Implementing Generative Adversarial Networks (GANs)

Here, we will create a comprehensive implementation of a Generative Adversarial Network (GAN) in Python. We'll go through the entire process step-by-step, from data generation to model training and evaluation. This implementation will be self-contained, generating its own data without requiring an external dataset.

Prerequisite libraries: Tensorflow and Keras

```
$pip install tensorflow
$pip install keras
```

Open the nano editor to write down the code

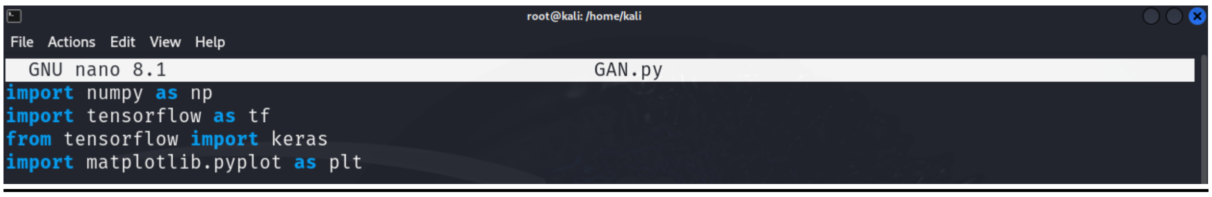
```
$nano GAN.py
```

```
┌──(kali㉿kali)-[~]
└─$ nano GAN.py
```

The code can be obtained from https://github.com/avnishnaithani/pythonforcybersecurity/blob/main/Domain%208/GAN.py

let's break down the implementation and discuss each step of the program:

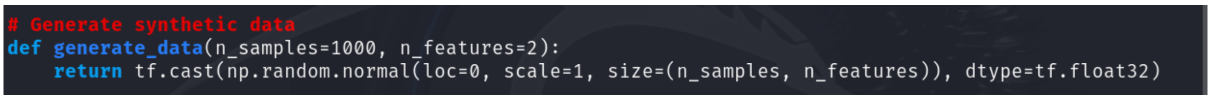
Import Required Libraries



```
  GNU nano 8.1                                          GAN.py
import numpy as np
import tensorflow as tf
from tensorflow import keras
import matplotlib.pyplot as plt
```

Data Generation

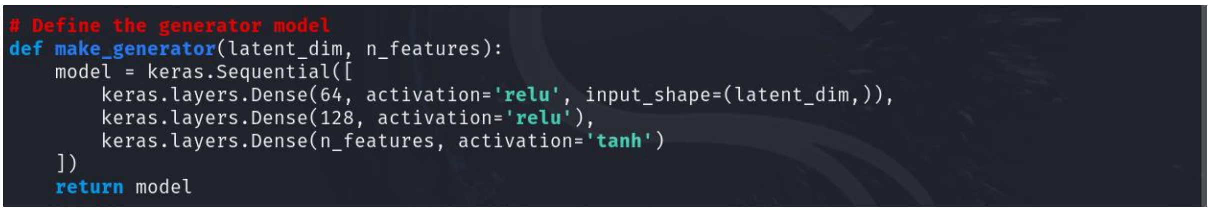
We created a generate_data function to produce synthetic data. This function generates random samples from a normal distribution with 2 features.



```python
# Generate synthetic data
def generate_data(n_samples=1000, n_features=2):
    return tf.cast(np.random.normal(loc=0, scale=1, size=(n_samples, n_features)), dtype=tf.float32)
```

Model Architecture

Generator



```python
# Define the generator model
def make_generator(latent_dim, n_features):
    model = keras.Sequential([
        keras.layers.Dense(64, activation='relu', input_shape=(latent_dim,)),
        keras.layers.Dense(128, activation='relu'),
        keras.layers.Dense(n_features, activation='tanh')
    ])
    return model
```

1. keras.layers.Dense(64, activation='relu', input_shape=(latent_dim,)):
   - This is the first layer of the generator.
   - Dense means every neuron in this layer is connected to every neuron in the previous layer.
   - 64 is the number of neurons in this layer. We chose 64 as a starting point - it's not too small to limit learning, nor too large to cause overfitting.
   - activation='relu': ReLU (Rectified Linear Unit) is an activation function. It helps the network learn complex patterns. Think of it as a switch that decides whether to pass information forward or not.
   - input_shape=(latent_dim,): This specifies the shape of the input. latent_dim is the size of the random noise vector we feed into the generator.
2. keras.layers.Dense(128, activation='relu'):
   - This is the second layer.

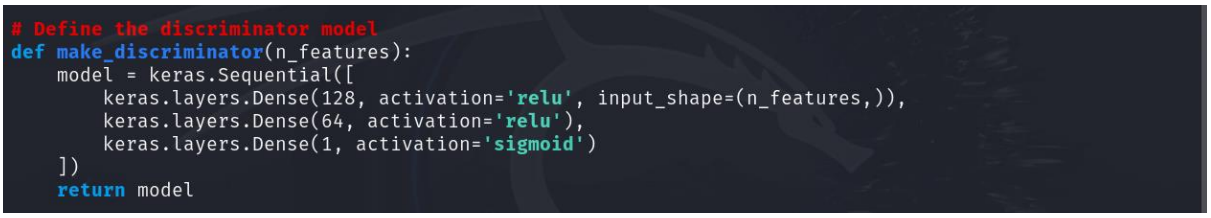
- We increase to 128 neurons to allow the network to learn more complex patterns.
- We use ReLU again for the same reasons as before.

3. keras.layers.Dense(n_features, activation='tanh'):

   - This is the output layer.
   - n_features is the number of features in our data (in this case, 2).
   - We use tanh activation here because it outputs values between -1 and 1, which matches our normalized data range.

Now, let's look at the discriminator:

Discriminator



```
# Define the discriminator model
def make_discriminator(n_features):
    model = keras.Sequential([
        keras.layers.Dense(128, activation='relu', input_shape=(n_features,)),
        keras.layers.Dense(64, activation='relu'),
        keras.layers.Dense(1, activation='sigmoid')
    ])
    return model
```

1. keras.layers.Dense(128, activation='relu', input_shape=(n_features,)):

   - This is the first layer of the discriminator.
   - We start with 128 neurons to capture complex patterns in the input data.
   - input_shape=(n_features,) specifies that the input shape matches our data features.

2. keras.layers.Dense(64, activation='relu'):

   - We reduce to 64 neurons in this layer. This helps the network focus on the most important features.

3. keras.layers.Dense(1, activation='sigmoid'):

   - This is the output layer.

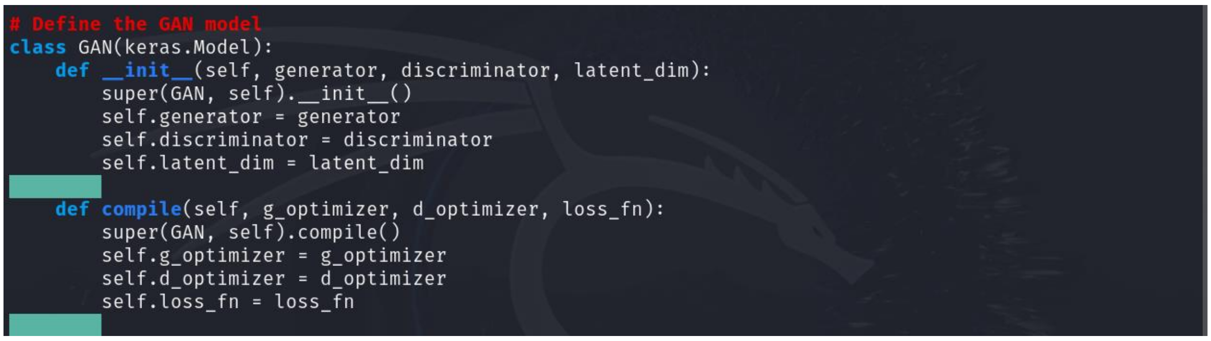
- It has only 1 neuron because the discriminator's job is to output a single value: the probability that the input is real (close to 1) or fake (close to 0).
- sigmoid activation is used because it squashes the output between 0 and 1, which is perfect for representing a probability.

Why these specific numbers? The choice of 64 and 128 neurons is somewhat arbitrary. They're "Goldilocks" numbers - not too small, not too big. For a simple problem like this, they work well. For more complex problems, you might need more neurons.

Why this structure? We generally start with a smaller number of neurons and increase them, then decrease again. This allows the network to first expand its understanding of the data, then concentrate on the most important features.

Remember, there's no one-size-fits-all in neural network architecture. These choices work well for our simple example, but for different problems, you might need to experiment with different architectures.

<u>GAN Implementation</u>



```
# Define the GAN model
class GAN(keras.Model):
    def __init__(self, generator, discriminator, latent_dim):
        super(GAN, self).__init__()
        self.generator = generator
        self.discriminator = discriminator
        self.latent_dim = latent_dim

    def compile(self, g_optimizer, d_optimizer, loss_fn):
        super(GAN, self).compile()
        self.g_optimizer = g_optimizer
        self.d_optimizer = d_optimizer
        self.loss_fn = loss_fn
```

1. class GAN(keras.Model):

- This line creates a new class called GAN that inherits from keras.Model.
- Inheriting from keras.Model gives our GAN class a lot of built-in functionality for training and evaluation.

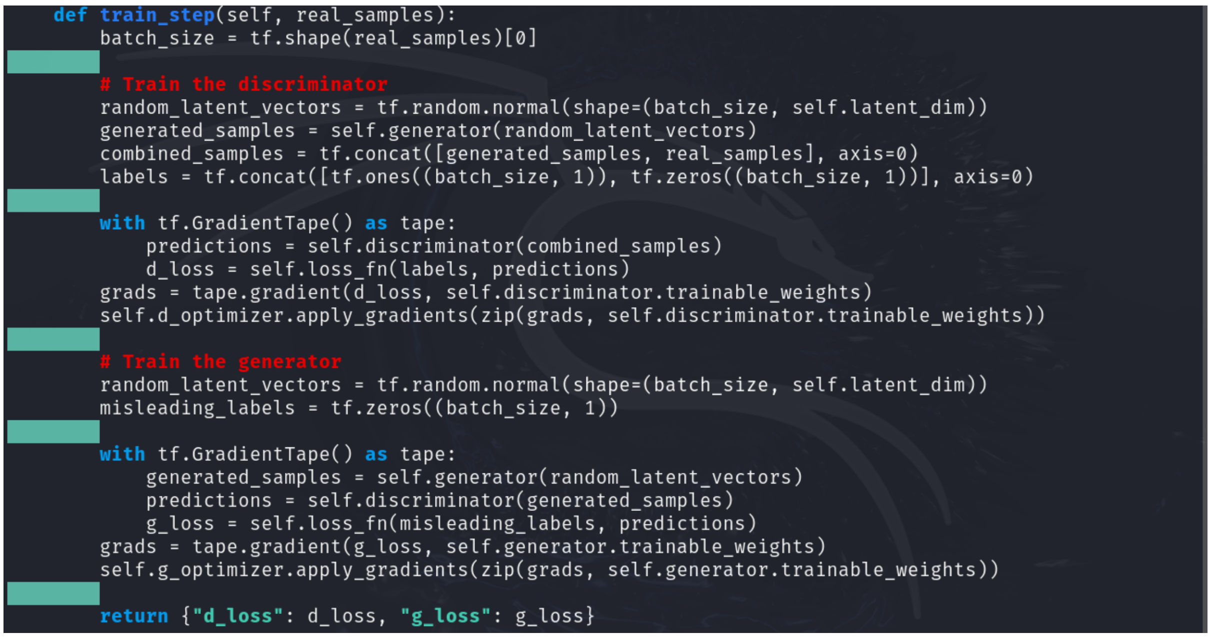
2. def __init__(self, generator, discriminator, latent_dim):
   - This is the constructor method. It's called when we create a new GAN object.
   - It takes three parameters:
     - generator: The generator model we created earlier.
     - discriminator: The discriminator model we created earlier.
     - latent_dim: The size of the random noise vector input to the generator.
3. super(GAN, self).__init__()
   - This line calls the constructor of the parent class (keras.Model).
   - It's necessary to properly initialize the keras.Model functionality.
4. self.generator = generator etc.
   - These lines store the generator, discriminator, and latent_dim as attributes of the GAN object.
   - This allows us to access these components later in other methods.

Now, let's look at the compile method:

1. def compile(self, g_optimizer, d_optimizer, loss_fn):
   - This method sets up the training configuration for our GAN.
   - It takes separate optimizers for the generator and discriminator, and a loss function.
2. super(GAN, self).compile()
   - This calls the compile method of the parent class (keras.Model).
3. self.g_optimizer = g_optimizer etc.

- These lines store the optimizers and loss function as attributes of the GAN object.

Finally, let's break down the train_step method:



```
    def train_step(self, real_samples):
        batch_size = tf.shape(real_samples)[0]

        # Train the discriminator
        random_latent_vectors = tf.random.normal(shape=(batch_size, self.latent_dim))
        generated_samples = self.generator(random_latent_vectors)
        combined_samples = tf.concat([generated_samples, real_samples], axis=0)
        labels = tf.concat([tf.ones((batch_size, 1)), tf.zeros((batch_size, 1))], axis=0)

        with tf.GradientTape() as tape:
            predictions = self.discriminator(combined_samples)
            d_loss = self.loss_fn(labels, predictions)
        grads = tape.gradient(d_loss, self.discriminator.trainable_weights)
        self.d_optimizer.apply_gradients(zip(grads, self.discriminator.trainable_weights))

        # Train the generator
        random_latent_vectors = tf.random.normal(shape=(batch_size, self.latent_dim))
        misleading_labels = tf.zeros((batch_size, 1))

        with tf.GradientTape() as tape:
            generated_samples = self.generator(random_latent_vectors)
            predictions = self.discriminator(generated_samples)
            g_loss = self.loss_fn(misleading_labels, predictions)
        grads = tape.gradient(g_loss, self.generator.trainable_weights)
        self.g_optimizer.apply_gradients(zip(grads, self.generator.trainable_weights))

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

This method performs one training step. It's called for each batch of data during training. Here's what it does:

1. It generates fake samples using the generator.
2. It combines these fake samples with real samples from the dataset.
3. It trains the discriminator to distinguish between real and fake samples.
4. It then trains the generator to produce samples that can fool the discriminator.

The tf.GradientTape() context is used to automatically compute gradients, which are then used to update the model weights using the optimizers.

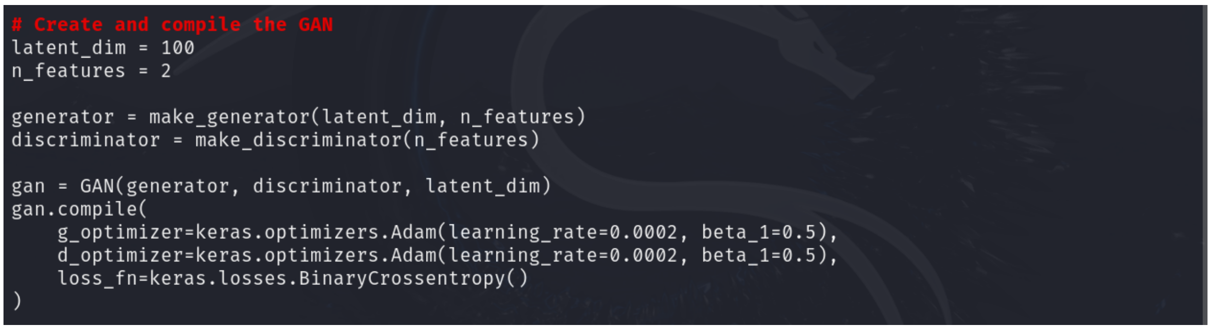
The method returns a dictionary with the loss values for both the discriminator and generator, which are used for monitoring the training progress.

This implementation follows the basic GAN training algorithm:

1. Train the discriminator on real and fake data.
2. Train the generator to fool the discriminator.

By alternating between these steps, the generator learns to produce more realistic data, while the discriminator gets better at distinguishing real from fake data.

Create and compile the GAN



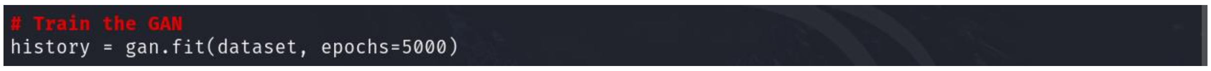
```
# Create and compile the GAN
latent_dim = 100
n_features = 2

generator = make_generator(latent_dim, n_features)
discriminator = make_discriminator(n_features)

gan = GAN(generator, discriminator, latent_dim)
gan.compile(
    g_optimizer=keras.optimizers.Adam(learning_rate=0.0002, beta_1=0.5),
    d_optimizer=keras.optimizers.Adam(learning_rate=0.0002, beta_1=0.5),
    loss_fn=keras.losses.BinaryCrossentropy()
)
```

1. latent_dim = 100:
   - This sets the dimension of the random noise vector input to the generator.
   - 100 is a common choice, but you could experiment with different values.
2. n_features = 2:
   - This specifies the number of features in our data.
   - We're working with 2D data, so we set this to 2.
3. generator = make_generator(latent_dim, n_features):
   - This creates our generator model using the function we defined earlier.

4. discriminator = make_discriminator(n_features):
   - This creates our discriminator model using the function we defined earlier.
5. gan = GAN(generator, discriminator, latent_dim):
   - This creates our GAN object, combining the generator and discriminator.

These steps set up our GAN with all the necessary components for training. We've prepared our data in a format that TensorFlow can efficiently work with, created our generator and discriminator models, combined them into a GAN, and set up the training configuration.

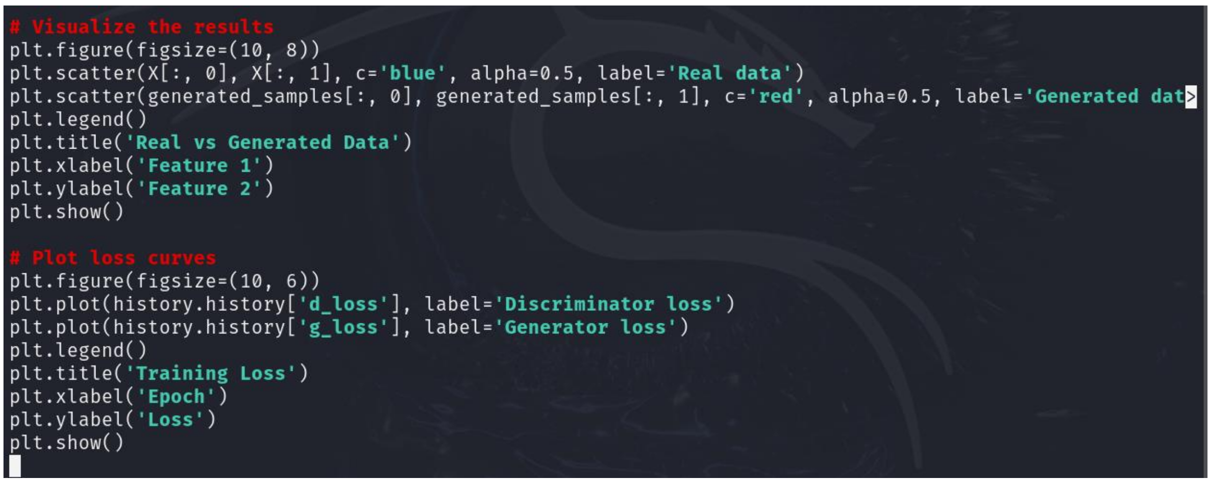
Training Process



```python
# Train the GAN
history = gan.fit(dataset, epochs=5000)
```

The GAN is trained for 5000 epochs on the generated dataset.

Visualization



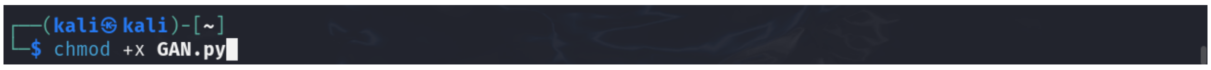
```python
# Visualize the results
plt.figure(figsize=(10, 8))
plt.scatter(X[:, 0], X[:, 1], c='blue', alpha=0.5, label='Real data')
plt.scatter(generated_samples[:, 0], generated_samples[:, 1], c='red', alpha=0.5, label='Generated dat>
plt.legend()
plt.title('Real vs Generated Data')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.show()

# Plot loss curves
plt.figure(figsize=(10, 6))
plt.plot(history.history['d_loss'], label='Discriminator loss')
plt.plot(history.history['g_loss'], label='Generator loss')
plt.legend()
plt.title('Training Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.show()
```

After training, we generate new samples and visualize them alongside the real data. We also plot the loss curves to monitor the training progress.

Now Save the file.

Convert it to an executable one using:

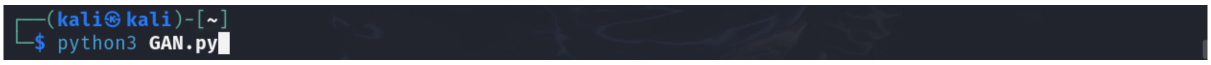
$ **`chmod +x GAN.py`**



```
┌──(kali㉿kali)-[~]
└─$ chmod +x GAN.py
```

and run the program using:

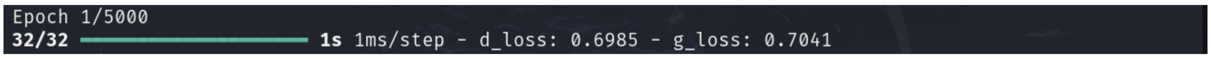
$ **`python3 GAN.py`**



```
┌──(kali㉿kali)-[~]
└─$ python3 GAN.py
```

<u>Output</u>

We get the following output: -



```
Epoch 1/5000
32/32 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - d_loss: 0.6985 - g_loss: 0.7041
```

First, we get 5000 lines of these (As we set training epoch to 5000 in the training process step). This shows the entire training process of our model.

Discriminator Loss (d_loss): The discriminator's job is to distinguish between real data and fake data generated by the generator. Its loss indicates how well it's performing this task:

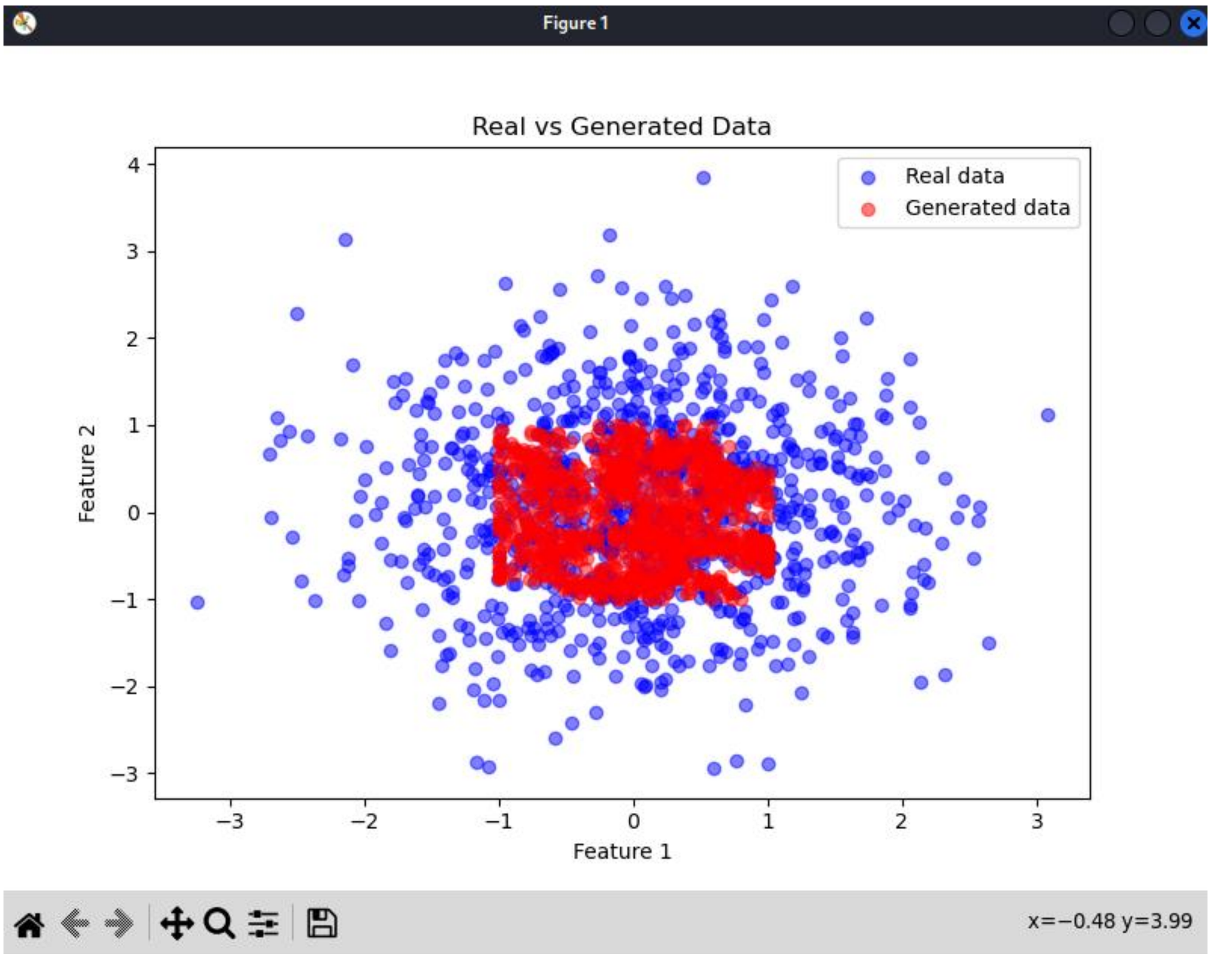
- Lower discriminator loss means the discriminator is better at distinguishing real from fake data.
- Higher loss suggests the discriminator is struggling to tell the difference.

Generator Loss (g_loss): The generator's goal is to create fake data that can fool the discriminator. Its loss reflects how successful it is:

- Lower generator loss means the generator is producing more convincing fake data.

- Higher loss suggests the generator's output is easily identified as fake by the discriminator.

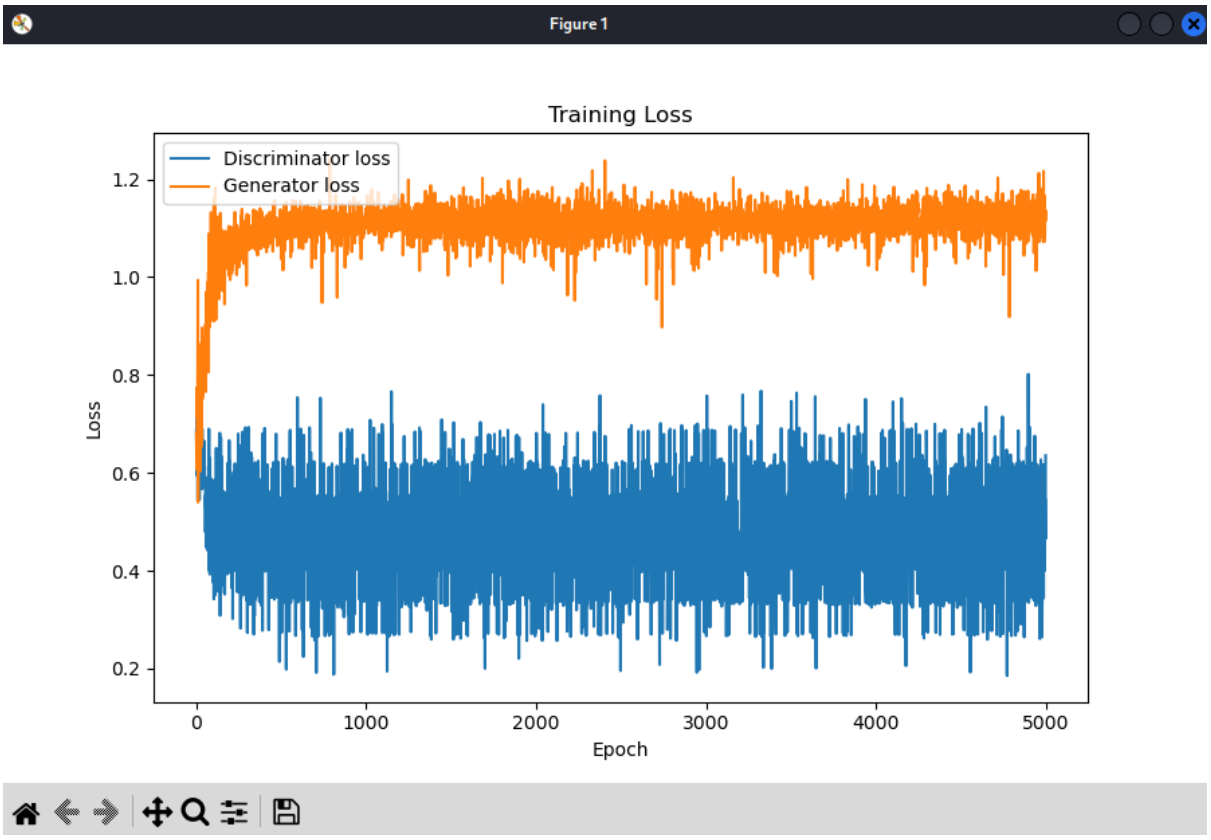
Next, we get the following figure: -



This scatter plot shows the distribution of real data (blue points) versus generated data (red points) in a 2D feature space.

1. Distribution: The real data appears to have a wider, more spread-out distribution, while the generated data is more concentrated in the center.
2. Overlap: There's significant overlap between real and generated data, especially in the central region. This suggests the generator has learned to produce data that resembles the real distribution to some extent.

3. Coverage: The generated data doesn't fully cover the range of the real data, particularly in the outer regions. This indicates that the generator may be struggling to capture the full variability of the real data distribution.
4. Density: The generated data appears denser in the central region, which might suggest mode collapse (a common issue in GANs where the generator produces limited varieties of samples).



This plot shows the training loss for both the discriminator and generator over 5000 epochs.

1. Discriminator loss (blue): It fluctuates but generally stays lower than the generator loss, ranging mostly between 0.2 and 0.8. This suggests the discriminator is performing relatively well in distinguishing real from fake data.
2. Generator loss (orange): It starts low but quickly rises and stabilizes at a higher level than the discriminator loss, mostly

between 1.0 and 1.2. This indicates the generator is consistently challenged in producing data that can fool the discriminator.

3. Relationship: The inverse relationship between the two losses is visible - when one goes up, the other often goes down, reflecting the adversarial nature of GAN training.
4. Non-convergence: The persistent gap between generator and discriminator loss and ongoing fluctuations suggests the GAN hasn't fully converged. This aligns with the limitations observed in the generated data distribution in the figure showing Real vs Generated data.

These outputs indicate that while the GAN has learned to generate data that resembles the real distribution to some extent, there's room for improvement in terms of capturing the full variability of the real data and achieving more stable training dynamics.

The more you will train the GAN, the better it becomes.

Remember, GANs can be challenging to train, and it's not uncommon to need several attempts with different hyperparameters to get good results. The output gives you a visual way to assess how well your GAN is performing and can guide you in making improvements.