Homework Assignment - 6

Name: Sagar Patel

NETID: sp5894

Question 1

In this problem we will see how minimax optimization (such as the one used for training GANs) is somewhat different from. For illustration, consider a simple problem, consider a simple function of 2 scalars:

$$min_x max_y \quad f(x, y) = 4x^2 - 4y^2$$

You can try graphing the function in Python to visualize this (there is no need to include the graph in your answer).

1.a

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

Solution:

Saddle point for $min_x max_y \quad f(x, y) = 4x^2 - 4y^2$ can be calculated using gradients.

$f_x(x, y) = 8x, \qquad f_y(x, y) = -8y$, and

$f_{xx}(x, y) = 8, \qquad f_{yy}(x, y) = -8,$ and, $f_{xy}(x, y) = 0$

Therefore, $f_x(x, y) = 0, \quad f_y(x, y) = 0$, we get $(0, 0)$ as a critical point.

$D = f_{xx}(x, y) f_{yy}(x, y) - f_{xy}(x, y) = -64 < 0$

Therefore, the Saddle Point is $(0, 0)$

1.b

Write down the gradient descent/ascent equations for this objective function starting from any arbitrary initial point.

Solution:

We know that the gradients can be found out by -

$$x_{i+1} = x_i - \eta f_x(x, y), \qquad y_{i+1} = y_i - \eta f_y(x, y)$$

Take the starting point s $(1, 1)$, we then get -

$$x_{i+1} = 1 - \eta 8, \qquad y_{i+1} = 1 - \eta 8$$

1.c

Determine the range of allowable step sizes for the ascent and descent required to ensure that the gradient descent/ascent converges to the saddle point.

Solution:

We know that for the convergence, the learning rate should either be 0 or 1.

$$0 < ||1 - \eta a|| < 1$$

When $a = 8$, we get $0 < \eta < \frac{1}{4}$

1.d

What if, instead of solving a minimax problem, you just did regular gradient descent for both x and y? Comment on the dynamics of the updates, and whether or not one would converge to the saddle point.

Solution:

$$x_{i+1} = x_i - \eta 8x, \qquad y_{i+1} = y_i + \eta 8y$$

As we can see that, it will **NEVER** converge to the saddle point $(0, 0)$.

Question 2

In this problem, the goal is to train and visualize the outputs of a simple Deep Convolutional GAN (DCGAN) to generate realistic-looking (but fake) images of clothing.

In [ ]:
```
import tensorflow as tf
import matplotlib.pyplot as plt
import numpy as np
from tensorflow.keras import layers
```

2.a

Use the FashionMNIST training dataset to train the DCGAN. APIs for downloading it are available in both PyTorch and TensorFlow. Images are grayscale and size 28 × 28.

In [ ]:
```
# Load the data

(x_train,y_train), (x_test, y_test) = tf.keras.datasets.fashion_mnist.load_
```

In [ ]:
```
# Normalizing the images

x_train = (x_train.astype(np.float32) - 127.5)/127.5
x_test = (x_test.astype(np.float32) - 127.5)/127.5
```

In [ ]:
```
# Batch the data

BATCH_SIZE = 256
x_train_batch = tf.data.Dataset.from_tensor_slices(x_train).batch(BATCH_SIZ
```

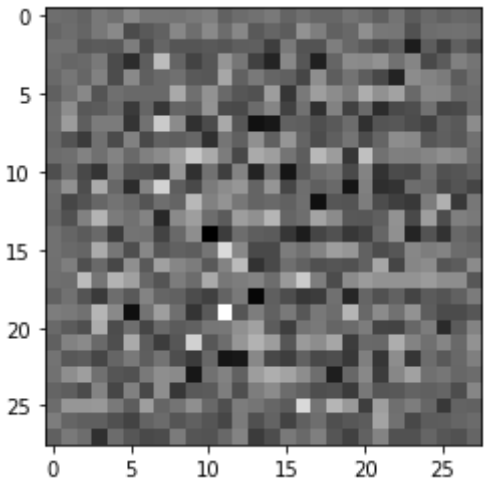
2.c

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 maps it to a vector of size $(7 \times 7 \times 256)$. No bias terms.
- several transpose 2D convolutions $(256 \times 7 \times 7 \rightarrow 128 \times 7 \times 7 \rightarrow 64 \times 14 \times 14 \rightarrow 1 \times 28 \times 28)$. 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).

In [ ]:
```
# Architecture for Generator

def build_generator():
    model = tf.keras.Sequential()

    model.add(layers.Dense(7*7*256, use_bias = False, input_shape=(z_dim,))
    model.add(layers.BatchNormalization())
    model.add(layers.LeakyReLU(alpha = 0.3))

    model.add(layers.Reshape((7, 7, 256)))

    model.add(layers.Conv2DTranspose(128, kernel_size = 5, strides = 1, pad
    model.add(layers.BatchNormalization())
    model.add(layers.LeakyReLU(alpha = 0.3))

    model.add(layers.Conv2DTranspose(64, kernel_size = 5, strides = 2, padd
    model.add(layers.BatchNormalization())
    model.add(layers.LeakyReLU(alpha = 0.3))

    model.add(layers.Conv2DTranspose(1, kernel_size = 5, strides = 2, paddi

    return model

generator = build_generator()

# generate random noise image

noise = tf.random.normal([1, 100])
generated_img = generator(noise, training=False)

plt.imshow(generated_img[0,:,:,0], cmap='gray')
```

Out[69]:
```
<matplotlib.image.AxesImage at 0x7ff27a670e80>
```



2.b

Use the following discriminator architecture (kernel size = 5 × 5 with stride = 2 in both directions):

- 2D convolutions(1×28×28→64×14×14→128×7×7)

- 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.

Here is a link that discusses how to appropriately choose padding and stride values in order to desired sizes.

In [ ]:
```
# Architecture for Discriminator Model

def build_discriminator():
    model = tf.keras.Sequential()

    model.add(layers.Conv2D(64, kernel_size = 5, strides = 2, padding='same
    model.add(layers.LeakyReLU())
    model.add(layers.Dropout(0.3))

    model.add(layers.Conv2D(128, kernel_size = 5, strides = 2, padding='sam
    model.add(layers.LeakyReLU())
    model.add(layers.Dropout(0.3))

    model.add(layers.Flatten())
    model.add(layers.Dense(1))

    return model

discriminator = build_discriminator()
decision_output = discriminator(generated_img)
print(decision_output)
```

```
tf.Tensor([[-0.00162477]], shape=(1, 1), dtype=float32)
```

2.d

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

In [ ]:
```
# Compute CrossEntropyLoss

cross_entropy = tf.keras.losses.BinaryCrossentropy(from_logits = True)

# Define Generator loss
def generator_loss(fake):
    return cross_entropy(tf.ones_like(fake), fake)

# Define discriminator loss
def discriminator_loss(real, fake):
    d_loss_real = cross_entropy(tf.ones_like(real), real)
    d_loss_fake = cross_entropy(tf.zeros_like(fake), fake)
    d_loss = d_loss_real + d_loss_fake
    return d_loss
```

In [ ]:
```
# Optimizers for generator and discriminator, learning rate = 10^(-4)

g_optimizer = tf.keras.optimizers.Adam(1e-4)
d_optimizer = tf.keras.optimizers.Adam(1e-4)
```

In [ ]:
```
EPOCHS = 50
z_dim = 100
images_generated = 9
seed = tf.random.normal([images_generated, z_dim])
```

2.e

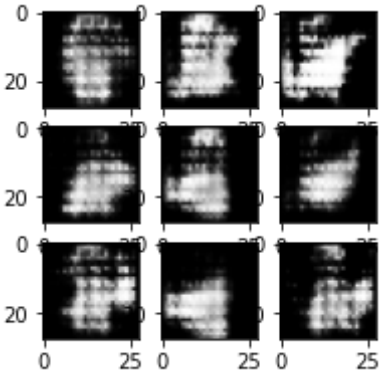
Train it for 50 epochs. You can use minibatch sizes of 16, 32, or 64. Training may take several minutes (or even up to an hour), so be patient! Display intermediate images generated after T = 10, T = 30, and T = 50 epochs. If the random seeds are fixed throughout then you should get results of the following quality:

In [ ]:
```
# Train function to update generator and discriminator

def train(x_train_batch):

    generator_losses_history = []
    discriminator_losses_history = []

    for epoch in range(EPOCHS):

        epoch_g_loss = 0
        epoch_d_loss = 0

        for img in x_train_batch:

            noise = tf.random.normal([BATCH_SIZE, z_dim])

            with tf.GradientTape() as g, tf.GradientTape() as d:

                generated_img = generator(noise, training=True)

                real = discriminator(img, training=True)
                fake = discriminator(generated_img, training=True)

                g_loss = generator_loss(fake)
                d_loss = discriminator_loss(real, fake)

                epoch_g_loss += float(g_loss)
                epoch_d_loss += float(d_loss)

            grad_generator = g.gradient(g_loss, generator.trainable_variabl
            grad_discriminator = d.gradient(d_loss, discriminator.trainable

            g_optimizer.apply_gradients(zip(grad_generator, generator.train
            d_optimizer.apply_gradients(zip(grad_discriminator, discriminat

        if (epoch == 9 or epoch == 29 or epoch == 49):
            print ('Epoch: {}'.format(epoch+1))
            print_images(generator, seed)

        generator_losses_history.append(epoch_g_loss / len(x_train_batch))
        discriminator_losses_history.append(epoch_d_loss / len(x_train_batc

    return (generator_losses_history, discriminator_losses_history)
```

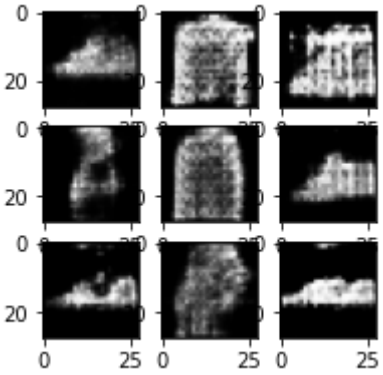
In [ ]:
```
def print_images(model, test_input):

    predictions = model(test_input, training=False)

    fig = plt.figure(figsize=(3, 3))

    for i in range(predictions.shape[0]):
        plt.subplot(3, 3, i+1)
        plt.imshow(predictions[i, :, :, 0] * 127.5 + 127.5, cmap='gray')

    plt.show()
```

In [ ]:
```
gl, dl = train(x_train_batch)
```

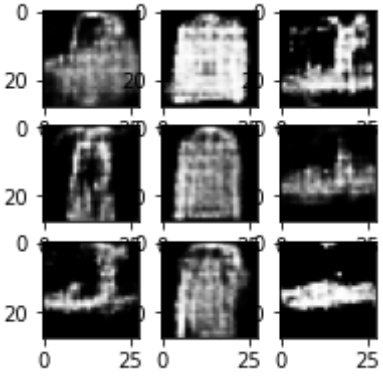
```
Epoch: 10
```



```
Epoch: 30
```



```
Epoch: 50
```

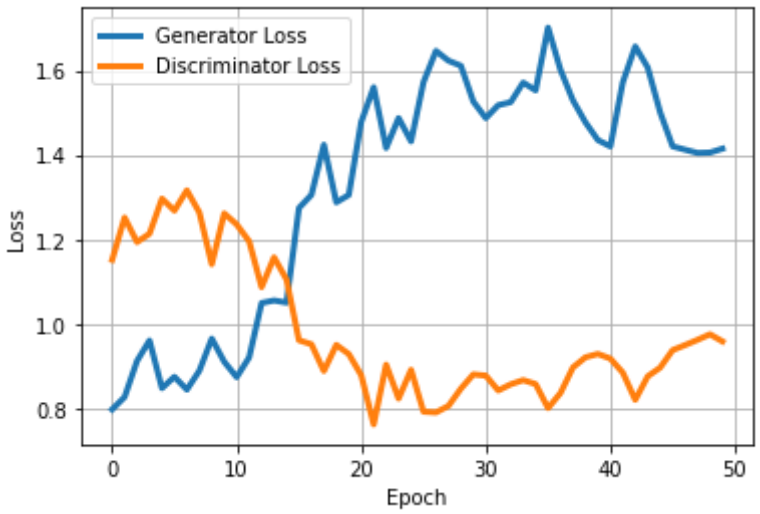


2.f

Report loss curves for both the discriminator and the generator loss over all epochs, and qualitatively comment on their behavior.

In [ ]:
```
plt.plot(range(50), gl, '-', linewidth=3, label = 'Generator Loss')
plt.plot(range(50), dl,'-',linewidth=3, label = 'Discriminator Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.grid(True)
plt.legend()
```

Out[77]: <matplotlib.legend.Legend at 0x7ff1f7e94940>



Comment on their behaviour

- The generator and discriminator are competing against each other hence while the discriminator learns how to identify fake images correctly, the loss decreases and this causes an increase in the loss of generator.
- If we train for up to 250 or 500 epoches, theoretically we should see the losses stabilizing and reaching an equilibrium.

In [ ]: