An introduction to Generative Adversarial Networks (GANs) 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 and populating the sine function, and to produce synthetic handwritten digits using the MNIST dataset. The primary objective was to learn, implement and demonstrate that GANs can be used to generate artificial data. The experiments demonstrated that a simple GAN could populate and approximate the underlying structure of sparse periodic data, while a conditional deep convolutional GAN could produce realistic image samples with a large, structured dataset. Visual evaluation showed that the generated output retained the essential features of the training data, with fidelity improving as the size of the dataset and the architectural complexity increased. Overall, this implementation illustrates the core adversarial dynamics and provides a foundation for future work applying generative models in practical data science contexts.

1 Introduction

Neural networks are fundamental in the field of data science with applications ranging from computer vision to natural language processing (Goodfellow et al., 2016). 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, fraud detection, and text generation (Goodfellow et al., 2014).

A GAN is made up of two agents: a generator network $G(Z)$ and a discriminator network $D(X)$. The generator maps random noise $\hat{z}$ 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 the probability that its outputs are classified as fake by the discriminator. The discriminator aims to maximise its classification accuracy of its two inputs. The result is a generator function that can reproduce the input data realistically.

The purpose of this report is to learn and implement GANs through a basic example reproducing and populating the sine function, and to train and evaluate the performance of a GAN trained on the famous MNIST dataset.

2 Background

Generative Adversarial Networks are made up of a generator model G and a discriminator model D (Goodfellow et al., 2014). The discriminator model is a binary classifier that distinguishes between real data samples and synthetic data produced by the generator model. By deriving the loss for an adversarial network, we can understand more deeply how we generate realistic data. 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 $\log(D(X))$ (correct classifications) and $\log(1 - D(X))$ (correct detection of synthetic data). Combining these, the discriminator objective for a single sample becomes

$$\max_D V(D, G) = \log(D(X)) + \log(1 - D(G(Z)))$$

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 shown in the loss equation results in data being generated that closely resembles the underlying data distribution. GANs have inspired a range of models, such as Deep Convolutional GANs (DCGANs) (Radford et al., 2016) for image generation and Conditional GANs (cGANs) (Mirza & Osindero, 2014) used for conditional sample generation. These two variations will be utilised in training the GAN on the MNIST dataset.

3 Methodology

3.1 GAN: Sine Wave

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. Listing 1 demonstrates how this is programmed in Python.

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

def sample_data(n=32, scale=8):
    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

```
class Generator(nn.Module):
    def __init__(self, size = 512):
        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 = 512):
        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):
        return self.net(X)
```

Listing 2: Generator and Discriminator Classes

In Listing 2, 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 inspired by a baseline GAN in TensorFlow (DigitalOcean, 2020). 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(generator.parameters(), lr
    =9*10**(-5), betas=(0.5, 0.999))
d_opt = torch.optim.Adam(discriminator.parameters(), lr
    =9*10**(-5), betas=(0.5, 0.999))

generator = Generator().cuda()
discriminator = Discriminator().cuda()
```

Listing 3: Define loss function, optimisers, initialise generator and discriminator functions

Listing 3 details the loss function and optimiser: Binary Cross-Entropy with Logits Loss and the Adam optimiser, respectively. 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 was selected with these beta values as they help reduce momentum-induced oscillations during training and are widely used for GANs (Apxml, 2023).

```
def train_step(x_real):
    generator.train()
    discriminator.train()
    bs = x_real.size(0)
    x_real = x_real.cuda()

    z = torch.randn(bs, 2, device = 'cuda')
    with torch.no_grad():
      x_fake = generator(z)

    d_opt.zero_grad()
    d_real_logit = discriminator(x_real)
    d_fake_logit = discriminator(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()

    z = torch.randn(bs, 2, device = 'cuda')
    g_opt.zero_grad()
    x_fake = generator(z)
    d_fake_logit = discriminator(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 4: Sine wave step function

This function iterates one step of the GAN training based on the training step by AI Monks (2021). 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. Next, 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 5: Sine Wave 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 Conditional Deep Convolution GAN: MNIST

Next, we apply a GAN to the MNIST dataset to further demonstrate its efficacy in generating synthetic data, now with images. The GAN will include a conditional architecture that will allow us to request the generation of

specified digits within the limits of the dataset (0-9). The GAN will also include a deep convolution architecture for identifying spatial correlations, useful for image training.

```
epochs = 10
img_size = 28
latent_dim = 100
channels = 1
lr = 10**-4
batch_size = 5
```

Listing 6: Parameters used for training.

```
transform = transforms.Compose(
    [transforms.Resize(img_size),
     transforms.ToTensor(),
     transforms.Normalize((0.5,), (0.5,))]
)

mnist_data = torchvision.datasets.MNIST(root='./data', train=True, download=True, transform=transform)
data_loader = torch.utils.data.DataLoader(mnist_data, batch_size=batch_size, shuffle=True)
```

Listing 7: Load and transform MNIST dataset

After loading the data, the features are resized and normalised. The datasets are split into training and testing sets, Listing 7 shows these steps.

```
class Generator(nn.Module):
  def __init__(self, latent_dim=100, n_classes=10, embedding_dim=100):
    super(Generator, self).__init__()
    self.label_conditioned_generator = nn.Sequential(
        nn.Embedding(n_classes, embedding_dim),
        nn.Linear(embedding_dim, 7 * 7)
    )

    self.latent = nn.Sequential(
        nn.Linear(latent_dim, 7 * 7 * 128),
        nn.LeakyReLU(0.2, inplace=True)
    )

    self.model = nn.Sequential(
        nn.ConvTranspose2d(in_channels=128 + 1, out_channels=128, kernel_size=4, stride=2, padding=1),
        nn.BatchNorm2d(128),
```

```
        nn.ReLU(inplace=True),
        nn.ConvTranspose2d(in_channels=128, out_channels=
    channels, kernel_size=4, stride=2, padding=1),
        nn.Tanh()
    )

  def forward(self, inputs):
        noise_vector, label = inputs
        label_output = self.label_conditioned_generator(label
    )
        label_output = label_output.view(-1, 1, 7, 7)
        latent_output = self.latent(noise_vector)
        latent_output = latent_output.view(-1, 128, 7, 7)
        concat = torch.cat((latent_output, label_output), dim
    =1)
        image = self.model(concat)
        return image

class Discriminator(nn.Module):
  def __init__(self, n_classes=10, embedding_dim=100):
    super(Discriminator, self).__init__()

    self.label_condition_disc = nn.Sequential(
        nn.Embedding(n_classes, embedding_dim),
        nn.Linear(embedding_dim, 28 * 28)
    )

    self.model = nn.Sequential(
        nn.Conv2d(in_channels=channels + 1, out_channels=64,
    kernel_size=3, stride=2, padding=1),
        nn.LeakyReLU(0.2, inplace=True),
        nn.Conv2d(in_channels=64, out_channels=128,
    kernel_size=3, stride=2, padding=1),
        nn.BatchNorm2d(128),
        nn.LeakyReLU(0.2, inplace=True),
        nn.Conv2d(in_channels=128, out_channels=1,
    kernel_size=7, stride=1, padding=0),
        nn.Flatten(),
        nn.Sigmoid()
    )

  def forward(self, inputs):
        img, label = inputs
        label_output = self.label_condition_disc(label)
        label_output = label_output.view(-1, 1, 28, 28)
```

```
        concat = torch.cat((img, label_output), dim=1)
        output = self.model(concat)
        return output
```

Listing 8: MNIST Generator and Discriminator

Listing 8 represents an extension beyond the simple GAN in Listing 2. The design is based on prior work on conditional GANs (LearnOpenCV, 2020) and deep convolutional GANs (AI Monks, 2021), both of which are essential for training and conditional generation.

```
for epoch in range(epochs):
  for i, (images, labels) in enumerate(data_loader):
    images = images.cuda()
    labels = labels.cuda()
    batch_size = images.size(0)

    opt_d.zero_grad()
    labels_r = torch.ones(batch_size, 1).cuda()
    output_r = discriminator((images, labels)).view(-1,1)
    loss_r = loss_fn(output_r, labels_r)
    loss_r.backward()

    z = torch.randn(batch_size, latent_dim).cuda()
    random_labels = torch.randint(0, 10, (batch_size,),
   device='cuda')
    images_f = generator((z, random_labels))
    label_f = torch.zeros(batch_size, 1).cuda()
    output_f = discriminator((images_f.detach(),
   random_labels)).view(-1,1)
    loss_f = loss_fn(output_f, label_f)
    loss_f.backward()
    opt_d.step()

    opt_g.zero_grad()
    z = torch.randn(batch_size, latent_dim).cuda()
    random_labels = torch.randint(0, 10, (batch_size,),
   device='cuda')
    images_f = generator((z, random_labels))
    output = discriminator((images_f, random_labels)).view
   (-1,1)
    loss_g = loss_fn(output, labels_r)
    loss_g.backward()
    opt_g.step()

    if (i+1)%100 == 0:
```

```
      print(f'Epochs [{epoch+1}/{epochs}], Batch[{i+1}/{len(data_loader)}],'
            f'D_real: {output_r.mean():.4f}, D_fake: {output_f.mean():.4f}, '
            f'Loss_D: {loss_r.item() + loss_f.item():.4f}, Loss_G: {loss_g.item():.4f}')
  with torch.no_grad():
    fixed_noise = torch.randn(10 * 5, latent_dim).cuda()
    fixed_labels = torch.arange(0, 10).repeat(5).cuda()
    samples_f = generator((fixed_noise, fixed_labels))
    samples_f = samples_f.cpu()
    show_img(torchvision.utils.make_grid(samples_f, nrow=5))
```

Listing 9: Training Loop and Step

The training loop is presented in Listing 9. Real and fake labels are balanced per batch and multiple loss components are aggregated. At the end of each training epoch, noise vectors are sampled to generate and visualise training progress.

```
def generate_digit(*digit_names):
    digit_map = {
        'zero': 0, 'one': 1, 'two': 2, 'three': 3, 'four': 4,
        'five': 5, 'six': 6, 'seven': 7, 'eight': 8, 'nine': 9
    }

    generated_images = []
    for digit_name in digit_names:

        label = torch.tensor([digit_map[digit_name]], device='cuda')
        z = torch.randn(1, latent_dim, device='cuda')

        generator.eval()
        with torch.no_grad():
            generated_image = generator((z, label)).cpu()
        generated_images.append(generated_image)

    if generated_images:
        display_grid = torch.cat(generated_images, dim=0)
        show_img(torchvision.utils.make_grid(display_grid, nrow=len(generated_images)))
```

Listing 10: Generate digits function

Listing 10 shows a utility function that generates specified digits. By providing digit names (e.g., 'one'), the trained generator produces images corresponding to those digits.

4 Results

4.1 Evaluation: Simple GAN

We first attempted to replicate a one-dimensional sine function using a simple GAN. The target function is shown in Figure 1, which illustrates a sparsely sampled sine wave.

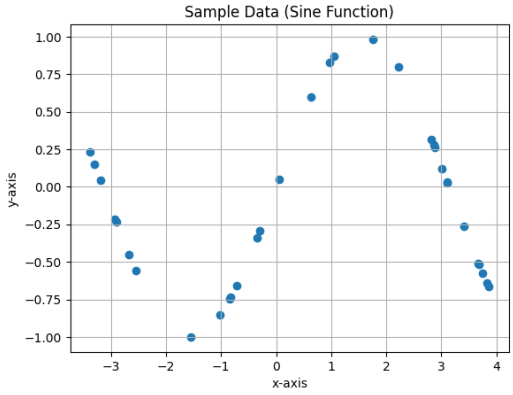

Figure 1: Sample Data: Sine Function

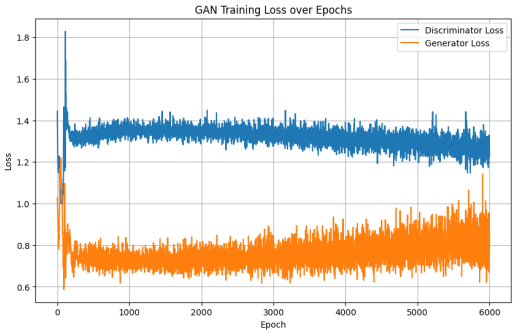

Figure 2: GAN Training Loss Curve for Sine Data

The training dynamics are shown in Figure 2. The loss gains instability over increasing epochs. This behaviour is due to the sparsity of the training data, in which the GAN may have difficulty capturing the underlying distribution of the periodic function. Figure 3 shows the comparison between the

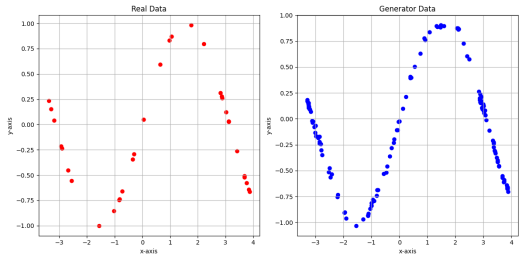

Figure 3: Original Sparse Sine Function (left) versus Generated Populated Sine Function (right)

original and the generated sine function. Although the generated curve does not perfectly reproduce the training distribution, it successfully captures the oscillatory shape of the sine wave. The size of the latent noise vector was increased, which improved the coverage of the input space, leading to a clearer approximation of the waveform. Overall, these results demonstrate that a simple GAN can reproduce periodic shapes. The sparsity of the data affected the loss stability, which remains a key limitation.

4.2 Evaluation: MNIST GAN

Next, we evaluated a conditional convolutional GAN trained on the MNIST dataset, which contains 60,000 training images. A sample of target images is shown in Figure 4.

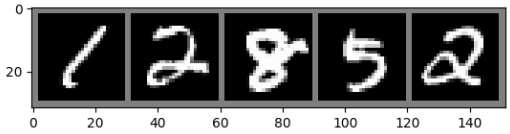

Figure 4: MNIST Random Sample: 1, 2, 8, 5, 2

Figure 5 shows the GAN output after the first training epoch. Although some digits are poorly formed, others are clearly recognisable (e.g., the zero

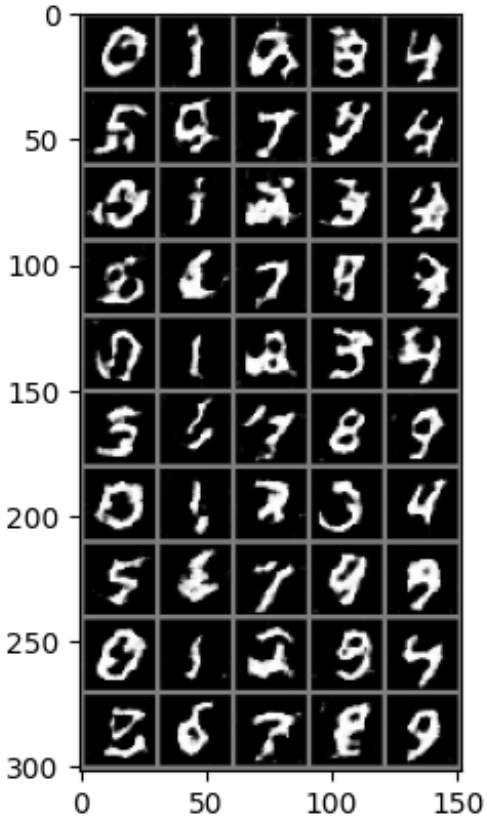

Figure 5: GAN MNIST Epoch One Progress

in the first row). This shows that the model rapidly learnt simple digit structures.

By Epoch 10, shown in Figure 6, virtually all digits are well formed. The generator effectively captured the digit structure and produced samples visually similar to the training distribution. Training completed in approximately 15 minutes.

The same digits (1, 2, 8, 5, 2) from Figure 4 were generated by the GAN shown in Figure 7. The digits are clear, and the GAN has successfully learnt to reproduce specified digits. These results indicate that the conditional convolutional GAN can effectively learn and reproduce the distribution of

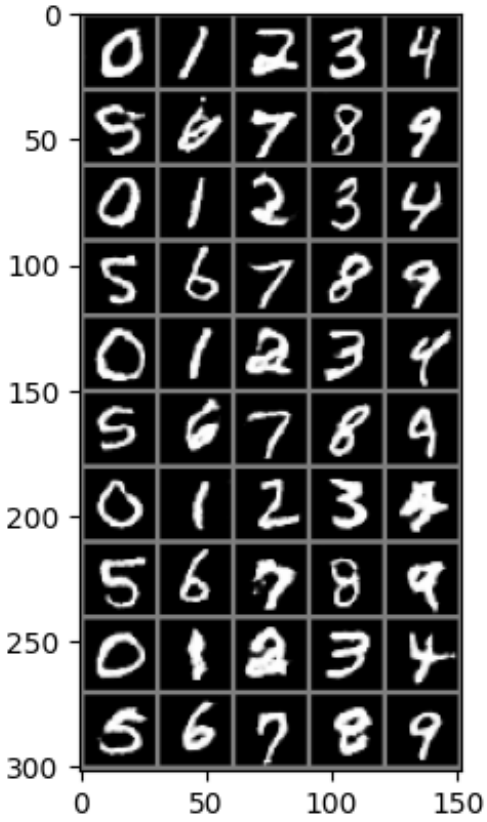

Figure 6: GAN MNIST Final Epoch Progress

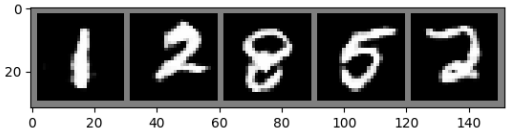

Figure 7: Generated Digits: 1, 2, 8, 5, 2

handwritten digits. In 15 minutes, the generated samples are of sufficient clarity and variability such that they are indistinguishable from real handwritten digits.

5 Discussion

The implementation of a simple GAN served to introduce the adversarial framework of GANs, rather than an evaluation of its performance. When tasked with reconstructing a sine function, the model successfully reproduced the underlying periodic structure and populated regions of sparsity. Although no quantitative error diagnostics were applied, the comparison in Figure 3 shows that the generated curve approximates the target function with reasonable fidelity. This suggests that data with clear underlying distributions can be recognised even through a minimal GAN architecture with limited precision when data is sparse.

Extending the framework to the MNIST dataset highlighted the advantages of a more sophisticated architecture. By combining a conditional and deep convolutional architecture for the GAN, the generator produced realistic digit images after only ten epochs of training (Figures 5-6). The final results in Figure 7 show that the GAN can generate digits conditionally and realistically. Although individual digit appearances differ slightly from the originals, the outputs are human recognisable and consistent with the training distribution illustrated in Figure 6.

Together, the two experiments indicate that GANs can learn both simple and complex data distributions. For small, sparse datasets, the generated results are less precise when synthesising data, whereas for larger, structured datasets, the outputs achieve a level of realism comparable to the training data. These findings were obtained using standard activation functions, binary cross-entropy loss, and Adam optimisation, with minimal hyperparameter tuning, highlighting that strong results can be achieved even without extensive architectural or training refinements. Future work should extend the experiments to additional datasets, and comparisons with alternative generative methods (e.g., diffusion models) would provide a broader insight into their relative strengths and weaknesses. Such investigations would help establish GANs not only as a demonstration of adversarial learning but also as a reliable tool for synthetic data generation in practical data science applications.

6 Conclusion

This project implemented Generative Adversarial Networks (GANs) in PyTorch on two tasks: reproducing a simple sine wave and generating handwritten digits from the MNIST dataset. The primary objective was to learn and implement GANs as an introductory deep learning framework.

The experiments demonstrated that a simple GAN can approximate the underlying structure of sparse periodic data, while a conditional convolutional GAN can produce realistic image samples with a large, structured dataset. Visual evaluation showed that the generated output retained the essential features of the training data, with fidelity improving as the size of the dataset and the architectural complexity increased.

Although limited to qualitative evaluation, these results underscore the ability of GANs to capture diverse data distributions. More rigorous evaluation and comparisons with alternative generative methods would further clarify their effectiveness. Overall, this implementation illustrates the core adversarial dynamics of GANs and provides a foundation for future work applying generative models in practical data science contexts.

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 Dependencies used

```
import torch
import torch.nn as nn
import torchvision
import torchvision.transforms as transforms
import torchvision.datasets as datasets
from torch.utils.data import DataLoader
import matplotlib.pyplot as plt
import numpy as np
import math
```

Listing 11: Dependencies used in Notebook

8 References

1. Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., & Bengio, Y. (2014). Generative adversarial nets. In *Advances in neural information processing systems* (pp. 2672–2680). Retrieved from `https://arxiv.org/pdf/1406.2661`

2. Mirza, M., & Osindero, S. (2014). Conditional generative adversarial nets. *arXiv preprint* arXiv:1411.01784. Retrieved from `https://arxiv.org/pdf/1411.01784`

3. Radford, A., Metz, L., & Chintala, S. (2016). Unsupervised representation learning with deep convolutional generative adversarial networks. In *International Conference on Learning Representations (ICLR 2016)*. Retrieved from `https://arxiv.org/pdf/1511.06434`

4. Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep learning*. MIT Press. Retrieved from `https://www.deeplearningbook.org`

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

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

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

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

9. AI Monks. (2021). *Generating GAN images from training on MNIST dataset*. Medium. Retrieved from `https://medium.com/aimonks/generating-gan-images-from-training-on-mnist-dataset-2e2ec53dfe96`

10. LearnOpenCV. (2020). *Conditional GAN (cGAN) in PyTorch and TensorFlow*. Retrieved from `https://learnopencv.com/conditional-gan-cgan-in-pytorch-and-tensorflow/`