ECE GY 7123 Deep Learning Assignment 3

Shubham Singh – sks9437@nyu.edu

---

1 Q1.

1. **(5 points)** *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 $\pi(a_i) = \text{softmax}(\theta_i)$ 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:

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

   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?

Answer:

Let us define the function π such that $\pi(a_i)$ is equal to the softmax of u_i, for $i = 1, \ldots, k$, where k represents the total number of actions, and u_i denotes a scalar parameter associated with each action. We can demonstrate that if action a_i is chosen, the adjustment in the parameters is given by the following equation:

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

The softmax function is a fundamental concept in machine learning and is defined as:

$$\pi(a_i) = \frac{e^{u_i}}{\sum_{j=1}^{k} e^{u_j}}. \tag{2}$$

This function takes a vector of real numbers as input and transforms them into a vector of values between 0 and 1, which can be interpreted as probabilities. The softmax function is

particularly useful in scenarios where we need to model categorical distributions, such as in multinomial logistic regression or reinforcement learning problems.

In the context of reinforcement learning, our objective is to maximize the expected reward, which can be expressed as:

$$E[R] = \sum_{i=1}^{k} \pi(a_i) R_i. \tag{3}$$

This equation represents the expected reward as the sum of the probabilities of each action multiplied by the respective rewards associated with those actions.

To optimize the parameters of our model, we can leverage the policy gradient theorem, which provides a way to compute the gradient of the expected reward with respect to the parameters. The gradient of $E[R]$ with respect to the parameter u_i is given by:

$$\frac{\partial E[R]}{\partial u_i} = \sum_{j=1}^{k} \frac{\partial \pi(a_j)}{\partial u_i} R_j. \tag{4}$$

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

$$\frac{\partial \pi(a_j)}{\partial u_i} = \pi(a_j)[\delta_{ij} - \pi(a_i)], \tag{5}$$

where δ_{ij} is the Kronecker delta, which is a function that takes the value 1 when $i = j$ and 0 otherwise. This leads us to the following expression for the gradient:

$$\frac{\partial E[R]}{\partial u_i} = \pi(a_i)[1 - \pi(a_i)] R_i. \tag{6}$$

Finally, applying the gradient ascent update rule with a step size η, we obtain the desired update equation:

$$\Delta u_i = \eta \frac{\partial E[R]}{\partial u_i} = \eta R_i [1 - \pi(a_i)]. \tag{7}$$

This equation provides a way to update the parameters u_i in the direction that maximizes the expected reward, taking into account the probability of selecting action a_i and the associated reward R_i.

(b): While the above update rule is effective, it can lead to unstable training if constant step sizes are used. This instability can arise due to two main reasons:

1. Large Updates: If the reward R_i and the step size η are both large, the parameter updates Δu_i can become significant, potentially causing oscillations or divergence of the parameters u_i. This can lead to unstable behavior and prevent convergence to optimal solutions.

2. Sensitivity to $\pi(a_i)$: The term $1 - \pi(a_i)$ in the update equation varies depending on the probability distribution. If $\pi(a_i)$ is close to 1, the update becomes negligible, leading to slow convergence or stagnation. Conversely, if $\pi(a_i)$ is close to 0, the update becomes substantial, potentially causing overshooting and instability.

To address these issues and ensure convergence of the parameters, several strategies can be employed:

1. Adaptive Learning Rates: Instead of using a constant step size, adaptive optimization algorithms like Adam or RMSprop can be utilized. These algorithms adjust the learning rate dynamically based on the magnitude of the gradient, which can help stabilize training and improve convergence.

2. Decaying Learning Rate: Another approach is to gradually decrease the step size η over time. This technique, known as learning rate decay, can help prevent large oscillations in the early stages of training and allow for finer adjustments as the training progresses, ultimately leading to better convergence.

3. Gradient Clipping: To mitigate the impact of large gradients, gradient clipping techniques can be employed. Gradient clipping limits the magnitude of the gradients to a predefined threshold, preventing excessive updates and improving stability during training.

4. Batch Normalization: Batch normalization is a widely used technique in deep learning that can help stabilize training by normalizing the input distributions across batches. By standardizing the inputs, batch normalization can reduce the internal covariate shift, leading to faster convergence and improved generalization.

By incorporating one or more of these techniques, the instability issues arising from constant step sizes can be mitigated, and the training process can be made more robust and stable, ultimately leading to better convergence and improved performance of the reinforcement learning model.

2 Q2.

2. **(5 points)** *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:

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

You can try graphing this function in Python if you like (no need to include it in your answer.

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.

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

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

d. (2 points). 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.

Answer:
(a) We begin by computing the partial derivatives of $f(x, y)$ with respect to x and y, which are given by $\frac{\partial f}{\partial x} = 8x$ and $\frac{\partial f}{\partial y} = -8y$, respectively. Setting both derivatives to zero, we find the stationary point at $(0, 0)$.

To determine the nature of this point, we evaluate the second derivatives:
$\frac{\partial^2 f}{\partial x^2} = 8$ and $\frac{\partial^2 f}{\partial y^2} = -8$.

The positive second derivative with respect to x and the negative second derivative with respect to y indicate that the point $(0, 0)$ is a saddle point.

(b) The iterative update equations for gradient descent/ascent, starting from an arbitrary initialization (x_0, y_0), are given by:

$$x_{k+1} = x_k - \alpha \cdot 8x_k,$$
$$y_{k+1} = y_k + \beta \cdot 8y_k,$$

where α and β represent the step sizes for x and y respectively.

(c) To ensure convergence of gradient descent/ascent, the step sizes α and β must satisfy certain conditions. For x, the convergence criterion for gradient descent is $|1 - \alpha \cdot 8| < 1$, which simplifies to $0 < \alpha < \frac{1}{4}$. Similarly, for y, the convergence criterion for gradient ascent is $|1 + \beta \cdot 8| < 1$, which yields $-\frac{1}{4} < \beta < 0$.

(d) Regular gradient descent applied to both variables would lead to updates converging to $(0, 0)$ if the step size α falls within the range $0 < \alpha < \frac{1}{4}$. However, it's important to note that gradient descent is generally not well-suited for minimax problems, as it seeks to minimize both variables. In a minimax scenario, we aim to minimize one variable while maximizing the other. While the updates might converge to the saddle point $(0, 0)$ in this specific case, it's not a reliable approach for minimax optimization problems.

3 Q3

3. **(5 points)** *Generative models*. In this problem, the goal is to train and visualize the outputs of a simple Deep Convolutional GAN (DCGAN) to generate realistic-looking (but synthetic) images of clothing items.

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

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

      - 2D convolutions ($1 \times 28 \times 28 \rightarrow 64 \times 14 \times 14 \rightarrow 128 \times 7 \times 7$)
      - 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.

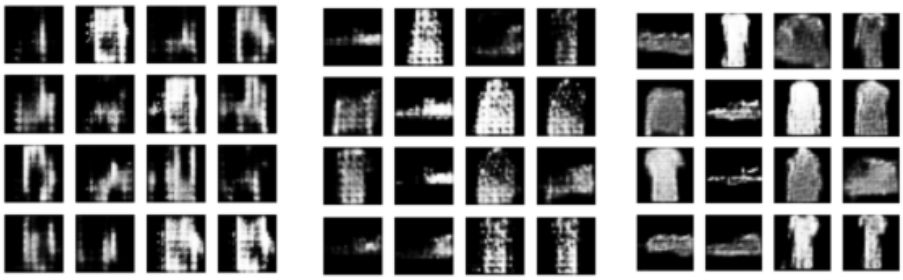
   c. Use the following generator architecture (which is essentially the reverse of a standard discriminative architecture). You can use the same kernel size. Construct:

      - a dense layer that takes a unit Gaussian noise vector of length 100 and maps it to a vector of size $7 * 7 * 256$. 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).

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

   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:



Answer:

Understanding DCGAN

DCGAN, short for Deep Convolutional Generative Adversarial Network, is a class of deep learning architectures designed for generating realistic images from random noise. It is a type of Generative Adversarial Network (GAN) that employs convolutional neural networks (CNNs) to generate high-quality images. DCGANs consist of two neural networks: a generator and a discriminator, which are trained simultaneously in an adversarial manner. The generator takes random noise as input and transforms it into synthetic images, while the discriminator tries to distinguish between real and fake images. Through adversarial training, the generator learns to generate images that are indistinguishable from real images, while the discriminator learns to become more accurate at distinguishing real from fake images.

The architecture of a DCGAN typically involves convolutional layers in both the generator and discriminator networks. In the generator, the convolutional layers progressively upsample the noise vector to generate increasingly complex features, ultimately producing synthetic images. Batch normalization is often applied to stabilize training, and activation functions like Leaky ReLU are used to prevent the generator from dying during training. In contrast, the discriminator consists of convolutional layers followed by activation functions like Leaky ReLU and Dropout layers to prevent overfitting. The discriminator's role is to classify whether an input image is real or generated by the generator.

During training, the generator and discriminator networks are trained iteratively. The generator generates synthetic images and tries to fool the discriminator, while the discriminator learns to become more accurate at distinguishing real from fake images. This process continues until a certain convergence criterion is met, such as reaching a predefined number of epochs or achieving satisfactory performance on a validation set.

For the specific task of generating images of clothing items using the FashionMNIST dataset, a DCGAN can be trained using the provided images. The discriminator architecture typically involves 2D convolutional layers with Leaky ReLU activations and Dropout layers, while the generator architecture is essentially the reverse of the discriminator. The binary cross-entropy

loss function is commonly used for training both the generator and discriminator, with the Adam optimizer and a learning rate of 10^{-4}. Training the DCGAN for around 50 epochs with minibatch sizes of 16, 32, or 64, while displaying intermediate images generated after certain epochs, can lead to the generation of realistic-looking synthetic images of clothing items.

Deep Convolutional Generative Adversarial Network (DCGAN) for Image Generation on FashionMNIST dataset

Import libraries

In [1]:

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.models import Sequential, save_model, load_model
from tensorflow.keras.layers import Dense, Conv2DTranspose, Conv2D, Flatten
from tensorflow.keras.layers import Reshape, Dropout, BatchNormalization, LeakyReLU
import numpy as np
import matplotlib.pyplot as plt
from tqdm import tqdm
from IPython import display
%matplotlib inline
```

Prepare data

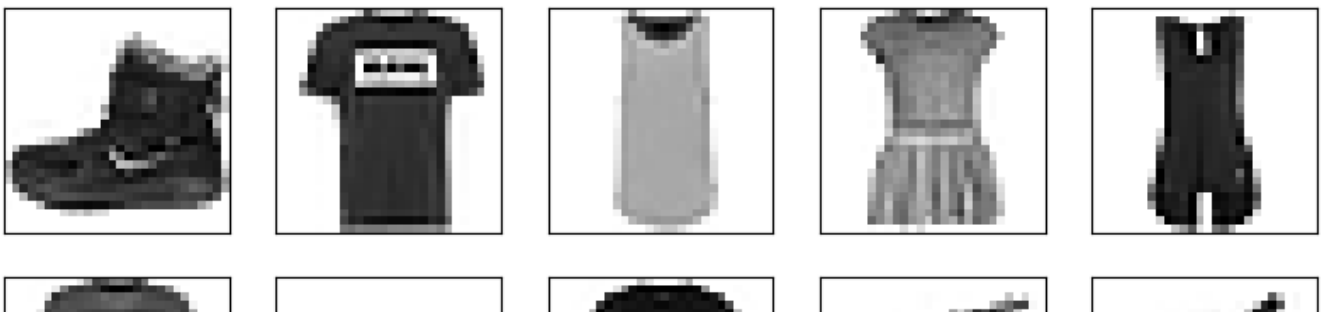
In [2]:

```
#loading fashion mnist data from keras
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.fashion_mnist.load_data()
#scaling the grayscale pixel values
x_train = x_train.astype(np.float32) / 255.0
x_test = x_test.astype(np.float32) / 255.0
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/train-l
abels-idx1-ubyte.gz
29515/29515 [==============================] - 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/train-i
mages-idx3-ubyte.gz
26421880/26421880 [==============================] - 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/t10k-la
bels-idx1-ubyte.gz
5148/5148 [==============================] - 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/t10k-im
ages-idx3-ubyte.gz
4422102/4422102 [==============================] - 0s 0us/step
```

In [3]:

```
#plotting the images in a 5x5 format
plt.figure(figsize=(10, 10))
for i in range(10):
    plt.subplot(5,5,i+1)
    plt.xticks([])
    plt.yticks([])
    plt.grid(False)
    plt.imshow(x_train[i], cmap=plt.cm.binary)
plt.show()
```



In [4]:

```
#using the tensorflow.data.Dataset API to make the training dataset
#we shuffle with buffer size 1000
batch_size = 32
dataset = tf.data.Dataset.from_tensor_slices(x_train).shuffle(1000)
dataset = dataset.batch(batch_size, drop_remainder=True).prefetch(1)
```

Generator

The generator is an essential component of a Generative Adversarial Network (GAN). Its primary function is to generate synthetic data, such as images, that closely resemble the real data present in the training set.

Layers:

1. **Input Layer (Dense):**
   - **The initial layer takes a noise vector as input, typically sampled from a normal distribution.**
   - **It has a fully connected Dense layer mapping the noise vector to a higher-dimensional space suitable for subsequent convolutional operations.**
   - **Output Shape: (7, 7, 256) - This shape is chosen based on the desired output size and the architecture of subsequent layers.**
2. **Reshape Layer:**
   - **Reshapes the output of the Dense layer into a 3D tensor, preparing it for the subsequent convolutional layers.**
   - **Output Shape: (7, 7, 256)**
3. **Transpose Convolutional Layers (Conv2DTranspose):**
   - **These layers perform the inverse operation of convolutional layers. Instead of reducing the spatial dimensions of the input, they increase them.**
   - **Each Conv2DTranspose layer is followed by Batch Normalization and Leaky ReLU activation, which helps stabilize and improve the training process.**
   - **The first Conv2DTranspose layer increases the spatial dimensions from 7x7 to 14x14, gradually upsampling the features.**
   - **The second Conv2DTranspose layer further increases the spatial dimensions from 14x14 to 28x28, matching the desired output size of the generator.**
   - **The output of the final Conv2DTranspose layer has a single channel and uses the tanh activation function to ensure that pixel values are within the range [-1, 1], suitable for image generation.**

Purpose:

The generator's objective is to learn to generate realistic images from random noise. By progressively upsampling the input noise and applying learned feature transformations, it produces synthetic images that are indistinguishable from real images to the discriminator.

In [5]:

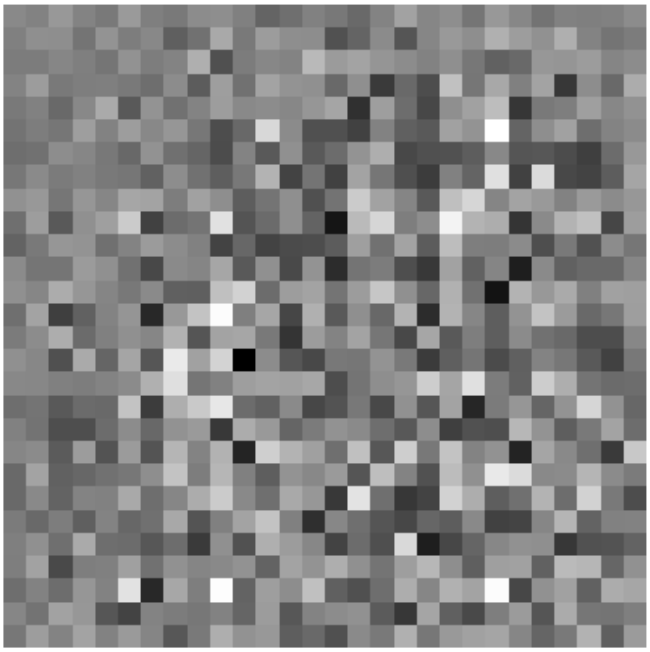
```
from tensorflow.keras.layers import Dense, Reshape, BatchNormalization, LeakyReLU, Conv2D
Transpose
from tensorflow.keras.models import Sequential

num_features = 100

# Defining the generator
generator = Sequential()
```

```
# Dense layer mapping from noise vector to 7*7*256
generator.add(Dense(7*7*256, input_shape=(num_features,), use_bias=False)) # No bias term
s
generator.add(Reshape((7, 7, 256)))

# Transpose convolutional layers
generator.add(Conv2DTranspose(128, kernel_size=(5, 5), strides=(1, 1), padding='same', u
se_bias=False)) # 7x7x256 -> 7x7x128
generator.add(BatchNormalization())
generator.add(LeakyReLU(0.3))

generator.add(Conv2DTranspose(64, kernel_size=(5, 5), strides=(2, 2), padding='same', us
e_bias=False)) # 7x7x128 -> 14x14x64
generator.add(BatchNormalization())
generator.add(LeakyReLU(0.3))

generator.add(Conv2DTranspose(1, kernel_size=(5, 5), strides=(2, 2), padding='same', act
ivation='tanh', use_bias=False)) # 14x14x64 -> 28x28x1
```

Sample Image generation from random noise

In [6]:

```
noise = tf.random.normal(shape=(1,num_features))
generated_image = generator(noise, training=False)
plt.imshow(generated_image[0, :, :, 0], cmap='gray')
plt.axis('off')
plt.show()
```



Discriminator

The discriminator is another crucial component of a Generative Adversarial Network (GAN). Its role is to discriminate between real and synthetic data by classifying input samples as either real or fake.

Layers:

1. **Convolutional Layers (Conv2D):**
    - **The discriminator starts with a series of convolutional layers responsible for extracting features from the input images.**
    - **Each Conv2D layer is followed by Leaky ReLU activation to introduce non-linearity and Dropout regularization to prevent overfitting.**
    - **The first Conv2D layer processes the input images with a kernel size of (5, 5) and a stride of (2, 2),**

reducing the spatial dimensions and increasing the depth of features to 64.
- The second Conv2D layer further processes the features, increasing the depth to 128 while maintaining the spatial dimensions.

2. **Flatten Layer:**
   - Flattens the output of the convolutional layers into a 1D tensor, preparing it for the final classification layer.
   - Output Shape: (N,) - Where N is the total number of elements in the flattened tensor.
3. **Dense Layer:**
   - The flattened tensor is fed into a Dense layer with a single neuron and sigmoid activation.
   - This Dense layer acts as a binary classifier, outputting a single probability score indicating the likelihood of the input being real (1) or fake (0).

Purpose:

The discriminator's objective is to learn to distinguish between real and synthetic images. By training on a combination of real images from the dataset and fake images generated by the generator, it learns to accurately classify input samples, providing feedback to the generator for improving its image generation process.

In [7]:

```
# Defining the discriminator
discriminator = Sequential()

# Convolutional layers
discriminator.add(Conv2D(64, (5, 5), strides=(2, 2), padding='same', input_shape=(28, 28
, 1)))
discriminator.add(LeakyReLU(0.3))
discriminator.add(Dropout(0.3))

discriminator.add(Conv2D(128, (5, 5), strides=(2, 2), padding='same'))
discriminator.add(LeakyReLU(0.3))
discriminator.add(Dropout(0.3))

# Flatten the output
discriminator.add(Flatten())

# Dense layer
discriminator.add(Dense(1, activation='sigmoid'))
```

In [8]:

```
decision = discriminator(generated_image)
print(decision)
```

```
tf.Tensor([[0.50050914]], shape=(1, 1), dtype=float32)
```

Training our model

In [9]:

```
from tensorflow.keras.optimizers import Adam

# Compile the discriminator with binary cross-entropy loss and Adam optimizer
discriminator.compile(loss='binary_crossentropy', optimizer=Adam(learning_rate=1e-4))

# Set discriminator as non-trainable before combining with the generator
discriminator.trainable = False

# Combine the generator and the discriminator
gan = Sequential()
gan.add(generator)
gan.add(discriminator)

# Compile the GAN with binary cross-entropy loss and Adam optimizer
```

```
gan.compile(loss='binary_crossentropy', optimizer=Adam(learning_rate=1e-4))
```

In [10]:

```
#random seed
seed = tf.random.normal(shape=(batch_size, num_features))
```

In [11]:

```
import tensorflow as tf
from tqdm import tqdm
from IPython import display

d_losses, g_losses = [], []

def train_dcgan(gan, dataset, batch_size, num_features, epochs=5):
    global d_losses, g_losses
    generator, discriminator = gan.layers
    for epoch in tqdm(range(epochs)):
        print("Epochs {}/{}".format(epoch+1, epochs))
        for x_batch in dataset:
            noise = tf.random.normal(shape=(batch_size, num_features))
            generated_images = generator(noise)
            x_fake_and_real = tf.concat([generated_images, x_batch], axis=0) #input for
discriminator
            y1 = tf.constant([[0.]] * batch_size + [[1.]] * batch_size) #output for disc
riminator
            discriminator.trainable = True
            d_loss = discriminator.train_on_batch(x_fake_and_real, y1)
            y2 = tf.constant([[1.]] * batch_size)
            discriminator.trainable = False
            g_loss = gan.train_on_batch(noise, y2)
            d_losses.append(d_loss)
            g_losses.append(g_loss)

        if (epoch+1) % 10 == 0 or epoch == 0 or (epoch+1) % 30 == 0 or (epoch+1) % 50 =
= 0:
            #display.clear_output(wait=True)
            generate_and_save_images(generator, epoch+1, seed)
```

In [12]:

```
def generate_and_save_images(model, epoch, test_input):
    predictions = model(test_input, training=False)
    fig = plt.figure(figsize=(10,10))

    for i in range(25):
        plt.subplot(5, 5, i+1)
        plt.imshow(predictions[i, :, :, 0] * 127.5 + 127.5, cmap='binary') #rescaling th
e pixel values back to 0-255 range
        plt.axis('off')
    plt.savefig('image_at_epoch_{:04d}.png'.format(epoch))
    plt.show()
```

In [13]:

```
x_train_dc_gan = x_train.reshape(-1, 28, 28, 1) * 2 - 1 #rescaling corectly so that ther
e is no discrepancy in any function
```

In [14]:

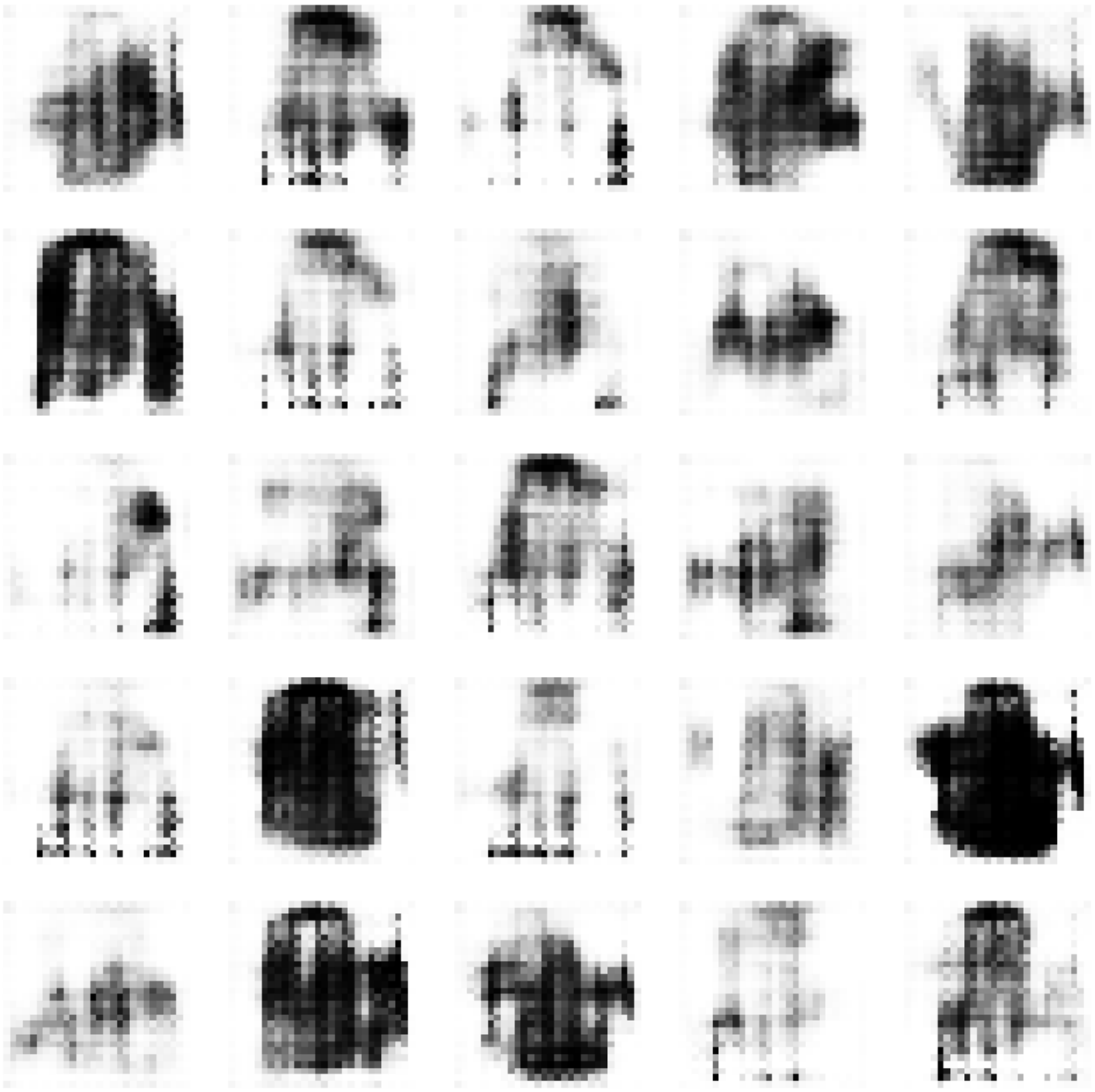
```
#batch_size = 32
dataset = tf.data.Dataset.from_tensor_slices(x_train_dc_gan).shuffle(1000)
dataset = dataset.batch(batch_size, drop_remainder = True).prefetch(1)
```

In [15]:

```
%%time
train_dcgan(gan, dataset, batch_size, num_features, epochs=50)
```

```
  0%|          | 0/50 [00:00<?, ?it/s]
```

Epochs 1/50



2%| | 1/50 [01:43<1:24:44, 103.77s/it]

Epochs 2/50

4%| | 2/50 [03:07<1:13:22, 91.72s/it]

Epochs 3/50

6%| | 3/50 [04:32<1:09:29, 88.71s/it]

Epochs 4/50

8%| | 4/50 [05:48<1:04:10, 83.70s/it]

Epochs 5/50

10%| | 5/50 [07:11<1:02:34, 83.43s/it]

Epochs 6/50

12%| | 6/50 [08:28<59:34, 81.23s/it]

Epochs 7/50

14%| | 7/50 [09:44<57:02, 79.60s/it]

Epochs 8/50

16%|▉ | 8/50 [10:59<54:44, 78.19s/it]

Epochs 9/50

18%|▉ | 9/50 [12:14<52:43, 77.16s/it]

Epochs 10/50

20%|▉ | 10/50 [13:30<51:13, 76.83s/it]

Epochs 11/50

22%|▉ | 11/50 [14:45<49:32, 76.23s/it]

Epochs 12/50

24%|▉ | 12/50 [16:02<48:26, 76.49s/it]

Epochs 13/50

26%|▉ | 13/50 [17:18<47:01, 76.26s/it]

Epochs 14/50

28%|▉ | 14/50 [18:33<45:34, 75.95s/it]

20%|██ | 14/50 [18:33<45:34, 75.95s/it]

Epochs 15/50

30%|███ | 15/50 [19:55<45:21, 77.75s/it]

Epochs 16/50

32%|███ | 16/50 [21:10<43:40, 77.06s/it]

Epochs 17/50

34%|███ | 17/50 [22:26<42:10, 76.68s/it]

Epochs 18/50

36%|███ | 18/50 [23:42<40:50, 76.58s/it]

Epochs 19/50

38%|███ | 19/50 [24:58<39:23, 76.25s/it]

Epochs 20/50

40%|████ | 20/50 [26:16<38:19, 76.65s/it]

Epochs 21/50

42%|████ | 21/50 [27:32<37:03, 76.67s/it]

42%|████ | 21/50 [27:32<37:03, 76.67s/it]

Epochs 22/50

44%|████ | 22/50 [28:48<35:38, 76.38s/it]

Epochs 23/50

46%|████ | 23/50 [30:04<34:18, 76.24s/it]

Epochs 24/50

48%|████ | 24/50 [31:19<32:55, 76.00s/it]

Epochs 25/50

50%|████ | 25/50 [32:35<31:40, 76.04s/it]

Epochs 26/50

52%|████ | 26/50 [33:52<30:25, 76.08s/it]

Epochs 27/50

54%|████ | 27/50 [35:06<29:00, 75.67s/it]

Epochs 28/50

56%|████ | 28/50 [36:22<27:45, 75.70s/it]

Epochs 29/50

58%|████ | 29/50 [37:41<26:48, 76.61s/it]

Epochs 30/50

60%|██████ | 30/50 [39:02<25:58, 77.91s/it]

Epochs 31/50

62%|██████▏ | 31/50 [40:22<24:54, 78.66s/it]

Epochs 32/50

64%|██████▍ | 32/50 [41:44<23:53, 79.64s/it]

Epochs 33/50

66%|██████▌ | 33/50 [43:03<22:32, 79.54s/it]

Epochs 34/50

68%|██████▊ | 34/50 [44:23<21:12, 79.53s/it]

Epochs 35/50

70%|███████ | 35/50 [45:43<19:56, 79.74s/it]

Epochs 36/50

72%|███████▏ | 36/50 [47:05<18:44, 80.32s/it]

Epochs 37/50

74%|███████▍ | 37/50 [48:27<17:30, 80.80s/it]

Epochs 38/50

76%|███████▌ | 38/50 [49:47<16:07, 80.66s/it]

Epochs 39/50

78%|███████▊ | 39/50 [51:07<14:45, 80.46s/it]

Epochs 40/50

80%|████████ | 40/50 [52:28<13:24, 80.47s/it]

Epochs 41/50

82%|████████ | 41/50 [53:48<12:03, 80.36s/it]

Epochs 42/50

84%|████████ | 42/50 [55:10<10:46, 80.83s/it]

Epochs 43/50

86%|████████ | 43/50 [56:30<09:24, 80.62s/it]

Epochs 44/50

88%|████████ | 44/50 [57:50<08:02, 80.38s/it]

Epochs 45/50

90%|████████ | 45/50 [59:09<06:41, 80.23s/it]

Epochs 46/50

92%|████████ | 46/50 [1:00:30<05:21, 80.36s/it]

Epochs 47/50

94%|████████ | 47/50 [1:01:52<04:02, 80.83s/it]

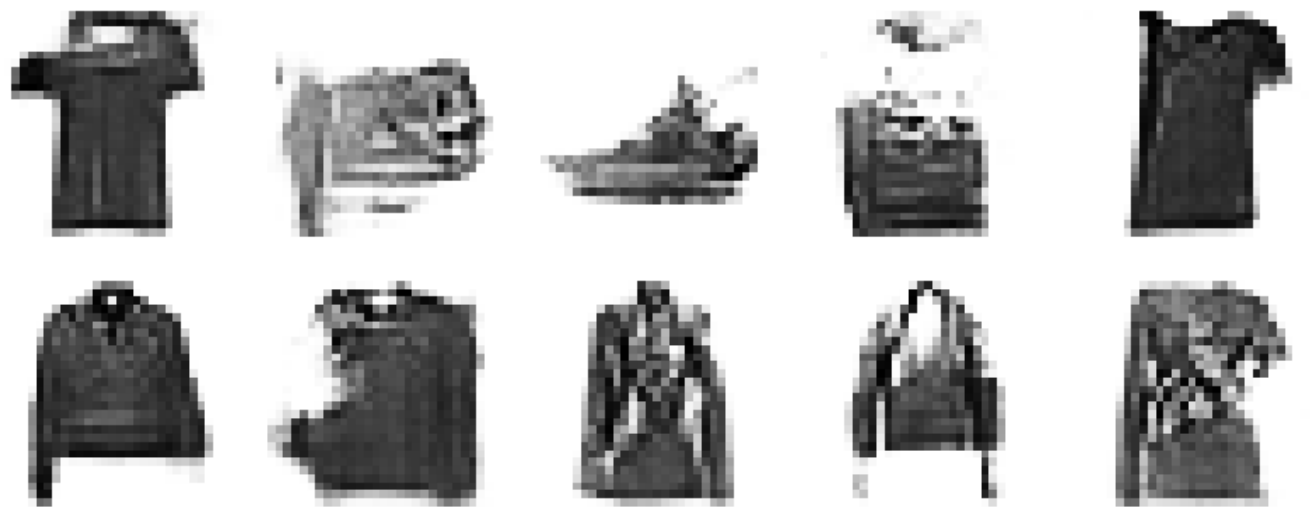
Epochs 48/50

96%|████████ | 48/50 [1:03:10<02:40, 80.03s/it]

Epochs 49/50

98%|████████ | 49/50 [1:04:30<01:19, 79.89s/it]

Epochs 50/50



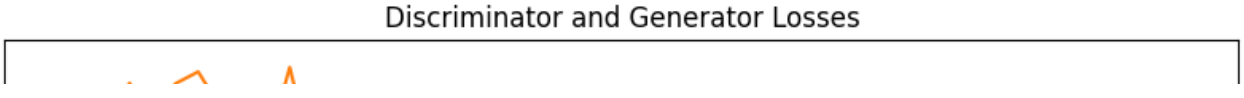
```
100%|██████████| 50/50 [1:05:53<00:00, 79.06s/it]

CPU times: user 1h 2min 46s, sys: 2min 4s, total: 1h 4min 50s
Wall time: 1h 5min 53s
```

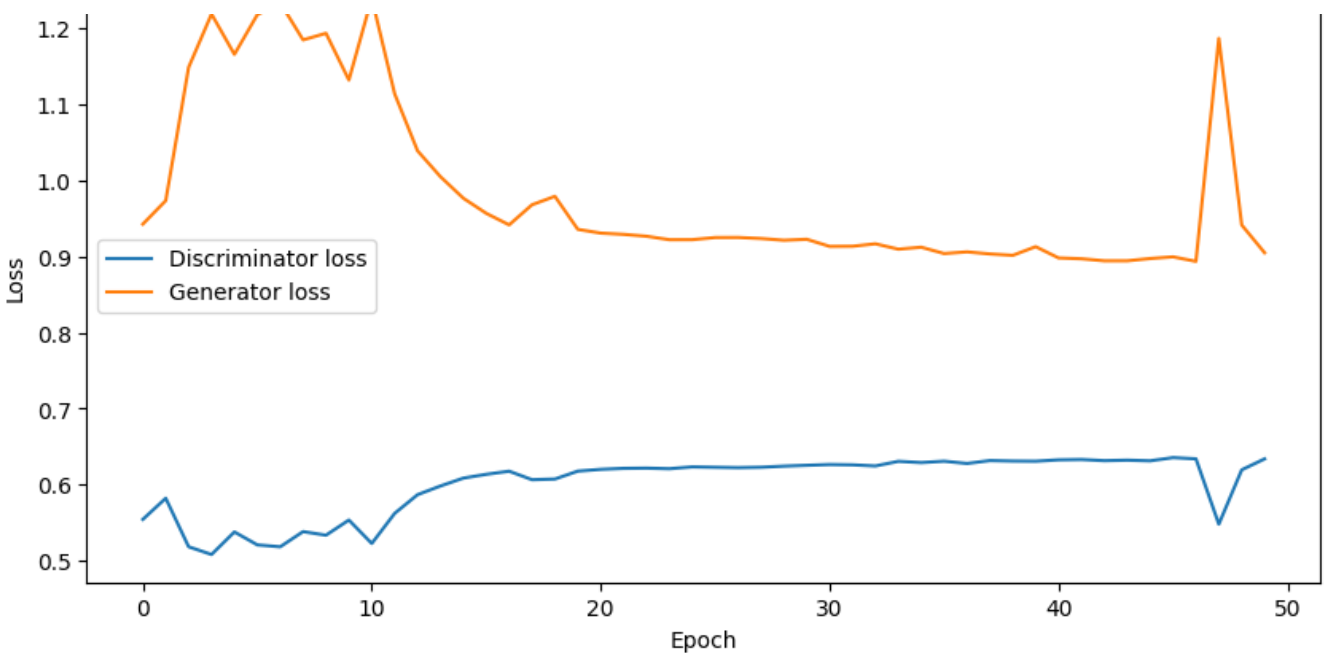
In [16]:

```
import numpy as np

# Assuming d_losses and g_losses contain the loss values
chunk_size = 1825
num_chunks = len(d_losses) // chunk_size

d_losses_avg = [np.mean(d_losses[i*chunk_size:(i+1)*chunk_size]) for i in range(num_chun
ks)]
g_losses_avg = [np.mean(g_losses[i*chunk_size:(i+1)*chunk_size]) for i in range(num_chun
ks)]

# Padding if the number of epochs is not perfectly divisible by chunk_size
if len(d_losses) % chunk_size != 0:
    d_losses_avg.append(np.mean(d_losses[num_chunks*chunk_size:]))
    g_losses_avg.append(np.mean(g_losses[num_chunks*chunk_size:]))

# Resize to length 50
d_losses_avg = np.resize(d_losses_avg, 50)
g_losses_avg = np.resize(g_losses_avg, 50)
```

In [17]:

```
# Plot loss curves
plt.figure(figsize=(10, 5))
plt.plot(d_losses_avg, label='Discriminator loss')
plt.plot(g_losses_avg, label='Generator loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('Discriminator and Generator Losses')
plt.legend()
plt.show()
```



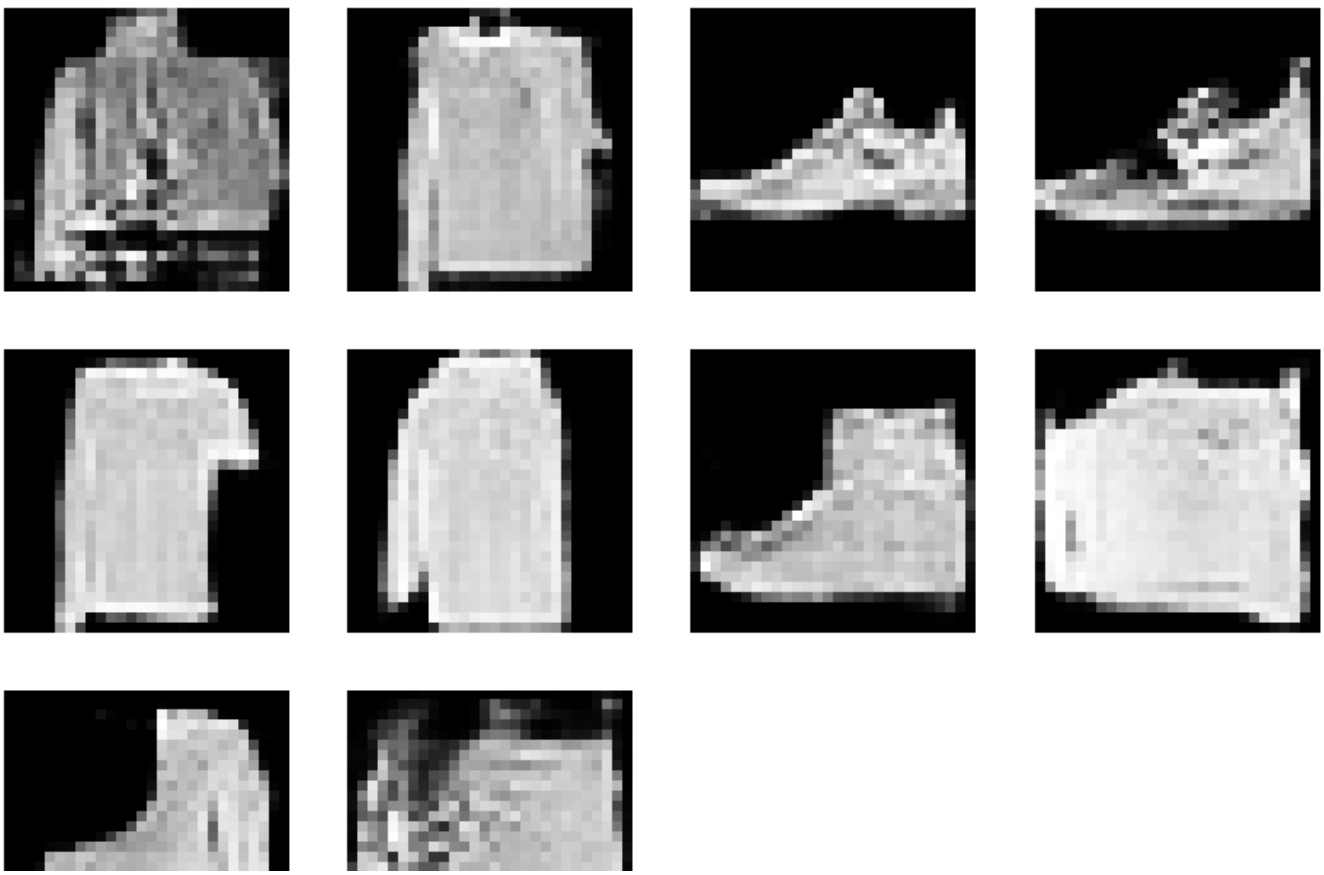


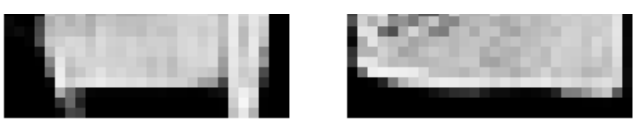
In [18]:

```
noise = tf.random.normal(shape=(batch_size, num_features))
generated_images = generator(noise)
```

In [19]:

```
import matplotlib.pyplot as plt

# Assuming `generated_images` contains the generated images
fig = plt.figure(figsize=(10, 10))
for i in range(10):  # Assuming you want to plot 10 images
    plt.subplot(4, 4, i + 1)
    plt.imshow(generated_images[i, :, :, 0] * 127.5 + 127.5, cmap='gray')
    plt.axis('off')

plt.show()
```





In [19]: