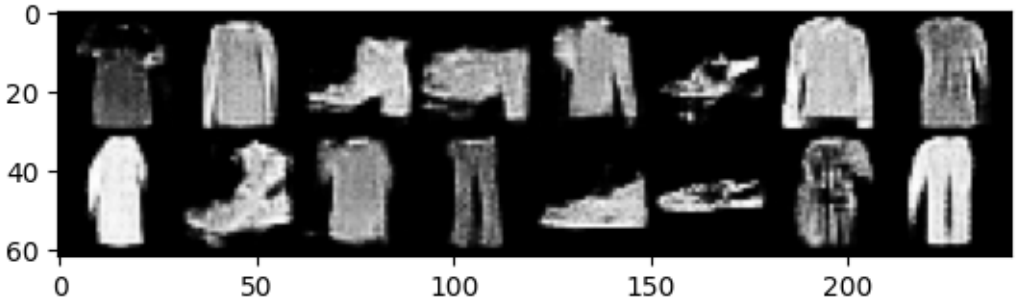
Train it for 50 epochs. You can use minibatch sizes of 16, 32, or 64. Training may take several minutes (or even up to an hour), so be patient! Display intermediate images generated after T = 10, T = 30, and T = 50 epochs.

[8]:
```
# Training loop for the GAN
# Displays loss statistics and generated images at specified epochs - 10th,␣
 ↪30th, 50th

disc_losses = []
gen_losses = []

for epoch in range(EPOCHS):
    loop = tqdm(enumerate(dataloader), total=len(dataloader), leave=True)
    for i, (images, _) in loop:
        images = images.to(device)
        real_labels = torch.ones(images.size(0), 1, device=device)
        fake_labels = torch.zeros(images.size(0), 1, device=device)
```

```
        real_outputs = disc_model(images)
        d_loss_real = criterion(real_outputs, real_labels)

        noise = torch.randn(images.size(0), 100, device=device)
        fake_images = gen_model(noise)
        fake_outputs = disc_model(fake_images.detach())
        d_loss_fake = criterion(fake_outputs, fake_labels)

        d_loss = d_loss_real + d_loss_fake
        disc_optimizer.zero_grad()
        d_loss.backward()
        disc_optimizer.step()

        outputs = disc_model(fake_images)
        g_loss = criterion(outputs, real_labels)
        gen_optimizer.zero_grad()
        g_loss.backward()
        gen_optimizer.step()

        disc_losses.append(d_loss.item())
        gen_losses.append(g_loss.item())

        loop.set_description(f"Epoch [{epoch + 1}/{EPOCHS}] :: Batch [{i + 1}/
↪{len(dataloader)}]")
        loop.set_postfix(Dloss=d_loss.item(), Gloss=g_loss.item())

    if (epoch + 1) in [10, 30, 50]:
        print(f'Epoch [{epoch + 1}/{EPOCHS}], d_loss: {d_loss.item():.4f},␣
↪g_loss: {g_loss.item():.4f}')
        with torch.no_grad():
            fixed_noise = torch.randn(16, 100, device=device)
            fake_images = gen_model(fixed_noise)
            img_grid = torchvision.utils.make_grid(fake_images, normalize=True)
            plt.imshow(img_grid.permute(1, 2, 0).cpu().numpy())
            plt.show()

# Save models
torch.save(gen_model.state_dict(), 'generator.pth')
torch.save(disc_model.state_dict(), 'discriminator.pth')
```

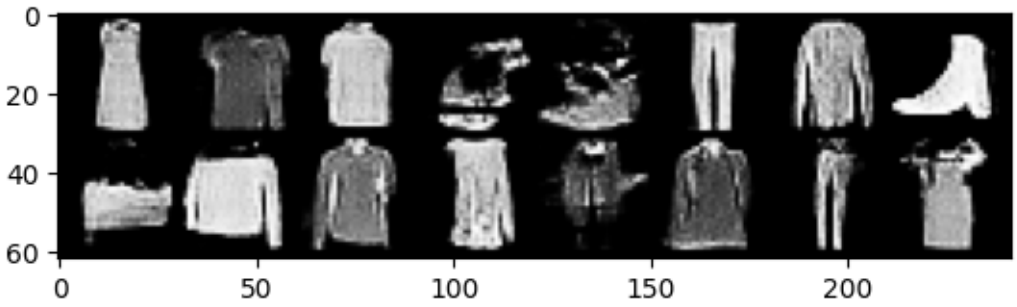
```
Epoch [1/50] :: Batch [938/938]: 100%|      | 938/938 [00:24<00:00,
38.33it/s, Dloss=1.19, Gloss=0.87]
Epoch [2/50] :: Batch [938/938]: 100%|      | 938/938 [00:23<00:00,
40.50it/s, Dloss=1.07, Gloss=1.12]
Epoch [3/50] :: Batch [938/938]: 100%|      | 938/938 [00:23<00:00,
39.89it/s, Dloss=1.12, Gloss=1.08]
```

```
Epoch [4/50] :: Batch [938/938]: 100%|      | 938/938 [00:23<00:00,
39.82it/s, Dloss=1.25, Gloss=1.06]
Epoch [5/50] :: Batch [938/938]: 100%|      | 938/938 [00:23<00:00,
40.16it/s, Dloss=1.21, Gloss=0.931]
Epoch [6/50] :: Batch [938/938]: 100%|      | 938/938 [00:23<00:00,
39.77it/s, Dloss=1.23, Gloss=0.836]
Epoch [7/50] :: Batch [938/938]: 100%|      | 938/938 [00:23<00:00,
39.52it/s, Dloss=1.21, Gloss=0.764]
Epoch [8/50] :: Batch [938/938]: 100%|      | 938/938 [00:23<00:00,
39.63it/s, Dloss=1.22, Gloss=0.898]
Epoch [9/50] :: Batch [938/938]: 100%|      | 938/938 [00:23<00:00,
40.00it/s, Dloss=1.28, Gloss=1.14]
Epoch [10/50] :: Batch [938/938]: 100%|      | 938/938 [00:22<00:00,
40.81it/s, Dloss=1.32, Gloss=1.04]

Epoch [10/50], d_loss: 1.3183, g_loss: 1.0445
```



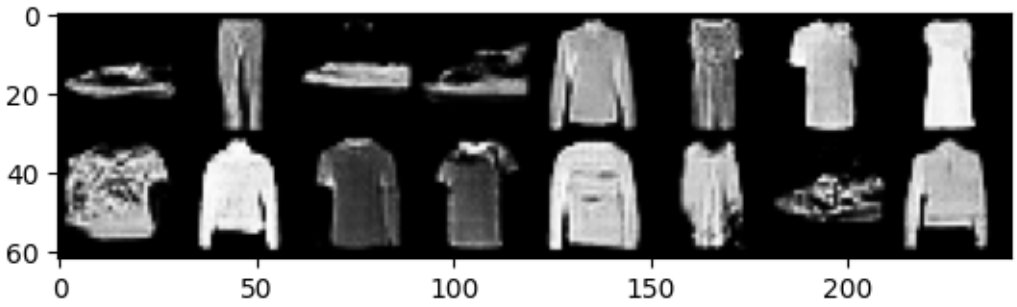

```
Epoch [11/50] :: Batch [938/938]: 100%|      | 938/938 [00:23<00:00,
40.19it/s, Dloss=1.29, Gloss=1.01]
Epoch [12/50] :: Batch [938/938]: 100%|      | 938/938 [00:22<00:00,
40.89it/s, Dloss=1.11, Gloss=0.954]
Epoch [13/50] :: Batch [938/938]: 100%|      | 938/938 [00:22<00:00,
40.82it/s, Dloss=1.27, Gloss=0.772]
Epoch [14/50] :: Batch [938/938]: 100%|      | 938/938 [00:23<00:00,
40.32it/s, Dloss=1.42, Gloss=0.833]
Epoch [15/50] :: Batch [938/938]: 100%|      | 938/938 [00:23<00:00,
40.48it/s, Dloss=1.36, Gloss=0.86]
Epoch [16/50] :: Batch [938/938]: 100%|      | 938/938 [00:23<00:00,
40.63it/s, Dloss=1.18, Gloss=0.855]
Epoch [17/50] :: Batch [938/938]: 100%|      | 938/938 [00:23<00:00,
40.64it/s, Dloss=1.12, Gloss=0.848]
Epoch [18/50] :: Batch [938/938]: 100%|      | 938/938 [00:23<00:00,
40.50it/s, Dloss=1.25, Gloss=0.963]
Epoch [19/50] :: Batch [938/938]: 100%|      | 938/938 [00:23<00:00,
40.46it/s, Dloss=1.27, Gloss=0.794]
Epoch [20/50] :: Batch [938/938]: 100%|      | 938/938 [00:23<00:00,
```

```
40.55it/s, Dloss=1.29, Gloss=0.83]
Epoch [21/50] :: Batch [938/938]: 100%|      | 938/938 [00:23<00:00,
40.33it/s, Dloss=1.39, Gloss=0.831]
Epoch [22/50] :: Batch [938/938]: 100%|      | 938/938 [00:23<00:00,
39.36it/s, Dloss=1.3, Gloss=0.926]
Epoch [23/50] :: Batch [938/938]: 100%|      | 938/938 [00:23<00:00,
40.76it/s, Dloss=1.32, Gloss=0.809]
Epoch [24/50] :: Batch [938/938]: 100%|      | 938/938 [00:23<00:00,
40.76it/s, Dloss=1.24, Gloss=0.984]
Epoch [25/50] :: Batch [938/938]: 100%|      | 938/938 [00:23<00:00,
40.53it/s, Dloss=1.24, Gloss=0.872]
Epoch [26/50] :: Batch [938/938]: 100%|      | 938/938 [00:22<00:00,
40.85it/s, Dloss=1.32, Gloss=0.881]
Epoch [27/50] :: Batch [938/938]: 100%|      | 938/938 [00:22<00:00,
41.01it/s, Dloss=1.2, Gloss=0.796]
Epoch [28/50] :: Batch [938/938]: 100%|      | 938/938 [00:22<00:00,
41.77it/s, Dloss=1.33, Gloss=0.929]
Epoch [29/50] :: Batch [938/938]: 100%|      | 938/938 [00:22<00:00,
40.83it/s, Dloss=1.32, Gloss=0.93]
Epoch [30/50] :: Batch [938/938]: 100%|      | 938/938 [00:22<00:00,
40.95it/s, Dloss=1.33, Gloss=0.984]

Epoch [30/50], d_loss: 1.3350, g_loss: 0.9837
```




```
Epoch [31/50] :: Batch [938/938]: 100%|      | 938/938 [00:23<00:00,
40.77it/s, Dloss=1.19, Gloss=0.953]
Epoch [32/50] :: Batch [938/938]: 100%|      | 938/938 [00:23<00:00,
40.48it/s, Dloss=1.26, Gloss=0.827]
Epoch [33/50] :: Batch [938/938]: 100%|      | 938/938 [00:23<00:00,
40.66it/s, Dloss=1.27, Gloss=0.926]
Epoch [34/50] :: Batch [938/938]: 100%|      | 938/938 [00:23<00:00,
40.73it/s, Dloss=1.31, Gloss=0.785]
Epoch [35/50] :: Batch [938/938]: 100%|      | 938/938 [00:23<00:00,
40.77it/s, Dloss=1.29, Gloss=0.906]
Epoch [36/50] :: Batch [938/938]: 100%|      | 938/938 [00:23<00:00,
40.62it/s, Dloss=1.32, Gloss=0.816]
```

```
Epoch [37/50] :: Batch [938/938]: 100%|      | 938/938 [00:23<00:00,
40.26it/s, Dloss=1.32, Gloss=0.92]
Epoch [38/50] :: Batch [938/938]: 100%|      | 938/938 [00:23<00:00,
40.64it/s, Dloss=1.35, Gloss=0.841]
Epoch [39/50] :: Batch [938/938]: 100%|      | 938/938 [00:22<00:00,
40.80it/s, Dloss=1.24, Gloss=0.964]
Epoch [40/50] :: Batch [938/938]: 100%|      | 938/938 [00:23<00:00,
40.73it/s, Dloss=1.3, Gloss=0.837]
Epoch [41/50] :: Batch [938/938]: 100%|      | 938/938 [00:22<00:00,
40.99it/s, Dloss=1.28, Gloss=0.888]
Epoch [42/50] :: Batch [938/938]: 100%|      | 938/938 [00:22<00:00,
40.99it/s, Dloss=1.22, Gloss=1.03]
Epoch [43/50] :: Batch [938/938]: 100%|      | 938/938 [00:23<00:00,
39.73it/s, Dloss=1.47, Gloss=0.792]
Epoch [44/50] :: Batch [938/938]: 100%|      | 938/938 [00:23<00:00,
39.13it/s, Dloss=1.17, Gloss=0.869]
Epoch [45/50] :: Batch [938/938]: 100%|      | 938/938 [00:24<00:00,
38.89it/s, Dloss=1.28, Gloss=0.837]
Epoch [46/50] :: Batch [938/938]: 100%|      | 938/938 [00:23<00:00,
39.76it/s, Dloss=1.24, Gloss=1.04]
Epoch [47/50] :: Batch [938/938]: 100%|      | 938/938 [00:23<00:00,
39.45it/s, Dloss=1.23, Gloss=0.784]
Epoch [48/50] :: Batch [938/938]: 100%|      | 938/938 [00:23<00:00,
39.95it/s, Dloss=1.43, Gloss=0.877]
Epoch [49/50] :: Batch [938/938]: 100%|      | 938/938 [00:23<00:00,
40.27it/s, Dloss=1.16, Gloss=0.952]
Epoch [50/50] :: Batch [938/938]: 100%|      | 938/938 [00:24<00:00,
38.98it/s, Dloss=1.27, Gloss=0.882]

Epoch [50/50], d_loss: 1.2707, g_loss: 0.8822
```



[9]:
```python
# Function to smooth losses for smoother plot
def smooth_curve(points, factor=0.8):
    smoothed_points = []
    for point in points:
```

```
        if smoothed_points:
            previous = smoothed_points[-1]
            smoothed_points.append(previous * factor + point * (1 - factor))
        else:
            smoothed_points.append(point)
    return smoothed_points

# Smooth losses for visualization
smoothed_disc_losses = smooth_curve(disc_losses)
smoothed_gen_losses = smooth_curve(gen_losses)

# Plotting the losses
plt.figure(figsize=(10, 5))
plt.title("Generator and Discriminator Loss During Training", fontsize=14)
plt.plot(smoothed_disc_losses, label="Discriminator", linewidth=2)
plt.plot(smoothed_gen_losses, label="Generator", linewidth=2)
plt.xlabel("Iterations", fontsize=12)
plt.ylabel("Loss", fontsize=12)
plt.legend()
plt.grid(True)
plt.show()
```

0.1 Observations

I observed notable advances in the generator's performance during GAN training on the FashionMNIST dataset, as it gradually started to produce more realistic images. The discriminator performed better in the beginning, but as learning rates and optimizer parameters were changed,

the two models eventually came into balance, minimizing problems like mode collapse. The ultimate image quality was higher, demonstrating the GAN's ability to produce realistic images.

Explanation(!Comments in the code also explains the code)

1. **Generator and Discriminator Architecture**

Generator: Takes a random noise vector and uses several layers of transposed convolutions (also known as deconvolutions) to upscale this input into a 28x28 image. Batch normalization and LeakyReLU activation (except in the output layer, which uses a tanh activation) are used to stabilize training and introduce non-linearity.

Discriminator: Takes an input image and uses convolutions to downscale it to a single scalar output that predicts whether the input image is real or fake. It uses LeakyReLU for activation and dropout for regularization to prevent overfitting.

2. **Setup and Data Loading**

Device Setup: Configures the PyTorch device to use GPU if available for faster computation.

Data Loading: The FashionMNIST dataset, which contains 28x28 grayscale images of fashion items, is loaded. A series of transformations (resizing, converting to tensor, and normalizing) are applied to prepare the data for the network.

3. **Training Process**

Loss Function: Binary cross-entropy loss is used for both the generator and discriminator.

Optimizers: The Adam optimizer is used for both models with a specific learning rate and beta parameters to control the learning dynamics.

Training Loop: In each epoch, the discriminator is trained first by optimizing its ability to distinguish real images from fake ones. The generator is then trained to fool the discriminator by optimizing towards getting the discriminator to predict 'real' for generated images. Losses are recorded and periodically, generated images are displayed to monitor the training progress.

4. **Output Visualization**

Plotting Losses: The losses for both generator and discriminator are plotted to observe the training process. This can help identify issues like mode collapse or whether the generator is overpowering the discriminator or vice versa.

Generated Images: At specified epochs, images generated by the generator are displayed to visually assess the improvement and realism of generated images over training.