GAN Project with Keras and MNIST Dataset

1. Project Overview
2. Data Preparation
    - Loading the Dataset
    - Reshaping the Data
    - Normalization
    - Shuffling and Batching
3. Latent Dimension and Weight Initialization
4. Building the Generator
5. Building the Discriminator
6. Loss Functions
7. Optimizers
8. Custom Training Loop
9. Image Callback
10. Key Takeaways

Project Overview

I built a Generative Adversarial Network (GAN) on the MNIST dataset (handwritten digits).

Environment Setup

My machine lacked CUDA support, but I found an alternative: OpenVINO — an open-source toolkit optimized for Intel processors. I installed it through my Conda environment.

Installed libraries:

- TensorFlow
- TensorFlow-CPU
- Matplotlib
- TensorFlow-Datasets
- Ipywidgets

Importing Modules

```
import tensorflow as tf

import matplotlib.pyplot as plt

import numpy as np

from tensorflow import keras

from tensorflow.keras import layers
```

```
from tensorflow import keras
from tensorflow.keras.layers import Dense
from tensorflow.keras.models import Sequential
```

Data Preparation

Load the Dataset

I prepared the training images by splitting the dataset into:

```
(mnist_train_images,_), (mnist_test_images,_) = keras.datasets.mnist.load_data()
```

Checked the data using matplotlib to ensure images loaded correctly.

Reshaping the Data

Reshaped to (60000, 28, 28) and converted to 32-bit floats:

```
mnist_train_images =  mnist_train_images.reshape(mnist_train_images.shape[0], 28, 28,
1).astype('float32')
```

Normalization

Pixel values (0 to 255) scaled to [-1, 1]:

```
mnist_train_images= (mnist_train_images - 127.5) / 127.5
```

This makes training faster and more stable.

Shuffling and Batching

- **BUFFER_SIZE = 60,000:** Ensures full dataset shuffling to prevent mode collapse.
- **BATCH_SIZE = 256:** Balanced between memory efficiency and training stability.

Converted to a TensorFlow dataset:

```
train_dataset =
tf.data.Dataset.from_tensor_slices(mnist_train_images).shuffle(BUFFER_SIZE).batch(BATC
H_SIZE)
```

Latent Dimensions and Weight Initialization

- **LATENT_DIM = 100:** Ensures the generator learns rich variations while maintaining stability.
- **Weight Initialization:** I used:

```
WEIGHT_UNIT= keras.initializers.RandomNormal(mean=0.01, stddev=0.02)
```

This shifts the mean slightly, giving the generator a head start to produce non-zero outputs.

Building the Generator

I used the DCGAN approach with 12 layers:

1. **Dense Layer:** Projects latent_dim into (7x7x256), no bias (BatchNorm handles it).
2. **Reshape Layer:** Converts to 3D tensor.

3. **3 Conv2DTranspose Layers:** Upscales feature maps.
4. **3 BatchNormalization Layers:** Normalizes activations.
5. **3 LeakyReLU Layers:** Avoids dead neurons with a slope of 0.2.
6. **Output Layer:** Conv2DTranspose with tanh activation.

Layer flow:

- **Input:** Takes random noise. Output: 1D vector (12544 neurons).
- **Reshape:** Converts 1D vector to (7, 7, 256).
- **Upsampling Block:** Output (7, 7, 128).
- **Second Upsampling Block:** Output (14, 14, 128).
- **Final Upsampling Block:** Output (28, 28, 1) grayscale MNIST digit

```
def build_generator():
  model = keras.Sequential([
    layers.Dense(7*7*256, use_bias=False, input_shape=(LATENT_DIM,)),
    layers.BatchNormalization(), # normalize the activation of the previous layer
    layers.LeakyReLU(),

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

    layers.Conv2DTranspose(128, (5, 5), strides=(1, 1), padding='same',
use_bias=False),
    layers.BatchNormalization(),
    layers.LeakyReLU(0.2),

    layers.Conv2DTranspose(128, (5, 5), strides=(2, 2), padding='same',
use_bias=False),
    layers.BatchNormalization(),
    layers.LeakyReLU(0.2),

    layers.Conv2DTranspose(1, (5, 5), strides=(2, 2), padding='same', use_bias=False,
activation='tanh')
  ])
  return model
```

Building the Discriminator

The discriminator acts as a binary classifier that classifies input as real (1) or fake (0):

1. **Conv2D Layer:** Downsamples (28x28) to (14x14). Output: (14, 14, 64).
2. **LeakyReLU:** Prevents dead neurons.
3. **Dropout Layer:** Drops 30% of neurons to prevent overfitting.

4. **Second Conv2D Layer:** Learns higher-level features. Output: (7, 7, 128).
5. **Flatten Layer:** Flattens output to 1D vector (6272).
6. **Output Layer:** Dense with a single neuron — outputs probability.

```
def build_discriminator():

  model = keras.Sequential([

      # First COnv2d layer: Detect low level features in the images (edges, textures,
...)

      layers.Conv2D(64, (5,5), strides=(2,2), padding='same', input_shape=[28,28,1]),



      # LeakyRelu activation function to avoid dead neurons and allow negative
gradients

      layers.LeakyReLU(0.2),

      # dropout to prevent overfitting and improve generalization

      layers.Dropout(0.3),



      # First COnv2d layer: learn more complex feature from the images

      layers.Conv2D(128, (5,5), strides=(2,2), padding='same'),

      layers.LeakyReLU(0.2),

      layers.Dropout(0.3),



      # Flatten the 2D feature maps into a 1D vector for the Dense layer

      layers.Flatten(),



      layers.Dense(1, activation='sigmoid')



  ])
```

```
  return model
```

Loss Functions

Used **Binary Cross-Entropy Loss**:

```
cross_entropy = keras.losses.BinaryCrossentropy()
```

real_loss = tf.keras.losses.binary_crossentropy(tf.ones_like(real_output), real_output)
fake_loss = tf.keras.losses.binary_crossentropy(tf.zeros_like(fake_output), fake_output)
total_loss = real_loss + fake_loss

```
def discriminator_loss(real_output, fake_output):
    real_loss = cross_entropy(tf.ones_like(real_output), real_output)
    fake_loss = cross_entropy(tf.zeros_like(fake_output), fake_output)
    total_loss = real_loss + fake_loss

    return total_loss
```

Optimizers

I used two separate Adam optimizers:

generator_optimizer = tf.keras.optimizers.Adam(1e-4, beta_1=0.5)
discriminator_optimizer = tf.keras.optimizers.Adam(1e-4, beta_1=0.5)

```
generator_optimizer = keras.optimizers.Adam(learning_rate=0.0002, beta_1=0.5)
discriminator_optimizer = keras.optimizers.Adam(learning_rate=0.0002, beta_1=0.05)
```

The lowered momentum helps stabilize training.

Custom Training Loop

I built a custom loop to track progress and handle both networks:

- **compile():** Stores optimizers and losses.
- **train_step():** Handles generator/discriminator updates with gradients.

```
# Building our sub ckass model

class GAN(keras.Model):

  def __init__(self, generator, discriminator):

    # initilize the GAN model  with the generator and discriminator

    super(GAN, self).__init__()

    self.generator = generator #set genrator model

    self.discriminator = discriminator # set discriminator moel




  def compile(self, g_optimizer, d_optimizer, g_loss, d_loss):

    super(GAN, self).compile()

    self.g_optimizer = g_optimizer # set the generator optimizer

    self.d_optimizer = d_optimizer # set the discriminator optimizer

    self.g_loss = g_loss # set the generator loss

    self.d_loss = d_loss # set the discriminator loss




  def train_step(self, real_images):

    # Training logic

    batch_size = tf.shape(real_images)[0]


    # Train discriminator

    noise = tf.random.normal([batch_size, LATENT_DIM])
```

```
    with tf.GradientTape() as d_tape:
        generated_images = self.generator(noise)
        real_output = self.discriminator(real_images)
        fake_output = self.discriminator(generated_images)
        d_loss = self.d_loss(real_output, fake_output)

    # Cimputing the gradients for the discriminator based in its loss
    d_gradients = d_tape.gradient(d_loss, self.discriminator.trainable_variables)
    self.d_optimizer.apply_gradients(zip(d_gradients,
self.discriminator.trainable_variables))


    # Train generator
    noise = tf.random.normal([batch_size, LATENT_DIM])
    with tf.GradientTape() as g_tape:
        generated_images = self.generator(noise)
        fake_output = self.discriminator(generated_images)
        g_loss = self.g_loss(fake_output)

    g_gradients = g_tape.gradient(g_loss, self.generator.trainable_variables)
    self.g_optimizer.apply_gradients(zip(g_gradients,
self.generator.trainable_variables))


    return {"d_loss": d_loss, "g_loss": g_loss}
```

Callback for Image Generation

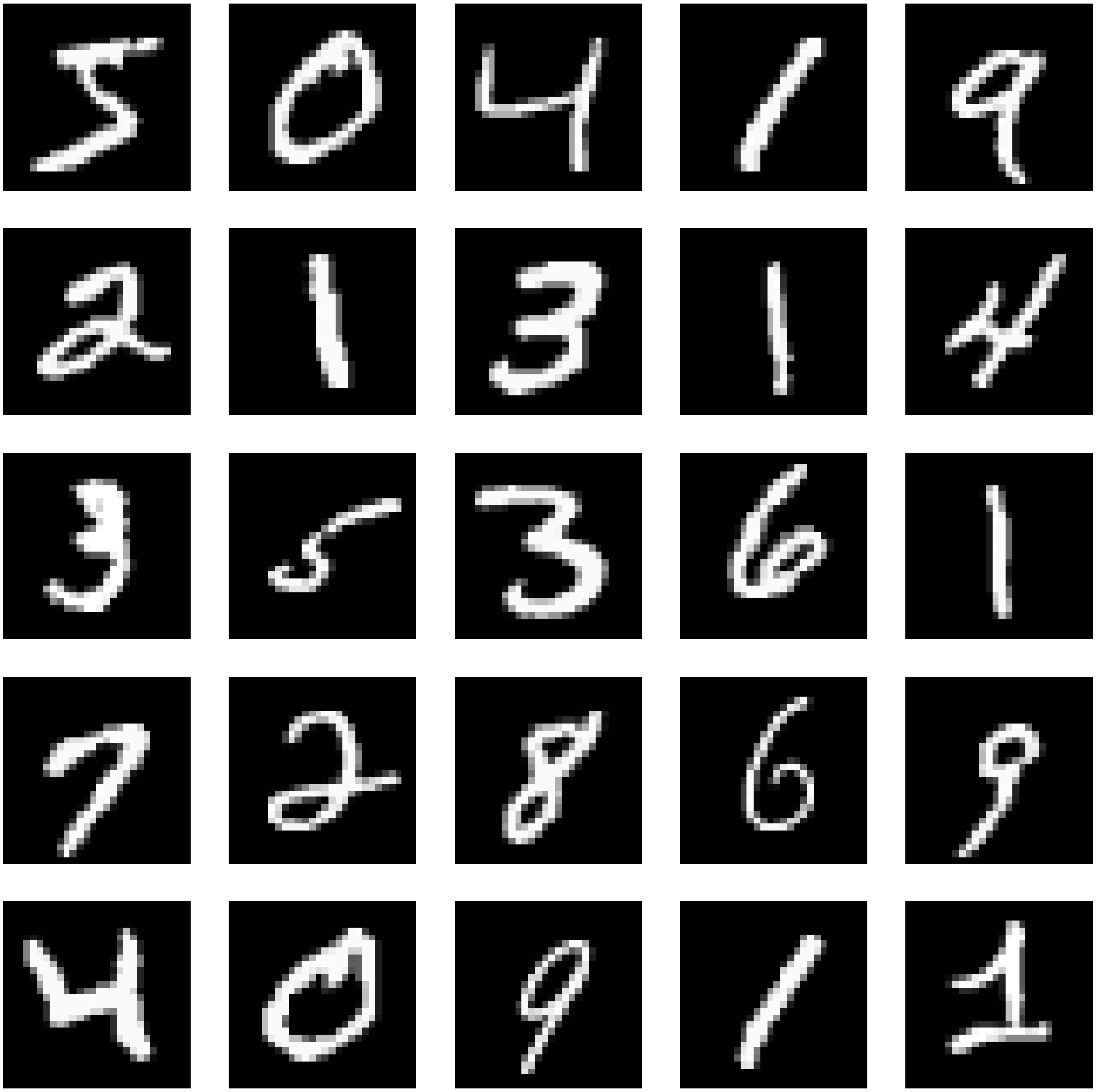
Generated images displayed every 5 epochs:

```
class ImageCallback(keras.callbacks.Callback):
    def __init__(self, num_images=16, latent_dim=100):
        self.num_images = num_images
        self.latent_dim = latent_dim
        self.seed = tf.random.normal([num_images, latent_dim])

    def on_epoch_end(self, epoch, logs=None):
        if epoch % 5 == 0:
            generated_images = self.model.generator(self.seed)
            generated_images = (generated_images * 127.5) + 127.5

            plt.figure(figsize=(10,10))
            for i in range(self.num_images):
                plt.subplot(4, 4, i+1)
                plt.imshow(generated_images[i].numpy().astype("uint8"), cmap="gray")
                plt.axis("off")
            plt.show()
```

```
class ImageCallback(keras.callbacks.Callback):

  def __init__(self, num_images=16, latent_dim=100):

    self.num_images = num_images

    self.latent_dim = latent_dim

    self.seed = tf.random.normal([num_images, latent_dim])



  def on_epoch_end(self, epoch, logs=None):

    if epoch % 5 == 0:

      generated_images = self.model.generator(self.seed)

      generated_images = (generated_images * 127.5) + 127.5 #rescaleto range [0, 233]



      plt.figure(figsize=(10,10))

      for i in range(self.num_images):
```

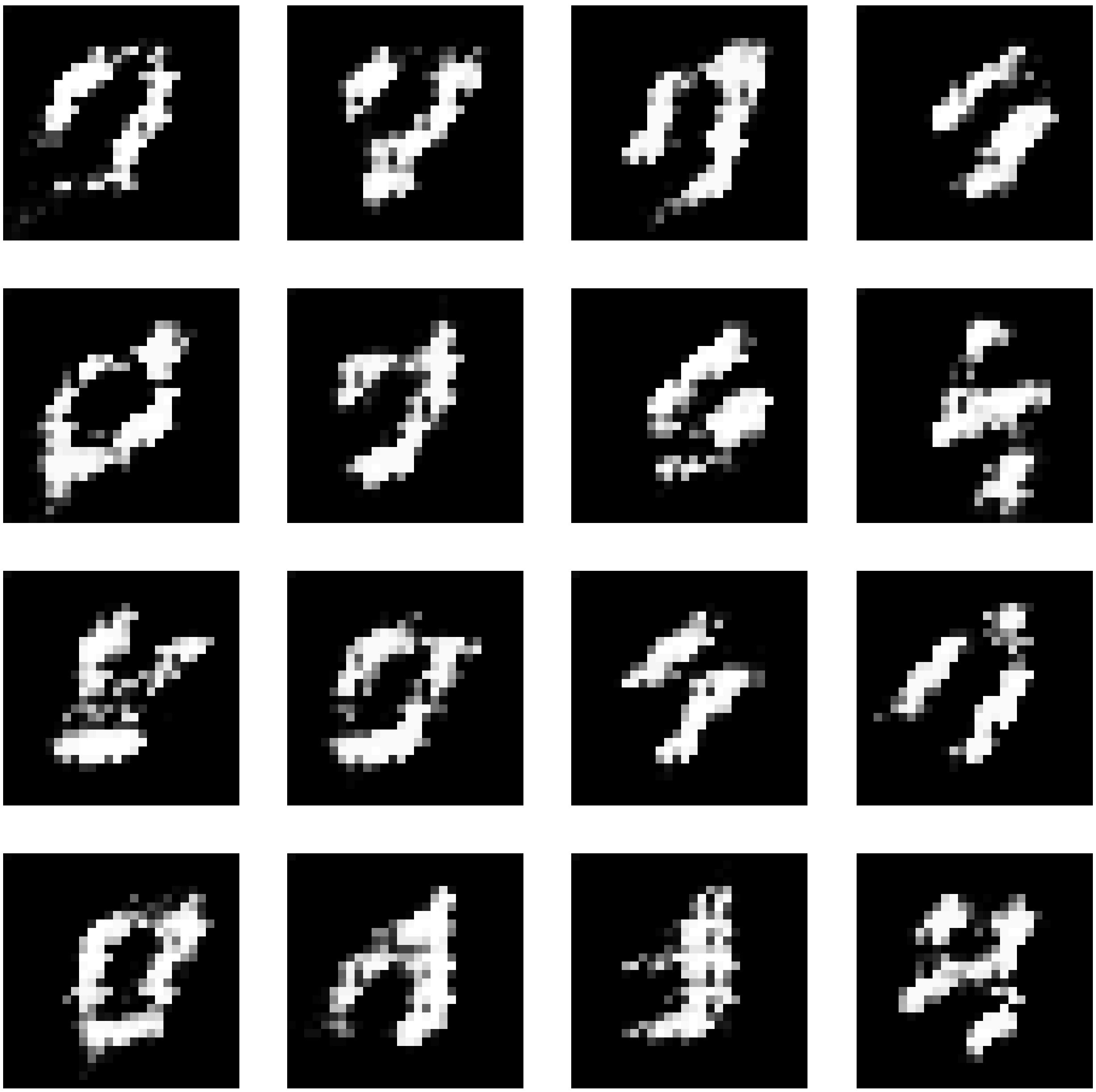
```
    plt.subplot(4, 4, i+1)

    plt.imshow(generated_images[i].numpy().astype("uint8"), cmap="gray")

    plt.axis("off")

  plt.show()
```

Initial dataset:



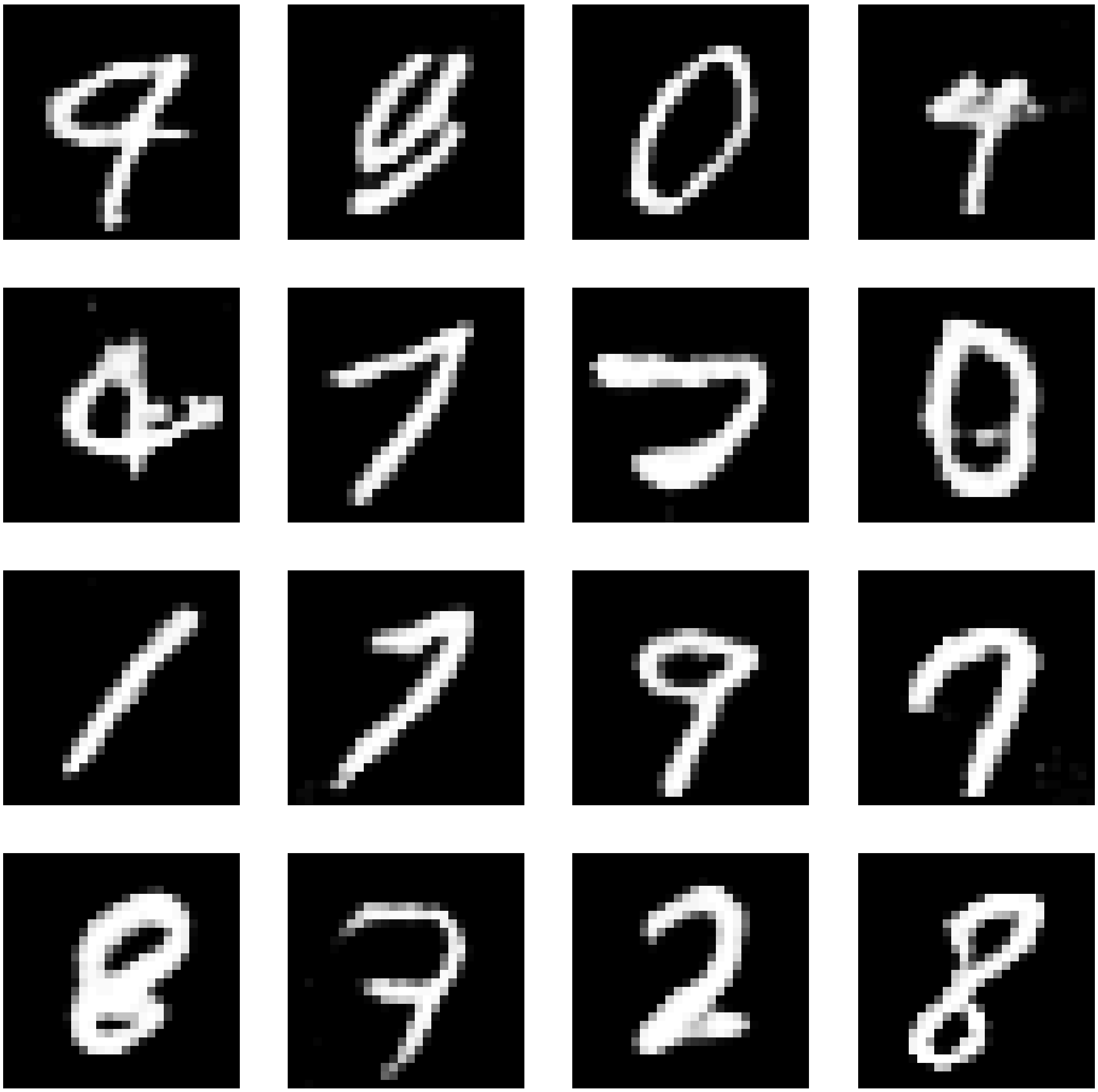
Epoch 1/100

[1m235/235[0m [32m━━━━━━━━━━━━━━━━━━━━[0m[37m[0m [1m0s[0m 2s/step - d_loss: 1.1056 - g_loss: 0.9011

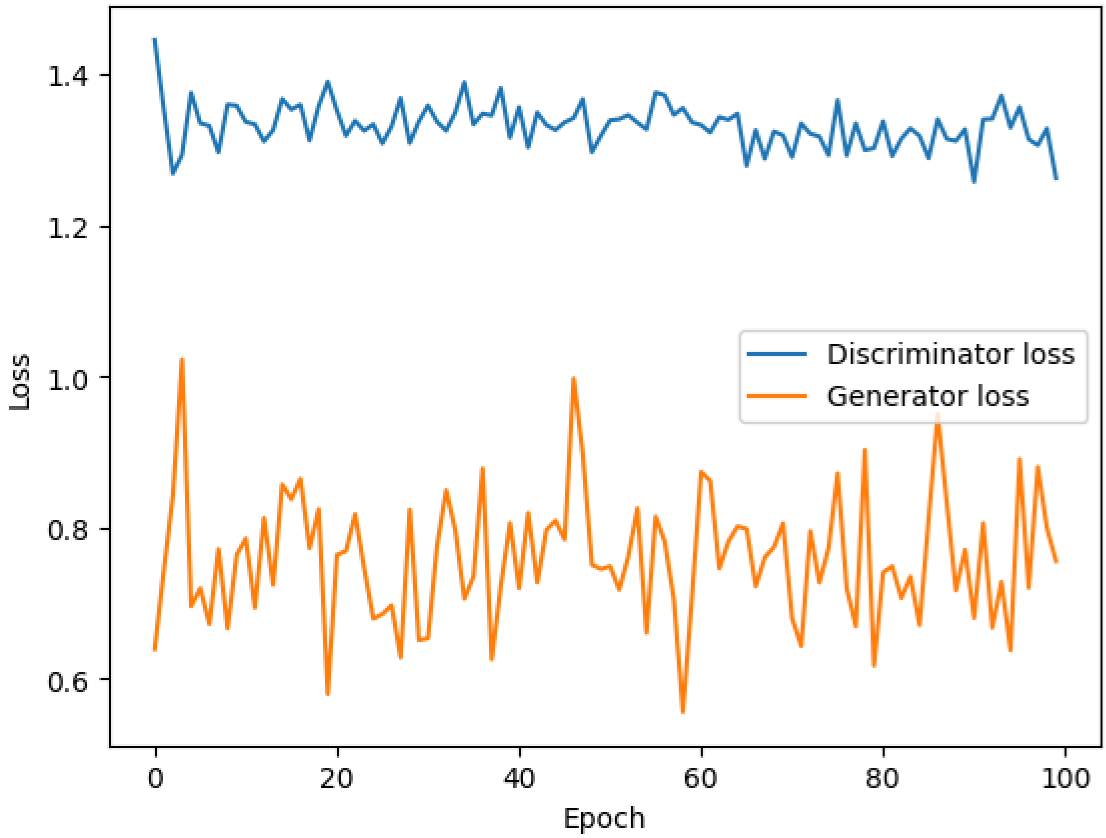


51st Epoch

Epoch 51/100
[1m235/235[0m [32m━━━━━━━━━━━━━━━━━━━━[0m[37m[0m [1m0s[0m 2s/step - d_loss: 1.3191 - g_loss: 0.7876



The model took around 16hours to train
Averagely an epoch took 400s



Final Thoughts

This project taught me the importance of balancing generator-discriminator power and how hyperparameters like batch size, latent dim, and weight initialization affect performance.