Deep Learning (CS-GY 6953)

Homework 3

Aditya Mittal - am13294

PROBLEM 1

Understanding policy gradients. In class we derived a general form of policy gradients. Let us consider a special case here which does not involve any neural networks. Suppose the step size is η. We consider the so-called bandit setting where past actions and states do not matter, and different actions a_i give rise to different rewards R_i.

(a) **Define the mapping π such that $\pi(a_i) =$ softmax(θ_i) for $i = 1, \ldots, k$, where k is the total number of actions and θ_i is a scalar parameter encoding the value of each action. Show that if action a_i is sampled, then the change in the parameters is given by:**

$$\Delta\theta_i = \eta R_i(1 - \pi(a_i)). \tag{1}$$

Solution: The softmax function is defined as follows:

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

We want to maximize the expected reward:

$$\mathbb{E}[R] = \sum_{i=1}^{k} \pi(a_i) R_i.$$

Using the policy gradient theorem, the gradient of $\mathbb{E}[R]$ with respect to the parameter θ_i is:

$$\frac{\partial \mathbb{E}[R]}{\partial \theta_i} = \sum_{j=1}^{k} \frac{\partial \pi(a_j)}{\partial \theta_i} R_j.$$

Next, we differentiate $\pi(a_j)$ with respect to θ_i:

$$\frac{\partial \pi(a_j)}{\partial \theta_i} = \pi(a_j)\left[\delta_{ij} - \pi(a_i)\right],$$

where δ_{ij} is the Kronecker delta, which is 1 if $i = j$ and 0 otherwise. Therefore,

$$\frac{\partial \mathbb{E}[R]}{\partial \theta_i} = \pi(a_i)\left[1 - \pi(a_i)\right] R_i.$$

Now, applying the gradient ascent update rule with a step size η:

$$\Delta\theta_i = \eta \frac{\partial \mathbb{E}[R]}{\partial \theta_i} = \eta R_i \left[1 - \pi(a_i)\right].$$

This proves the desired result.

(b) **If constant step sizes are used, intuitively explain why the above update rule might lead to unstable training. How would you fix this issue to ensure convergence of the parameters?**

Solution: With constant step sizes, the update rule can lead to unstable training due to the following reasons:

(a) **Large Updates:** If the reward R_i is large and the step size η is also large, the parameter updates $\Delta\theta_i$ can be significant, leading to potential oscillations or divergence of the parameters θ_i.

(b) **Sensitivity to $\pi(a_i)$:** The term $1 - \pi(a_i)$ varies depending on the probability distribution. If $\pi(a_i)$ is close to 1, the update becomes negligible, while if $\pi(a_i)$ is close to 0, the update becomes significant.

To fix this issue and ensure convergence, you can use:

(a) **Adaptive Learning Rates:** Using optimization algorithms like Adam or RMSprop, which adjust the learning rate based on the magnitude of the gradient, can stabilize training.

(b) **Decaying Learning Rate:** Decreasing the step size η over time can help prevent large oscillations and allow for finer adjustments as training progresses.

(c) **Clipping Gradients:** Clipping the gradient values can prevent large updates when the gradients are very large.

(d) **Batch Normalization:** Using techniques like batch normalization can stabilize training by normalizing the input distributions.

Choosing any of the above techniques should help stabilize the training process.

PROBLEM 2

Minimax Optimization In this problem we will see how training GANs is somewhat fundamentally different from regular training. Consider a simple problem where we are trying to minimax a function of two scalars:

$$\min_x \max_y f(x, y) = 4x^2 - 4y^2. \tag{2}$$

(a) **Determine the saddle point of this function. A saddle point is a point (x, y) for which f attains a local minimum along one direction and a local maximum in an orthogonal direction.**

Solution:

- Differentiating $f(x, y)$ with respect to x and y:

$$\frac{\partial f}{\partial x} = 8x \quad \text{and} \quad \frac{\partial f}{\partial y} = -8y.$$

- Setting both derivatives to zero, we obtain $x = 0$ and $y = 0$.
- Thus, the stationary point is $(x, y) = (0, 0)$.
- The second derivatives are $\frac{\partial^2 f}{\partial x^2} = 8$ and $\frac{\partial^2 f}{\partial y^2} = -8$.
- Since $\frac{\partial^2 f}{\partial x^2} > 0$ and $\frac{\partial^2 f}{\partial y^2} < 0$, this point is a saddle point.

(b) **Write down the gradient descent/ascent equations for solving this problem starting at some arbitrary initialization (x_0, y_0).**

Solution:

- The gradient descent update for x is:

$$x_{k+1} = x_k - \alpha \cdot 8x_k.$$

- The gradient ascent update for y is:

$$y_{k+1} = y_k + \beta \cdot 8y_k.$$

- Here, α and β are the step sizes for x and y, respectively.

(c) **Determine the range of allowable step sizes to ensure that gradient descent/ascent converges.**

Solution:

- For x, the convergence criterion for gradient descent is:

$$|1 - \alpha \cdot 8| < 1.$$

- Simplifying this inequality, we have:

$$0 < \alpha < \frac{1}{4}.$$

- For y, the convergence criterion for gradient ascent is:

$$|1 + \beta \cdot 8| < 1.$$

- Simplifying this inequality, we have:

$$-\frac{1}{4} < \beta < 0.$$

(d) **What if you just did regular gradient descent over both variables instead? Comment on the dynamics of the updates and whether there are special cases where one might converge to the saddle point anyway.**

Solution:

- The gradient descent update for both x and y would be:

$$x_{k+1} = x_k - \alpha \cdot 8x_k \quad \text{and} \quad y_{k+1} = y_k - \alpha \cdot (-8y_k).$$

- The updates would converge to $(0, 0)$ if α is chosen in the range $0 < \alpha < \frac{1}{4}$.
- In this special case, both x and y would converge to the saddle point $(0, 0)$.
- However, in general, gradient descent is not suitable for minimax problems as it minimizes both variables, while a minimax problem requires minimizing one and maximizing the other.

Problem 3

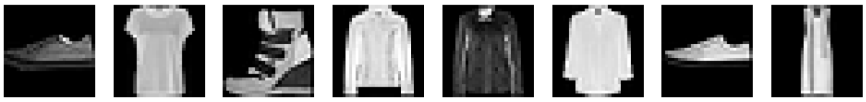
```
# Importing Libraries
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms
from torch.utils.data import DataLoader
import matplotlib.pyplot as plt
import numpy as np
import random

# Set random seed for reproducibility
random.seed(42)
torch.manual_seed(42)
```

```
<torch._C.Generator at 0x7e9c50126390>
```

```
# Preparing the Data
transform = transforms.Compose([transforms.ToTensor(),
                                transforms.Normalize((0.5,), (0.5,))])

trainset = torchvision.datasets.FashionMNIST(root='./data', train=True,
                                             download=True, transform=transform)
trainloader = DataLoader(trainset, batch_size=64, shuffle=True)
```

```
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-images-idx3-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-images-idx3-ubyte.gz to ./data/FashionMNIST/raw/train-ima
100%|██████████| 26421880/26421880 [00:02<00:00, 9324344.17it/s]
Extracting ./data/FashionMNIST/raw/train-images-idx3-ubyte.gz to ./data/FashionMNIST/raw

Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-labels-idx1-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-labels-idx1-ubyte.gz to ./data/FashionMNIST/raw/train-lab
100%|██████████| 29515/29515 [00:00<00:00, 157035.94it/s]
Extracting ./data/FashionMNIST/raw/train-labels-idx1-ubyte.gz to ./data/FashionMNIST/raw

Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-images-idx3-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-images-idx3-ubyte.gz to ./data/FashionMNIST/raw/t10k-image
100%|██████████| 4422102/4422102 [00:01<00:00, 2904232.09it/s]
Extracting ./data/FashionMNIST/raw/t10k-images-idx3-ubyte.gz to ./data/FashionMNIST/raw

Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-labels-idx1-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-labels-idx1-ubyte.gz to ./data/FashionMNIST/raw/t10k-label
100%|██████████| 5148/5148 [00:00<00:00, 5583728.21it/s]Extracting ./data/FashionMNIST/raw/t10k-labels-idx1-ubyte.gz to ./data/Fashion
```

```
# Visualizing the Data
dataiter = iter(trainloader)
images, labels = next(dataiter)

# Plot the first 8 images
fig, axes = plt.subplots(1, 8, figsize=(12, 3))
for idx, ax in enumerate(axes):
    ax.imshow(images[idx].numpy().squeeze(), cmap='gray')
    ax.axis('off')
plt.show()
```



```
# Defining the Discriminator
class Discriminator(nn.Module):
    def __init__(self):
        super(Discriminator, self).__init__()
        self.main = nn.Sequential(
            nn.Conv2d(1, 64, 5, 2, 2), # (channels, output_channels, kernel_size, stride, padding)
            nn.LeakyReLU(0.3),
            nn.Dropout(0.3),
            nn.Conv2d(64, 128, 5, 2, 2),
            nn.LeakyReLU(0.3),
            nn.Dropout(0.3),
            nn.Flatten(),
            nn.Linear(128*7*7, 1)
        )

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

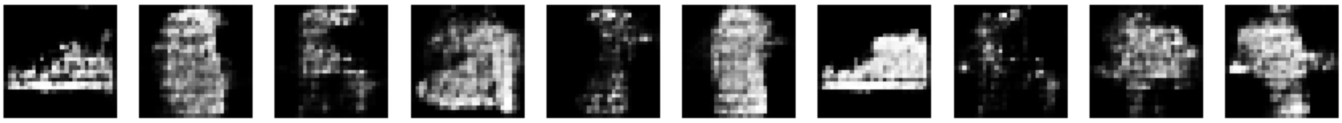
```
# Defining the Generator
class Generator(nn.Module):
    def __init__(self):
        super(Generator, self).__init__()
        self.main = nn.Sequential(
            nn.Linear(100, 256*7*7, bias=False),
            nn.BatchNorm1d(256*7*7),
            nn.LeakyReLU(0.3),
            nn.Unflatten(1, (256, 7, 7)),
            nn.ConvTranspose2d(256, 128, 5, 1, 2, bias=False),
            nn.BatchNorm2d(128),
            nn.LeakyReLU(0.3),
            nn.ConvTranspose2d(128, 64, 5, 2, 2, output_padding=1, bias=False),
            nn.BatchNorm2d(64),
            nn.LeakyReLU(0.3),
            nn.ConvTranspose2d(64, 1, 5, 2, 2, output_padding=1),
            nn.Tanh()
        )

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

```
# Initializing the Models
D = Discriminator()
G = Generator()

criterion = nn.BCEWithLogitsLoss()
d_optimizer = optim.Adam(D.parameters(), lr=0.0001)
g_optimizer = optim.Adam(G.parameters(), lr=0.0001)

# Create lists to track losses
d_losses, g_losses = [], []
```

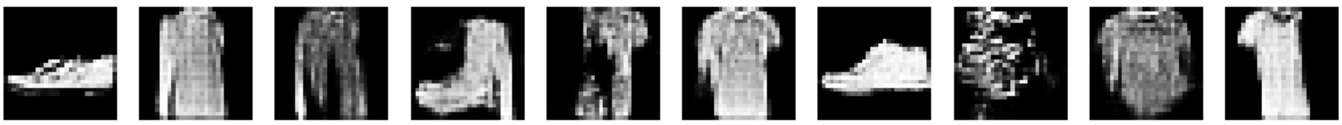
```
# Training the DCGAN
num_epochs = 50
fixed_noise = torch.randn(64, 100)

def train_discriminator(real_data, fake_data):
    batch_size = real_data.size(0)

    # Train on Real Data
    real_labels = torch.ones(batch_size, 1)
    output = D(real_data)
    d_loss_real = criterion(output, real_labels)
    real_score = torch.sigmoid(output)

    # Train on Fake Data
    fake_labels = torch.zeros(batch_size, 1)
    output = D(fake_data)
    d_loss_fake = criterion(output, fake_labels)
    fake_score = torch.sigmoid(output)

    # Backprop and optimize
    d_loss = d_loss_real + d_loss_fake
    D.zero_grad()
    d_loss.backward()
    d_optimizer.step()

    return d_loss, real_score, fake_score


def train_generator(fake_data):
    batch_size = fake_data.size(0)

    # Generate fake labels that seem real
    real_labels = torch.ones(batch_size, 1)
    output = D(fake_data)
    g_loss = criterion(output, real_labels)

    # Backprop and optimize
    G.zero_grad()
    g_loss.backward()
    g_optimizer.step()

    return g_loss

for epoch in range(num_epochs):
    d_epoch_losses = []
    g_epoch_losses = []

    for real_images, _ in trainloader:
        # Generate fake images
        noise = torch.randn(real_images.size(0), 100)
        fake_images = G(noise)

        # Train the discriminator
        d_loss, real_score, fake_score = train_discriminator(real_images, fake_images)
        d_epoch_losses.append(d_loss.item())

        # Train the generator
        fake_images = G(torch.randn(real_images.size(0), 100))
```

```python
        fake_images = G(torch.randn(real_images.size(0), 100))
        g_loss = train_generator(fake_images)
        g_epoch_losses.append(g_loss.item())

    d_losses.append(sum(d_epoch_losses) / len(d_epoch_losses))
    g_losses.append(sum(g_epoch_losses) / len(g_epoch_losses))

    print(f'Epoch [{epoch+1}/{num_epochs}] | '
          f'D Loss: {d_losses[-1]:.4f} | G Loss: {g_losses[-1]:.4f} | '
          f'Real Score: {real_score.mean().item():.4f} | Fake Score: {fake_score.mean().item():.4f}')

    # Display intermediate results
    if epoch in [9, 29, 49]:
        with torch.no_grad():
            fake_images = G(fixed_noise).detach().numpy()

        fig, axes = plt.subplots(1, 10, figsize=(15, 5))
        for i, ax in enumerate(axes):
            ax.imshow(np.squeeze(fake_images[i]), cmap='gray')
            ax.axis('off')
        plt.show()
```

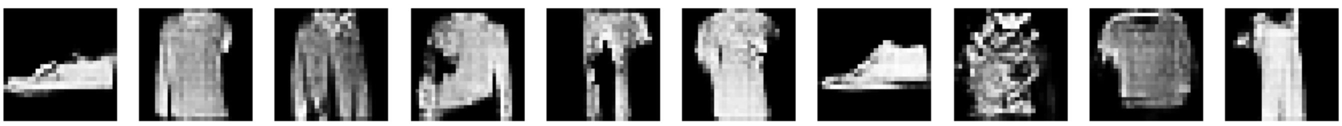
```
Epoch [1/50] | D Loss: 0.5943 | G Loss: 2.5842 | Real Score: 0.7112 | Fake Score: 0.1822
Epoch [2/50] | D Loss: 0.8685 | G Loss: 1.8305 | Real Score: 0.6873 | Fake Score: 0.3163
Epoch [3/50] | D Loss: 0.9258 | G Loss: 1.5726 | Real Score: 0.6891 | Fake Score: 0.2523
Epoch [4/50] | D Loss: 0.9405 | G Loss: 1.5619 | Real Score: 0.6299 | Fake Score: 0.2901
Epoch [5/50] | D Loss: 0.9037 | G Loss: 1.6281 | Real Score: 0.7620 | Fake Score: 0.2319
Epoch [6/50] | D Loss: 0.8329 | G Loss: 1.7278 | Real Score: 0.7720 | Fake Score: 0.2367
Epoch [7/50] | D Loss: 0.7881 | G Loss: 1.8210 | Real Score: 0.7213 | Fake Score: 0.4369
Epoch [8/50] | D Loss: 0.7995 | G Loss: 1.8218 | Real Score: 0.6632 | Fake Score: 0.2779
Epoch [9/50] | D Loss: 0.7865 | G Loss: 1.8399 | Real Score: 0.7605 | Fake Score: 0.2394
Epoch [10/50] | D Loss: 0.8049 | G Loss: 1.7911 | Real Score: 0.7907 | Fake Score: 0.3515
```



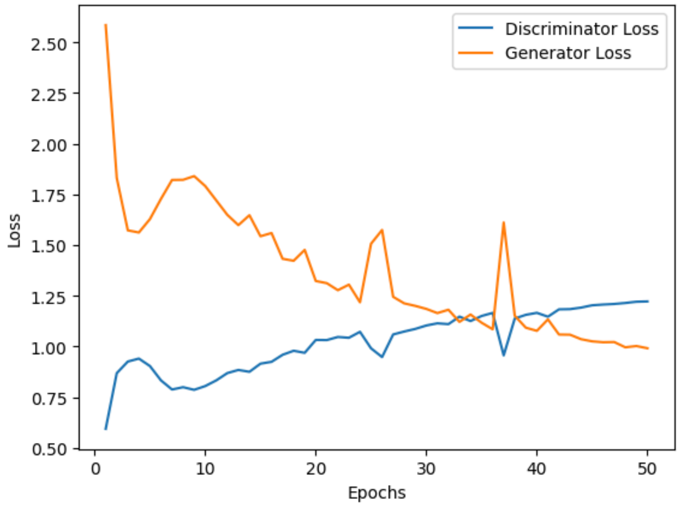
```
Epoch [11/50] | D Loss: 0.8330 | G Loss: 1.7202 | Real Score: 0.6863 | Fake Score: 0.3077
Epoch [12/50] | D Loss: 0.8689 | G Loss: 1.6492 | Real Score: 0.7576 | Fake Score: 0.2818
Epoch [13/50] | D Loss: 0.8845 | G Loss: 1.5983 | Real Score: 0.6865 | Fake Score: 0.2593
Epoch [14/50] | D Loss: 0.8753 | G Loss: 1.6472 | Real Score: 0.7071 | Fake Score: 0.3020
Epoch [15/50] | D Loss: 0.9154 | G Loss: 1.5435 | Real Score: 0.7350 | Fake Score: 0.3702
Epoch [16/50] | D Loss: 0.9245 | G Loss: 1.5594 | Real Score: 0.7295 | Fake Score: 0.3052
Epoch [17/50] | D Loss: 0.9592 | G Loss: 1.4324 | Real Score: 0.6558 | Fake Score: 0.3149
Epoch [18/50] | D Loss: 0.9790 | G Loss: 1.4227 | Real Score: 0.6530 | Fake Score: 0.2719
Epoch [19/50] | D Loss: 0.9690 | G Loss: 1.4762 | Real Score: 0.6929 | Fake Score: 0.2598
Epoch [20/50] | D Loss: 1.0324 | G Loss: 1.3231 | Real Score: 0.6073 | Fake Score: 0.3401
Epoch [21/50] | D Loss: 1.0317 | G Loss: 1.3122 | Real Score: 0.5466 | Fake Score: 0.3074
Epoch [22/50] | D Loss: 1.0470 | G Loss: 1.2776 | Real Score: 0.7715 | Fake Score: 0.3679
Epoch [23/50] | D Loss: 1.0431 | G Loss: 1.3050 | Real Score: 0.7093 | Fake Score: 0.3963
Epoch [24/50] | D Loss: 1.0725 | G Loss: 1.2180 | Real Score: 0.6623 | Fake Score: 0.3729
Epoch [25/50] | D Loss: 0.9910 | G Loss: 1.5066 | Real Score: 0.7124 | Fake Score: 0.3833
Epoch [26/50] | D Loss: 0.9481 | G Loss: 1.5744 | Real Score: 0.7057 | Fake Score: 0.4231
Epoch [27/50] | D Loss: 1.0594 | G Loss: 1.2450 | Real Score: 0.6058 | Fake Score: 0.2504
Epoch [28/50] | D Loss: 1.0738 | G Loss: 1.2127 | Real Score: 0.6101 | Fake Score: 0.3008
Epoch [29/50] | D Loss: 1.0867 | G Loss: 1.2003 | Real Score: 0.6326 | Fake Score: 0.3819
Epoch [30/50] | D Loss: 1.1036 | G Loss: 1.1854 | Real Score: 0.5511 | Fake Score: 0.4214
```



```
Epoch [31/50] | D Loss: 1.1144 | G Loss: 1.1647 | Real Score: 0.7016 | Fake Score: 0.3759
Epoch [32/50] | D Loss: 1.1102 | G Loss: 1.1813 | Real Score: 0.6091 | Fake Score: 0.4052
Epoch [33/50] | D Loss: 1.1474 | G Loss: 1.1214 | Real Score: 0.5852 | Fake Score: 0.3863
Epoch [34/50] | D Loss: 1.1255 | G Loss: 1.1576 | Real Score: 0.5542 | Fake Score: 0.4581
Epoch [35/50] | D Loss: 1.1504 | G Loss: 1.1174 | Real Score: 0.6410 | Fake Score: 0.3510
Epoch [36/50] | D Loss: 1.1657 | G Loss: 1.0850 | Real Score: 0.5515 | Fake Score: 0.3022
Epoch [37/50] | D Loss: 0.9563 | G Loss: 1.6114 | Real Score: 0.5887 | Fake Score: 0.2779
Epoch [38/50] | D Loss: 1.1379 | G Loss: 1.1516 | Real Score: 0.6336 | Fake Score: 0.4545
Epoch [39/50] | D Loss: 1.1562 | G Loss: 1.0928 | Real Score: 0.6965 | Fake Score: 0.3781
Epoch [40/50] | D Loss: 1.1661 | G Loss: 1.0773 | Real Score: 0.5446 | Fake Score: 0.3351
Epoch [41/50] | D Loss: 1.1469 | G Loss: 1.1338 | Real Score: 0.5883 | Fake Score: 0.3400
Epoch [42/50] | D Loss: 1.1832 | G Loss: 1.0591 | Real Score: 0.6780 | Fake Score: 0.3342
Epoch [43/50] | D Loss: 1.1842 | G Loss: 1.0583 | Real Score: 0.6211 | Fake Score: 0.4239
Epoch [44/50] | D Loss: 1.1918 | G Loss: 1.0360 | Real Score: 0.5920 | Fake Score: 0.4134
Epoch [45/50] | D Loss: 1.2032 | G Loss: 1.0255 | Real Score: 0.5755 | Fake Score: 0.3886
Epoch [46/50] | D Loss: 1.2069 | G Loss: 1.0210 | Real Score: 0.6318 | Fake Score: 0.3845
Epoch [47/50] | D Loss: 1.2097 | G Loss: 1.0219 | Real Score: 0.5693 | Fake Score: 0.3967
Epoch [48/50] | D Loss: 1.2145 | G Loss: 0.9961 | Real Score: 0.6107 | Fake Score: 0.4522
Epoch [49/50] | D Loss: 1.2208 | G Loss: 1.0024 | Real Score: 0.5956 | Fake Score: 0.4221
Epoch [50/50] | D Loss: 1.2223 | G Loss: 0.9914 | Real Score: 0.5815 | Fake Score: 0.4741
```



```
# Plotting Loss Curves
plt.figure()
plt.plot(range(1, num_epochs+1), d_losses, label='Discriminator Loss')
plt.plot(range(1, num_epochs+1), g_losses, label='Generator Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()
```




```

```