Implementing a Generative Adversarial Network (GAN) for Sample Generation in PyTorch

Patrick Naumczyk

August 2025

Abstract

Generative Adversarial Networks (GANs) are a form of neural network that uses two agents to generate synthetic examples that resemble real-world examples. The purpose of this report is to implement GANs through a basic example reproducing the sine function, and to apply the same model to a dataset retrieved from Kaggle relating the perceived levels of anxiety and depression in students. The primary objective was to explore whether GAN could be applied to augment data in a low-resource setting. The results demonstrate that GAN was able to generate synthesised data that closely resembled the original dataset. The synthetic data generated by the GAN is statistically consistent with the real dataset and can be effectively used for additional modelling tasks such as predictive modelling.

1 Introduction

Neural networks are fundamental in the field of data science with applications ranging from computer vision to natural language processing. Generative Adversarial Networks (GANs) are a form of network that uses two agents to generate synthetic examples that resemble real-world examples. Successful use cases of GAN include realistic image synthesis, text generation, and data augmentation.

A GAN is composed of two agents: a generator network $G(Z)$ and a discriminator network $D(X)$. The generator maps random noise z into synthetic samples to replicate the underlying data distribution. The role of the discriminator is to distinguish the synthetic data from the real data. Training is modelled after Game Theory as a minimax game: The generator aims to minimise it's probability of it's outputs being classified as fake by the discriminator. The discriminator aims to maximise it's classification accuracy of it's two inputs. The result is a generator function that can reproduce the input data realistically.

The purpose of this report is to implement GANs through a basic example reproducing the sine function, and to apply the same model to a dataset retrieved from Kaggle relating the perceived levels of anxiety and depression in students.

2 Background

Generative Adversarial Networks is composed of a generator model G and a discriminator model D. The discriminator model is a binary classifier that distinguishes between real data samples and synthetic data produced by the generator model. We begin from the definition of a binary cross-entropy loss function, which for a single sample is defined as

$$L(\hat{y}, y) = y.log(\hat{y}) + (1 - y).log(1 - \hat{y})$$

Here, $\hat{y}$ is the predicted probability and $y \in \{0, 1\}$ is the true label.

If a real sample X passes through the discriminator, the true label is one and the predicted value $D(X)$. The loss reduces to

$$L(D(X), 1) = log(D(X))$$

If a synthetic sample produced by the generator model $G(Z)$ (where Z is a random noise vector) is passed through the discriminator, the true label is zero and the predicted value $D(G(Z))$. The loss reduces to

$$L(D(G(Z)), 0) = \log\left(1 - D(G(Z))\right)$$

The discriminator D therefore maximises the output $\log\left(D(X)\right)$ (correct classifications) and $\log\left(1 - D(X)\right)$ (correct detection of synthetic data). Combining these, the discriminator objective for a single sample becomes

$$\max_{D} V(D, G) = \log\left(D(X)\right) + \log\left(1 - D(G(Z))\right)$$

For a dataset with more than a single sample, a collection of probabilities is expected. We take expectations with respect to the real data distribution $P(X)$ and the noise distribution $P(Z)$. We write

$$\max_D V(D,G) = \mathbb{E}_{X\sim P(X)}[\log(D(X))] + \mathbb{E}_{Z\sim P(Z)}[\log(1 - D(G(Z)))]$$

The role of the generator is to fool the discriminator into predicting its outputs as real data. This action is equivalent to minimising the probability of $G(Z)$ which, in turn, minimises the ability of D to distinguish the generated data. Formally, this is written as

$$\min_G V(D,G) = \mathbb{E}_{X\sim P(X)}[\log(D(X))] + \mathbb{E}_{Z\sim P(Z)}[\log(1 - D(G(Z)))]$$

The minimax problem that defines GAN training is, therefore, written as

$$\min_G \max_D V(D,G) = \mathbb{E}_{X\sim P(X)}[\log(D(X))] + \mathbb{E}_{Z\sim P(Z)}[\log(1 - D(G(Z)))]$$

In practice, training alternates between the discriminator (by freezing the generator) and the generator (by freezing the discriminator). The adversarial dynamic results in data being generated that closely resembles the underlying data distribution.

The ability of Generative Adversarial Networks in mimicking underlying data distributions makes them applicable to data augmentation tasks, useful for generating synthetic but realistic data samples situated for training other models.

Although we have only explored the vanilla version of GANs; GANs have inspired an range of variation to the models such as Deep Convolutional GANs (DCGAN) for image generation, and Conditional GANs (cGAN) used for sample generation.

3 Methodology

3.1 Simple GAN

We begin by defining the function we wish to replicate: the sine function. A set of training data is generated by sampling random x values and computing the corresponding $y = \sin x$. This dataset serves as the real data distribution that the GAN will train on.

```
import numpy as np
import torch
from torch import nn
import math
import matplotlib.pyplot as plt

def generate_y(x):
    return math.sin(x)

def sample_data(n=256, scale=10):
    data = []

    x = scale*(np.random.random_sample((n,))-0.5)

    for i in range(n):
        yi = generate_y(x[i])
        data.append([x[i], yi])

    return torch.tensor(data).float()
```

Listing 1: Generate Sine Function and Sample Data

Following the generation of our data, we can construct the generator and discriminator networks.

```
class Generator(nn.Module):
    def __init__(self, size = 128):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(2, size),
            nn.LeakyReLU(),
            nn.Linear(size, size),
            nn.LeakyReLU(),
            nn.Linear(size, 2),
        )

    def forward(self, Z):
        return self.net(Z)

class Discriminator(nn.Module):
    def __init__(self, size = 128):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(2, size),
            nn.LeakyReLU(),
            nn.Linear(size, size),
            nn.LeakyReLU(),
```

```
            nn.Linear(size, 2),
            nn.Linear(2,1)
        )

    def forward(self, X):
        logit = self.net(X)
        return logit
```

Listing 2: Generator and Discriminator Classes

Two classes are created: the generator and the discriminator. The generator is a sequential neural network composed of connected linear layers with Leaky ReLU activations. The output is a two-dimensional vector, intended to be the input and output values of the sine function.

The discriminator is programmed similarly to the generator. It maps an input vector through its hidden layers to produce a logit. We next define the loss function and the optimisers.

```
loss_fn = nn.BCEWithLogitsLoss()
g_opt = torch.optim.Adam(G.parameters(), lr=0.0002, betas
    =(0.5, 0.999))
d_opt = torch.optim.Adam(D.parameters(), lr=0.0002, betas
    =(0.5, 0.999))
```

Listing 3: Define loss function and optimisers.

Logits are more numerically stable than passing through the sigmoid function. The Binary Cross-Entropy with Logit loss was chosen as the loss function. The Adam optimiser is selected as with the particular beta values as it reduces momentum-induced oscillations during training and is widely used for GANs.

We initialise the generator and discriminator and assign them to the GPU.

```
device = 'cuda' if torch.cuda.is_available() else 'cpu'
G = Generator().to(device)
D = Discriminator().to(device)
```

Listing 4: Intialise G(Z) and D(X)

```
def train_step(x_real):
    G.train()
    D.train()
    bs = x_real.size(0)
    x_real = x_real.to(device)

    # Train discriminator
```

```
    z = torch.randn(bs, 2, device = device)
    with torch.no_grad():
      x_fake = G(z)

    d_opt.zero_grad()
    d_real_logit = D(x_real)
    d_fake_logit = D(x_fake)

    real_targets = torch.ones_like(d_real_logit)
    fake_targets = torch.zeros_like(d_fake_logit)

    d_loss = loss_fn(d_real_logit, real_targets) + loss_fn(
   d_fake_logit, fake_targets)
    d_loss.backward()
    d_opt.step()

    # Train generator
    z = torch.randn(bs, 2, device = device)
    g_opt.zero_grad()
    x_fake = G(z)
    d_fake_logit = D(x_fake)
    g_loss = loss_fn(d_fake_logit, real_targets)
    g_loss.backward()
    g_opt.step()

    return d_loss.item(), g_loss.item()
```

Listing 5: Step function

This function iterates one step of the GAN training. The discriminator receives a real batch of data as well as a fake batch where it produces logits for each vector. The loss is computed, and its parameters are adjusted. The generator produces fake samples; the samples are passed into the discriminator where they are used to calculate the loss for the generator.

We implement a training loop and record loss values.

```
d_losses = []
g_losses = []
for epoch in range(6000):
  d_loss, g_loss = train_step(data)
  d_losses.append(d_loss)
  g_losses.append(g_loss)
  if epoch % 100 == 0:
        print(f"epoch {epoch}: D {d_loss:.3f} | G {g_loss:.3f
```

```
    }")
```

Listing 6: Training loop

Monitoring loss values provides insight into the stability of the training and allows adjustment of the epoch value for efficient training.

3.2 Data Augmentation

Next, we apply a GAN to a real-world dataset to demonstrate its efficacy in generating synthetic data for model training. Two variables were extracted from the dataset: Depression level and anxiety level. This is a simple dataset that did not need pre-processing.

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler

dataset = pd.read_csv('StressLevelDataset.csv')
dataset = dataset[['depression', 'anxiety_level']]
scaled_data = dataset
```

Listing 7: Load Data

```
scaler = MinMaxScaler()
scaled_data[['depression', 'anxiety_level']] = scaler.
    fit_transform(scaled_data[['depression', 'anxiety_level'
    ]])

X_train, X_test, y_train, y_test = train_test_split(
    scaled_data.depression, scaled_data.anxiety_level,
    test_size=0.2, random_state=42)
X_train_xs1, X_train_xs2, y_train_xs1, y_train_xs2 =
    train_test_split(X_train, y_train, train_size=20,
    random_state=42) # small data set (20) to imitate
    limitations

data_xs1 = torch.tensor([X_train_xs1.values, y_train_xs1.
    values]).float().T
```

Listing 8: Scale and set up training sets

After loading the data, the features are scaled to $[0, 1]$ to stabilise GANs training. The datasets are split into training and testing sets. To simulate scarce data, we further split the data into a very small sample size of only 20

data points. Deliberately limiting the training data highlights the potential benefit of augmentation with GANs.

```
num_samples = len(X_train_xs2)
z = torch.randn(num_samples, 2, device=device)
synthetic_data_tensor = G(z)
synthetic_data = synthetic_data_tensor.detach().cpu().numpy()
```

Listing 9: Generate synthetic mental health data

For the GAN architecture, we use the same generator and discriminator from the simple GAN example, but we adjusted the hidden size from 128 to 64 units as it improved stability. We train the models for 3500 epochs (this value was optimised after multiple training sessions).

```
df_synth = pd.DataFrame(synthetic_data)
df_synth.columns = ['depression', 'anxiety_level']

df_synth[['depression', 'anxiety_level']] = scaler.
    inverse_transform(df_synth[['depression', 'anxiety_level'
    ]])

df_synth = df_synth.astype(int)
df_synth = df_synth[((df_synth['depression'] <= 27) & (
    df_synth['anxiety_level'] <= 21))]

dataset = pd.read_csv('StressLevelDataset.csv')
dataset = dataset[['depression', 'anxiety_level']]

X_train, X_test, y_train, y_test = train_test_split(dataset.
    depression, dataset.anxiety_level, test_size=0.2,
    random_state=42)
```

Listing 10: Preprocess synthetic data

The synthetic values are post-processed to ensure that they are realistic and compatible with the original dataset. The generated data is unscaled, values are redefined as integers since the original dataset used discrete scores, and records outside valid scoring ranges ($depression \leq 27$, $anxiety \leq 21$) are discarded.

Finally, the original dataset is reloaded to avoid scaling inconsistencies and the test split is redefined on the unscaled data for evaluation.

```
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score

```

```
X_test_real = X_test.values.reshape(-1, 1)
y_test_real = y_test.values.reshape(-1, 1)

X_train_synth = df_synth.depression.values.reshape(-1, 1)
y_train_synth = df_synth.anxiety_level.values.reshape(-1, 1)

model_synth = LinearRegression()
model_synth.fit(X_train_synth, y_train_synth)

y_pred_synth = model_synth.predict(X_test_real)

mse_synth = mean_squared_error(y_test_real, y_pred_synth)
r2_synth = r2_score(y_test_real, y_pred_synth)
```

Listing 11: Linear regression model for synthetic data

```
X_train_real = dataset.depression.values.reshape(-1, 1)
y_train_real = dataset.anxiety_level.values.reshape(-1, 1)

model_real = LinearRegression()
model_real.fit(X_train_real, y_train_real)

y_pred_real = model_real.predict(X_test_real)

mse_real = mean_squared_error(y_test_real, y_pred_real)
r2_real = r2_score(y_test_real, y_pred_real)
```

Listing 12: Linear regression model for real data

To evaluate the utility of the synthetic data, we trained a linear regression model on GAN-generated samples and compared its performance with a regression model trained on the original dataset. Both models are evaluated on the same test set, using the mean squared error (MSE) and the R^2 score as metrics.

4 Results

4.1 Evaluation of Simple GAN

We first attempted to replicate the sine function using a simple GAN. The target function is shown in figure 1, which represents the sine wave.

The training loss in Figure 2 indicates that the training loss stabilises very quickly with both models converging to a balanced state.

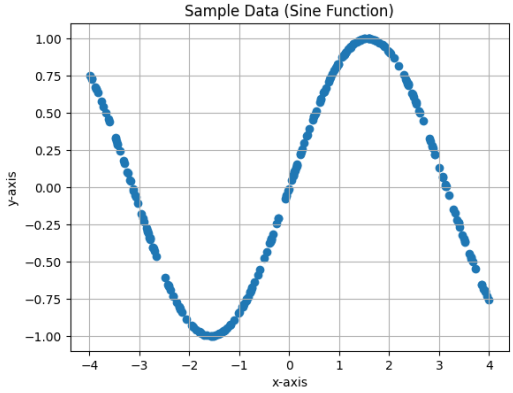

Figure 1: Sample Data (Sine Function)

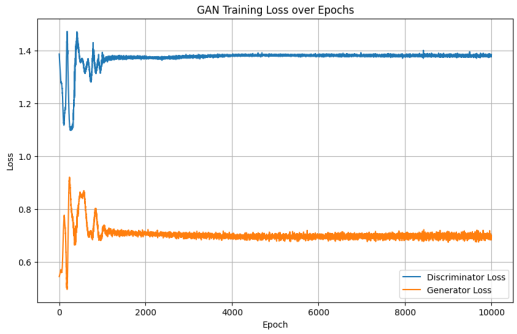

Figure 2: Simple GANs Training Loss Curve

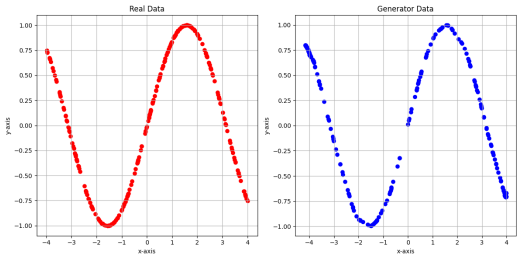

Figure 3: Original Sine Function (left) versus Generated Sine Function (right)

Figure 3 shows the comparison between the actual sine function and the generated sine function. The generated curve captures the oscillatory shape of the sine wave, demonstrating that the GAN was able to approximate the

underlying data distribution.

4.2 Evaluation of augmented data

Next, we evaluate the GAN trained on the student mental health dataset. Original data is shown in Figure 4, where anxiety levels are plotted against depression scores. Although the full dataset contains more than 1100 observations, only 20 were used to train the GAN, simulating a low-resource scenario.

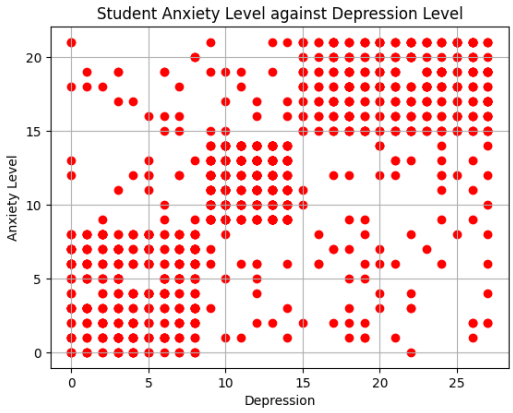

Figure 4: Student Anxiety Level against Depression Level

Figure 5 shows the loss generated during the learning of 3500 epochs of training. The losses for each model quickly converge, demonstrating that the

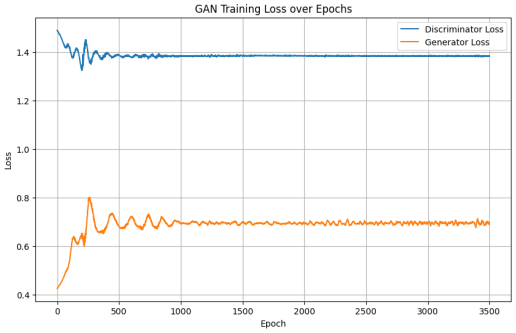

Figure 5: Student Mental Health GANs Training Loss Curve

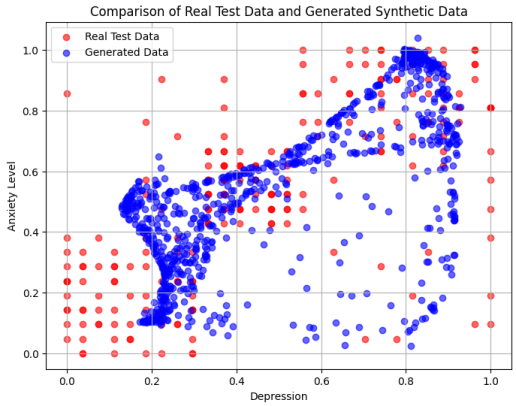

Figure 6: Real Data versus Synthetic Data Preprocessing (Scaled)

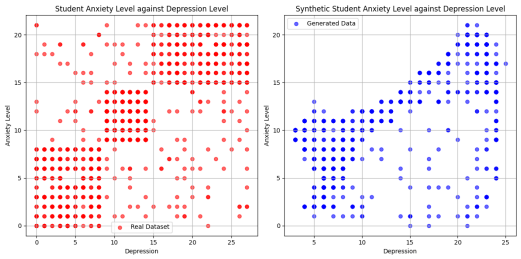

Figure 7: Real Data (left) versus Generated Data (right)

GAN is quickly learning the underlying data distribution.

The 860 points of synthetic data produced by the generator before post-processing is displayed in Figure 6. What looks to be two hands twisting a rubber band, the generated distribution broadly follows the structure of the original dataset, suggesting that the GAN has learnt meaningful patterns even from a very small input sample.

After processing, the generated dataset is compared with the real dataset in Figure 7. The synthetic data points align more closely with the real distribution, but it is not perfect. Some data points are missing in the bottom left and top right quadrants, which removes the square-like structure we see in the real dataset.

To assess the practical value of the synthetic data, linear regression models were trained on both the real and generated datasets. The two resulting models are shown in Figure 8.

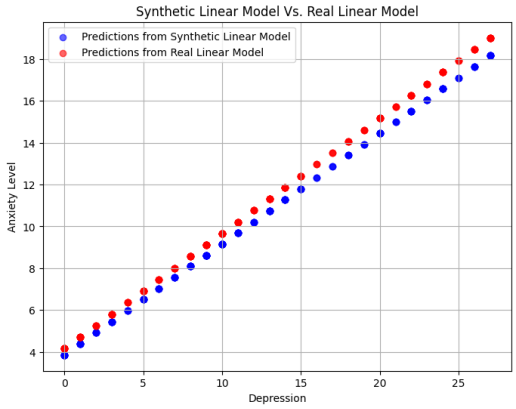

Figure 8: Synthetic Linear Model versus Real Linear Model

The regression model trained on the real dataset achieved an R^2 score of 0.44 and a mean squared error (MSE) of 20.91 on the test set. The regression model trained on GAN-generated data achieved a nearly identical performance, with an R^2 score of 0.44 and an MSE of 20.95. This suggests that the synthetic data generated by the GAN is statistically consistent with the original dataset and can be effectively used for additional modelling tasks such as predictive modelling.

5 Discussion

The implementation of a simple GAN served to introduce the adversarial framework of GANs, rather than an evaluation of its performance. The model was tasked with reconstructing a sine function. The loss curves in Figure 2, show the loss stabilising for the discriminator approximately 1.39 for the discriminator function and 0.7 for the generator function. The loss of the generator function becomes unstable with the number of epochs, suggesting difficulty consistently fooling the discriminator.

Although no error diagnostics were performed, the comparison of the real sine wave and the sine wave generated in Figure 3 is visually similar. This suggests that data with clear underlying distributions can be recognised even through a minimal GAN architecture.

We next applied the GAN framework to the student mental health dataset in figure 4. Data were processed to show student levels of depression versus anxiety levels (Figure 4). Only 20 data points, out of the original 1100, were used in training to simulate a low-resource environment.

Training was initially attempted for 10,000 epochs, but noticed increasing instability of the generator's loss around 4000 epochs. This led to adjusting the epochs parameter to 3500.

The generator model produced 840 additional samples from the limited training set (Figure 6). The raw GAN output were continuous floating point values, and post-processing converted them into discrete scores aligning with the original data format. After this step, the synthetic data resembled the real data more closely (Figure 7).

Figure 8 and the results section demonstrated that the GANs generated a data structure virtually identical to the original data. Although the data look a bit different, modelling the data with linear regression yields the same results. Although the structure was not entirely similar, the model effectively captured the underlying distribution.

Most importantly, the linear regression model trained on the synthetic data produced metrics nearly identical to the real data, demonstrating that the generated data, while not identical in appearance, retained sufficient structure for predictive modelling.

Although promising, the dataset used here is simple, discrete values with only two variables and a clear linear trend. The results produced by this simple example cannot be taken as evidence that GAN-based augmentation will universally succeed. Further exploration in datasets with continuous values and high-dimensionality with GANs is required.

Additionally, the models used a standard choice of activation functions, loss, and optimisers, with minimal hyperparameter tuning. Incorporating more advanced variations of GANs, such as Conditional GANs (cGAN) could improve both the stability and fidelity of generated examples.

Future work should expand the evaluation by applying GAN-based augmentation to more complex datasets, as well as exploring variations of GANs for data augmentation. Comparing GAN augmentation with traditional augmenting methods would also clarify its advantages. Such work would help establish the credibility of GANs as a reliable tool for synthetic data genera-

tion, particularly in data science scenarios where traditional modelling risks overfitting.

6 Conclusion

A Generative Adversarial Network (GAN) was implemented and tested for sample generation in PyTorch using a dataset that measures student depression and anxiety levels. The primary objective was to explore whether GANs could be applied to augment data in a low-resource setting.

The results demonstrate that GAN was able to generate synthetic data that closely resembled the original dataset. When evaluated using a linear regression model, the performance of the model trained on the synthetic data was nearly identical to that of the model trained on the real data, indicating that the generated samples retained the underlying structure of the distribution.

Although the dataset was simple and with a clear trend, the project highlights the potential applications of GANs as a tool for data augmentation in data-scarce scenarios. Synthetic data could support model validation, mitigate overfitting, and expand the range of experimental testing where real data are limited. However, broader experimentation across more complex and diverse datasets is required before strong claims can be made about the reliability of GAN-based sample generation.

7 Appendix

7.1 Code Repository

The complete implementation of the models and experiments can be found at: `https://github.com/patricknzk/GANs-PyTorch`

7.2 Datasets

- Original Dataset: `https://docs.google.com/spreadsheets/d/1m5fA4JStMOkqPs3037BIbzof3_o6B8bRg6I3XAnoPr0/edit?usp=sharing`
- Synthetic Dataset: `https://docs.google.com/spreadsheets/d/179kdh_YrWj211asaOyac4U6TP7nhTqBse30HhCy3iw0/edit?usp=sharing`

8 References

1. DigitalOcean. (2020). *Implementing GANs in TensorFlow*. Retrieved from https://www.digitalocean.com/community/tutorials/implementing-gans-in-tensorflow

2. Analytics Vidhya. (2021). *Understanding GANs: Deriving the Adversarial Loss from Scratch*. Retrieved from https://medium.com/analytics-vidhya/understanding-gans-deriving-the-adversarial-loss-from-scratch-ccd8b683d7e2

3. IBM. (n.d.). *Generative Adversarial Networks (GANs)*. Retrieved from https://www.ibm.com/think/topics/generative-adversarial-networks

4. Kaggle. (2022). *Student Stress Monitoring Datasets*. Retrieved from https://www.kaggle.com/datasets/mdsultanulislamovi/student-stress-monitoring-datasets

5. Apxml. (2023). *GAN Training Stability and Hyperparameter Tuning*. Retrieved from https://apxml.com/courses/synthetic-data-gans-diffusion/chapter-3-gan-training-stability-optimization/gan-hyperparameter-tuning