Generating images using a Deep Convolutional GANs

Author: Kai Deng

Supervisor: Serhiy Yanchuk

A thesis submitted in partial fulfilment of the
requirements for the degree of
MSc Mathematical Modelling and Machine Learning

School of Mathematical Sciences,
University College Cork,
Ireland

September 2024

Declaration of Authorship

This report is wholly the work of the author, except where explicitly stated otherwise. The source of any material which was not created by the author has been clearly cited.

Date: 11/09/2024

Signature: Kai Deng

Acknowledgements

I would like to express my deepest gratitude to my supervisor, Dr Serhiy Yanchuk, for his invaluable guidance, continuous support, and insightful feedback throughout the course of my research. His expertise and encouragement have been instrumental in the successful completion of this thesis.

Abstract

Generative Adversarial Networks (GANs) have become a pivotal tool in the field of generative modeling, offering a novel approach to generating realistic data through the adversarial training of a generator and a discriminator. This thesis investigates the performance of GANs in generating high-quality images, with a particular focus on the challenges associated with model structure, training stability, and evaluation metrics.

In this study, I employed the Animal Faces-HQ (AFHQ) dataset, consisting of 16,130 high-resolution images, to train a standard GAN model aimed at generating realistic cat images. The research explored the impact of various architectural choices, including the use of convolutional layers versus dense layers, as well as the effects of data augmentation techniques on model performance. The experiments revealed that convolutional architectures significantly outperformed dense ones, highlighting the importance of spatial feature extraction in image generation tasks.

The study also examined the role of data augmentation in GAN training, finding that while augmentation can introduce useful variability into the training data, it may also lead to optimization challenges if not carefully applied. The final model, trained on downscaled images due to hardware limitations, was able to produce cat images, demonstrating the effectiveness of GANs in image synthesis.

The findings contribute to a deeper understanding of how GAN architecture and training strategies influence the quality of generated images, offering insights for future research and practical applications in fields requiring high-quality image generation. The study concludes by suggesting directions for further exploration, including the potential benefits of more advanced GAN variants and higher-resolution training data.

Contents

List of Figures

Chapter 1

Introduction

Generative Adversarial Networks (GANs) have emerged as a powerful class of generative models that revolutionize generative modeling by framing it as a game between two networks: a generator network that produces synthetic data from noise and a discriminator network that distinguishes between the generated data and real data [1]. These networks, introduced in 2014, have found applications in various fields, including materials science, radiology, and computer vision [2], [3], [4]. For example, CycleGAN has been effectively applied in the medical field, notably in medical imaging tasks. It has enhanced liver lesion classification through GAN-based synthetic medical image augmentation, surpassing traditional data augmentation methods in sensitivity and specificity [5]. Abdal et al. (2019) demonstrated the effectiveness of StyleGAN in tasks such as image deformation and style transfer [6]. GANs have gained significant attention in the computer vision community due to their ability to generate data without explicitly modeling the probability density function [3].

Despite these advances, training GANs remains a challenging task due to issues such as mode collapse, instability during training, and the requirement for substantial computational resources. Additionally, evaluating the performance of GANs is not straightforward, as traditional metrics such as accuracy do not adequately capture the quality or diversity of the generated data. This complexity has led to the development of various GAN architec-

tures and training techniques aimed at improving the stability and quality of generated images.

The purpose of this thesis is to contribute to a better understanding of the factors influencing the performance of GANs, particularly in the context of generating realistic images. To achieve this, I will utilize the Animal Faces-HQ (AFHQ) dataset, which comprises 16,130 high-quality images with a resolution of 512×512 pixels, to train a standard GAN model specifically designed for generating realistic cat pictures. The high resolution of these images presents both an opportunity and a challenge, as it requires careful consideration of model structure and training strategies to avoid issues such as overfitting or insufficient model capacity.

This thesis is structured as follows: Chapter 2 provides an overview of historical models of image generation, including Deep Boltzmann Machines, Variational Autoencoders, and Noise Contrastive Estimation. Chapter 3 discusses the theoretical background of GANs, focusing on their objective functions, the training process, and methods for evaluating their performance. Chapter 4 presents the experimental work conducted, including model selection, architecture exploration, and the impact of data augmentation, followed by the application of the GAN model to the AFHQ dataset. Finally, Chapter 5 discusses the results, highlights the implications of the findings, and suggests directions for future research.

Chapter 2

Historical Models of Image Generation

In this chapter, I will explore three key generative modeling techniques: Noise Contrastive Estimation (NCE), Variational Autoencoders (VAEs), and Diffusion Models. These models represent significant milestones in the development of generative modeling approaches, each with its unique advantages and limitations.

2.1 Noise Contrastive Estimation

2.1.1 Principles of NCE

Noise Contrastive Estimation (NCE) was introduced in 2010 by Gutmann and Hyvärinen as a method for estimating parameters of unnormalized probabilistic models. It offers an alternative to Maximum Likelihood Estimation (MLE), which can be computationally expensive for large-scale models. NCE reframes the complex task of normalizing probability distributions into a more tractable binary classification problem [7].

In traditional MLE, training models for unnormalized probability distributions involves calculating the partition function, which is often computationally intractable. NCE avoids this issue by introducing noise samples drawn from a known distribution. The model is then trained to discriminate

between real data samples and noise samples. Specifically, the model assigns higher probabilities to real data and lower probabilities to noise, indirectly learning the data distribution without requiring normalization [8].

2.1.1.1 NCE Architecture and Process

At the core of Noise Contrastive Estimation is the idea of reframing MLE as a binary classification task. In traditional MLE, training models for unnormalized probability distributions involves calculating the partition function, which is computationally intractable in large-scale models. NCE avoids this issue by introducing noise samples drawn from a known distribution. The model is then trained to discriminate between real data samples and noise samples. Specifically, the model assigns higher probabilities to real data and lower probabilities to noise, thus indirectly learning the data distribution without requiring normalization [8].

The objective function in NCE is derived by reformulating the likelihood of a data point x as the probability that it is real, rather than noise. Given a dataset D with real samples x_i and noise samples x_j, the probability $P(y = 1|x)$, where $y = 1$ indicates a real sample and $y = 0$ indicates a noise sample, is given by:

$$P(y = 1|x) = \frac{p_\theta(x)}{p_\theta(x) + kp_{\text{noise}}(x)} \tag{2.1}$$

Where:

- $p_\theta(x)$ is the unnormalized model probability assigned to the data sample x.
- $p_{\text{noise}}(x)$ is the known probability of the noise sample.
- k is the ratio of noise samples to real samples.

The corresponding probability for a noise sample is given by:

$$P(y = 0|x) = \frac{kp_{\text{noise}}(x)}{p_\theta(x) + kp_{\text{noise}}(x)} \tag{2.2}$$

The NCE objective then maximizes the log-probabilities of real data and noise data being correctly classified:

$$\mathcal{L}_{NCE} = \sum_{i=1}^{N} \left[\log P(y = 1|x_i) + \sum_{j=1}^{k} \log P(y = 0|x_j) \right] \tag{2.3}$$

This reformulation circumvents the need to compute the partition function, which makes NCE particularly effective for large-scale models.

As shown in Figure 2.1, the architecture of a typical NCE model includes an input layer, hidden layer, and output layer. The input layer takes in data samples (both real and noise), while the hidden layer learns a representation of these samples. The output layer, modeled as a binary classification task, outputs is a probabilities indicating whether a given sample is real or noise.

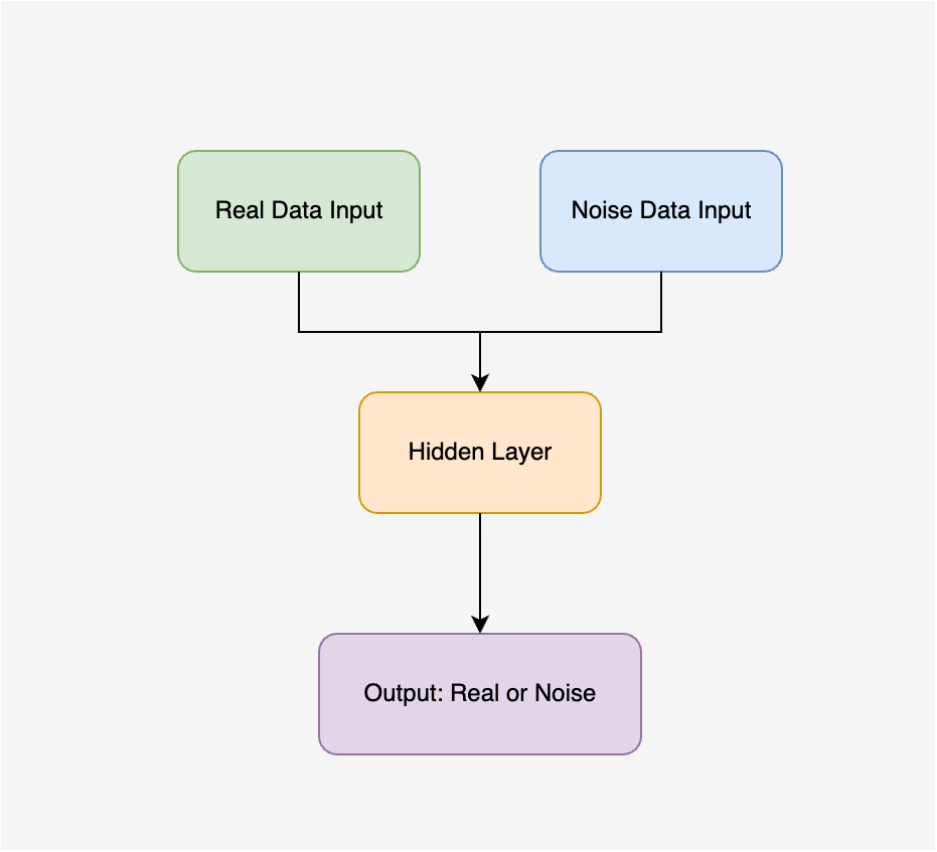


Figure 2.1: NCE architecture showing the input, hidden, and output layers.

- **Data and Noise Samples**: NCE introduces noise samples from a known distribution to compare with the actual data. These noise samples serve as negative examples in the binary classification task, while real data serves as positive examples [7].

- **Binary Classification**: The task of differentiating between real and noise samples is formalized as a binary classification problem, which can be optimized using standard logistic regression techniques [9].

By transforming the estimation problem in this way, NCE simplifies the computation and circumvents the need for calculating the partition function, which is often the bottleneck in MLE for large-scale models.

2.1.2 Comparison with GANs

Noise Contrastive Estimation (NCE) and Generative Adversarial Networks (GANs) are both prominent techniques used in machine learning, but they differ significantly in their approach to model training and estimation.

- **Training Stability**: One of the primary advantages of NCE over GANs is its stability during training. GANs often face challenges such as mode collapse and unstable training dynamics due to the adversarial nature of the training process, where the generator and discriminator are pitted against each other. In contrast, NCE transforms the model training process into a binary classification problem, which is generally more stable and can be optimized using standard logistic regression techniques [9].

- **Computational Efficiency**: GANs require training two models (generator and discriminator) simultaneously, which can lead to higher computational costs, particularly when tuning their interactions for optimal performance. NCE, on the other hand, simplifies the problem by reducing it to a comparison between real data and noise samples. This avoids the adversarial framework and allows for more efficient training, especially in large-scale models [10].

- **Handling Unnormalized Models**: NCE is particularly effective for training unnormalized probabilistic models, where the partition function is difficult or impossible to compute directly. GANs, however, are typically used for generating realistic data samples, and while they do not require explicit normalization, they do not address the estimation of unnormalized probabilistic models as NCE does [11].

- **Model Interpretability**: In NCE, the model explicitly learns to estimate the probability of data samples relative to noise, which can provide insights into the underlying data distribution. In contrast, GANs focus primarily on generating realistic samples, and their internal workings (particularly the latent space) can be less interpretable compared to NCE-based models.

- **Use in Language Models and Word Embeddings**: NCE is particularly advantageous in applications such as word embeddings and large-scale language models, where normalizing the likelihood function over a large vocabulary is computationally prohibitive. While GANs have been explored in text generation tasks, NCE's efficiency in handling large vocabulary sizes makes it a more suitable choice for these types of problems [11].

2.1.3 Applications of NCE

The utility of Noise Contrastive Estimation extends to various machine learning tasks. It is particularly effective in scenarios involving large datasets and unnormalized probabilistic models:

- **Word Embeddings**: NCE is widely used in training word embeddings in natural language processing, where the vocabulary size makes exact normalization computationally infeasible.

- **Language Models**: In addition to word embeddings, NCE has been applied to train large-scale language models where traditional likelihood-based methods may become computationally prohibitive.

- **Energy-Based Models**: As mentioned earlier, NCE is effective in training energy-based models, where the partition function is challenging to compute directly.

By leveraging NCE, models can scale efficiently, making it a useful tool in a wide range of applications, from natural language processing to computer vision.

2.1.4 Limitations of NCE

While Noise Contrastive Estimation has proven to be an effective technique, it is not without limitations. One of the primary challenges lies in selecting an appropriate noise distribution. The choice of noise distribution is critical to the model's success; a poorly chosen noise distribution can lead to inaccurate parameter estimates and slower convergence during training [8].

- **Noise Distribution Sensitivity**: NCE relies heavily on the assumption that the noise distribution is sufficiently different from the true data distribution. If the noise distribution is not carefully selected, the model may fail to accurately distinguish between real and noise samples [8].

- **Complex Data Distributions**: NCE may struggle with highly complex data distributions, especially in cases where defining an appropriate noise distribution is difficult [8].

Despite these limitations, NCE remains a popular technique for training large-scale models, especially in cases where traditional MLE is impractical due to the computational cost of normalizing the probability distribution.

2.2 Variational Autoencoders (VAEs)

Variational Autoencoders (VAEs) were introduced in 2013 and are one of the earliest forms of generative models. VAEs aim to model the underlying distribution of data by learning a compressed representation, or latent vector, z, of the input data x.

2.2.1 VAE Architecture and Training

2.2.1.1 VAE Structure

The architecture of VAEs consists of an encoder that maps input data into a latent space, where the latent variables are typically assumed to follow a Gaussian distribution. This assumption simplifies the learning process, as it allows for the use of techniques such as the reparameterization trick, which enables backpropagation through stochastic layers [12]. The decoder then reconstructs the input data from the latent variables, ensuring that the model captures the essential features of the data distribution.

As shown in Figure 2.2, the VAE architecture consists of two main components:

- **Encoder**: The encoder takes the input data x and compresses it into a latent representation z. The latent variables are sampled from a Gaussian distribution, which simplifies optimization.
- **Decoder**: The decoder takes the latent variable z and attempts to reconstruct the original data x', aiming to generate data that resembles the input.

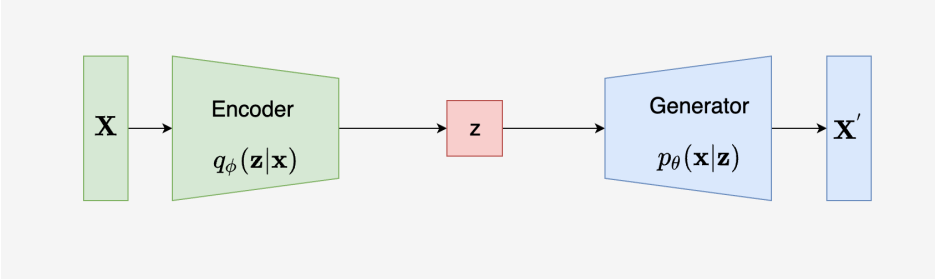


Figure 2.2: VAE structure showing the encoder and decoder processes.

2.2.1.2 VAE Loss Function

The loss function of a VAE combines reconstruction loss with a Kullback-Leibler (KL) divergence term, which regularizes the latent space and encourages the model to learn a smooth and continuous representation [13].

VAEs are trained to maximize the variational lower bound, which balances the accuracy of the reconstruction and ensures that the latent space is smooth and continuous. This allows the model to generate new data by sampling from the learned latent space and reconstructing the data using the decoder.

The loss function of VAE is as follows:

$$\mathcal{L} = \mathbb{E}_{q_\phi(z|x)}[\log p_\theta(x|z)] - D_{KL}(q_\phi(z|x)\|p(z)) \tag{2.4}$$

Where:

- $\mathcal{L}$: The total loss that the model aims to minimize.
- $\mathbb{E}_{q_\phi(z|x)}[\log p_\theta(x|z)]$: The expected log-likelihood of reconstructing the data x' from the latent variable z. The distribution $q_\phi(z|x)$ represents the encoder, while $p_\theta(x|z)$ represents the decoder's attempt to reconstruct the input data.
- $D_{KL}(q_\phi(z|x)\|p(z))$: The Kullback-Leibler (KL) divergence, which measures the difference between the encoder's latent distribution $q_\phi(z|x)$ and the prior distribution $p(z)$, typically assumed to be Gaussian.

2.2.2 Comparison with GANs

In comparison to Generative Adversarial Networks (GANs), VAEs offer several key advantages. One of the primary benefits of VAEs is their simpler and more stable training process. GANs involve training two models simultaneously—a generator and a discriminator—which can result in unstable convergence and issues like mode collapse, where the generator fails to capture the diversity of the data. In contrast, VAEs have a single objective function that combines reconstruction loss and KL divergence, making the optimization process more straightforward [12].

Furthermore, the structured latent space of VAEs allows for meaningful interpolation between latent variables, enabling smooth transitions between generated data points. This makes VAEs particularly useful for tasks such as

image generation, anomaly detection, and data imputation, where the ability to smoothly interpolate between data points is crucial [14][15]. GANs, on the other hand, do not explicitly model the latent space, which can limit their interpretability and ability to interpolate between generated samples.

However, GANs are known for producing sharper and more realistic images compared to VAEs, especially in high-resolution image generation tasks. This is because the adversarial loss in GANs encourages the generator to produce outputs that closely resemble real data, whereas VAEs tend to produce blurrier images due to the use of a Gaussian prior in the latent space [16]. Despite this, VAEs are more flexible and scalable, as they can be trained with standard gradient descent methods and do not require the complex adversarial setup that GANs use.

2.2.3 Applications of VAEs

The structured latent space of VAEs not only allows for efficient sampling but also supports various applications. These include:

- **Image Generation**: VAEs are capable of generating new images by sampling from the latent space and decoding them to the image space.
- **Anomaly Detection**: VAEs can identify outliers by measuring the reconstruction error, where high reconstruction loss may indicate anomalous data points.
- **Data Imputation**: VAEs can be used to fill in missing data by sampling from the latent space and reconstructing the missing portions of the data.

2.2.4 Limitations of VAEs

One notable limitation of VAEs is their difficulty in modeling discrete data. Since VAEs rely on backpropagation through continuous latent variables, handling discrete data types effectively poses a challenge [17]. Additionally, the Gaussian assumption in the latent space can sometimes lead to less sharp

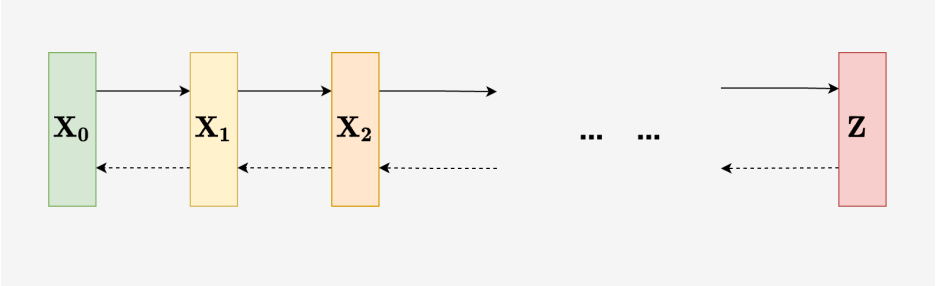


Figure 2.3: Diffusion model structure showing the forward and reverse processes.

or less diverse outputs compared to GANs. Despite these limitations, VAEs remain a powerful tool for representing high-dimensional complex data by learning a low-dimensional latent space in an unsupervised manner [18].

2.3 Diffusion Models

Diffusion models are the most recent advancement in generative models, first introduced in the early 2020s. These models take a different approach by progressively adding noise to data and then learning to reverse the process, effectively "denoising" the noisy data back into its original form.

The process involves two key steps, as shown in Figure 2.3:

- **Forward Process**: Gradually adds Gaussian noise to data x_0, creating noisy versions of the data $x_1, x_2, \ldots, x_T$. Each step increases the level of noise, eventually leading to a completely noisy version z.
- **Reverse Process**: The model learns to reverse the noise addition process, starting from the fully noisy version z, and progressively denoising it to recover data that resembles the original input x_0.

2.3.1 Forward Process

In the forward process, starting from the original data x_0, Gaussian noise is added step-by-step to generate increasingly noisy versions of the data. Mathematically, the forward process is described as:

$$q(x_t|x_{t-1}) = \mathcal{N}(x_t; \sqrt{\alpha_t}x_{t-1}, \beta_t I) \tag{2.5}$$

Where:

- x_0, x_1, x_2, ..., x_T: The sequence of noisy data at each time step.
- α_t: The scaling factor applied to the data at each step.
- β_t: The variance of the Gaussian noise added at each step t.
- $\mathcal{N}(x; \mu, \sigma^2)$: A Gaussian distribution with mean μ and variance σ^2.

The process continues until we reach the final state z, which is almost entirely noise.

2.3.2 Reverse Process

Once the noisy data is generated, the reverse process begins. The model learns to reverse the noise addition process to gradually recover the original data. The reverse process is defined as:

$$p_\theta(x_{t-1}|x_t) = \mathcal{N}(x_{t-1}; \mu_\theta(x_t, t), \Sigma_\theta(x_t, t)) \tag{2.6}$$

Where:

- x_t: The noisy data at time step t.
- $\mu_\theta(x_t, t)$: The model's predicted mean at step t, parameterized by θ.
- $\Sigma_\theta(x_t, t)$: The model's predicted variance at step t, parameterized by θ.
- $\mathcal{N}(x; \mu, \sigma^2)$: A Gaussian distribution with mean μ and variance σ^2.

The reverse process progressively reduces the noise added in the forward process, resulting in data that closely resembles the original input.

2.3.3 Loss Function

The training objective for diffusion models is to minimize the difference between the real data distribution and the distribution of generated data across all time steps:

$$L = \sum_{t=1}^{T} \mathbb{E}_{x_0,\epsilon}[\|\epsilon - \epsilon_\theta(x_t, t)\|^2] \tag{2.7}$$

Where:

- L: The loss function to be minimized.
- T: The total number of time steps in the diffusion process.
- x_0: The original data sample.
- x_t: The data at time step t, after adding noise.
- ϵ: The noise added to the data at each step.
- $\epsilon_\theta(x_t, t)$: The model's estimate of the noise at time step t.

2.3.4 Comparison with GANs

Diffusion models differ significantly from Generative Adversarial Networks (GANs) in both training dynamics and generated data quality. One of the key advantages of diffusion models is their training stability. While GANs often suffer from issues like mode collapse and unstable training due to the adversarial nature of training two competing networks (generator and discriminator), diffusion models employ a simpler training process by learning to reverse the noise addition without requiring a discriminator [**?**].

Furthermore, diffusion models can generate high-quality and diverse samples by progressively denoising data, allowing for more controlled sample generation. In contrast, GANs tend to generate sharper but sometimes less diverse images due to the tendency of the generator to overfit to certain modes of the data distribution. Diffusion models, by their nature, ensure

a gradual reconstruction from noise, which can avoid some of the issues of GANs [**?**].

However, GANs are often preferred for tasks requiring very sharp and realistic images, especially in high-resolution image synthesis. Diffusion models, while generally producing realistic images, may not always match the sharpness achieved by GANs in certain domains, particularly in tasks like photorealistic image generation.

2.3.5 Applications of Diffusion Models

Diffusion models have found a wide range of applications, especially in fields where stability and quality of generation are important. Some notable applications include:

- **Image Generation**: Diffusion models have proven effective in generating photorealistic images, similar to GANs, but with more stable training dynamics.
- **Text-to-Image Generation**: Recent advancements have applied diffusion models to text-to-image synthesis, where a text prompt is converted into a corresponding image. These models can produce diverse outputs based on input descriptions.
- **Speech Synthesis**: Diffusion models have also been applied to generating high-quality audio data. For example, they are used in text-to-speech systems to generate realistic human speech [**?**].
- **Anomaly Detection**: Similar to VAEs, diffusion models can be used to detect anomalies by evaluating how well a noisy sample can be denoised. Poor reconstructions may indicate that the input is anomalous or different from the training data.

2.3.6 Limitations of Diffusion Models

Despite their advantages, diffusion models have several limitations:

- **Computationally Intensive**: One of the primary drawbacks of diffusion models is that they are computationally expensive. The iterative process of adding and removing noise requires significantly more computation compared to GANs, which generate data in a single forward pass through the generator [**?**].

- **Generation Speed**: Diffusion models are slower than GANs in terms of sample generation. Since the process involves multiple time steps (often hundreds or even thousands), generating a single sample can take much longer compared to GANs, which generate samples almost instantaneously.

- **Sample Sharpness**: Although diffusion models tend to produce more diverse outputs compared to GANs, the sharpness of the generated images may not always match the high-quality, photorealistic outputs that GANs can achieve, especially in tasks that require very fine details.

Chapter 3

Theoretical Background

This chapter covers the theoretical foundations of Generative Adversarial Networks (GANs), including their objective function, the training dynamics between the generator and discriminator, and the use of metrics like Fréchet Inception Distance (FID) to evaluate performance.

3.1 Objective Function of GAN

Generative Adversarial Networks (GANs) consist of a generator G and a discriminator D, both implemented using artificial neural networks. The parametrization of GANs involves defining the network structure of these components and initializing their weights and biases [19]. The success of GANs relies on balancing the training of these two networks, where the G aims to produce samples $G(z)$ that mimic real data distributions $p_{data}(x)$ by input noise z, while the D learns to differentiate between real and generated samples [20]. The training process involves iteratively updating the weights and biases of the networks through adversarial training, where the generator tries to deceive the discriminator, and the discriminator aims to accurately classify samples [21]. The objective function for GAN has the following form:

$$V(D,G) = \mathbb{E}_{x\sim p_{data}(x)}[\log D(x)] + \mathbb{E}_{z\sim p_z(z)}[\log(1 - D(G(z)))]. \tag{3.1}$$

where:

- $z \sim p_z(z)$: The distribution from which the noise z is sampled. The purpose of using a noise distribution is to provide random input for the generator, which transforms this noise into more complex, high-dimensional data.

  Example: If we assume $p_z(z) = \mathcal{N}(0, 1)$, then z could be a vector of random values sampled independently from $\mathcal{N}(0, 1)$. The generator would then take this random vector z and map it to a synthetic image, such as a cat image, that resembles data from the real world.

- $x \sim p_{data}(x)$: A sample x drawn from the true data distribution $p_{data}(x)$.

- $D(G(z))$: The output of the discriminator for the data $G(z)$ generated by the generator. It indicates the probability that the discriminator believes that the generated data comes from the real data distribution.

- $D(x)$: The output of the discriminator for the real data x. It indicates the probability that the discriminator believes that the real data x comes from the real data distribution.

- $\mathbb{E}_{x \sim p_{data}(x)}[\log D(x)]$: The average value of $\log D(x)$ for all samples x from the true data distribution $p_{data}(x)$.

- $\mathbb{E}_{z \sim p_z(z)}[\log(1 - D(G(z)))]$: The average value of $\log(1 - D(G(z)))$ for all samples z from the noise distribution $p_z(z)$.

It describes the game process between the Generator (G) and the Discriminator (D). Specifically, the formula defines a minimax game between the Discriminator and the Generator.

The following formulas:

$$\max_D V(D, G) = \mathbb{E}_{x \sim p_{data}(x)}[\log D(x)] + \mathbb{E}_{z \sim p_z(z)}[\log(1 - D(G(z)))]. \quad (3.2)$$

$$\min_G V(G) = \mathbb{E}_{z \sim p_z(z)}[\log(1 - D(G(z)))]. \tag{3.3}$$

explains the continuous balance between the generator and discriminator. As the generator improves its ability to create more realistic data, the discriminator adjusts to better distinguish real data from generated data. This interaction drives the GAN model to produce outputs that resemble real data more closely.

The formula (3.2) explains the goal of the discriminator, which is to maximize the value of $V(D, G)$. The first component measures how likely the discriminator is to accurately identify real data, while the second component reflects the discriminator's ability to correctly recognize generated data as fake. Conversely, the generator's objective is to minimize $V(D, G)$ by producing data that makes the second term larger, aiming to trick the discriminator into believing the generated data is genuine.

The formula (3.3) shows the generator tries to minimize $V(G)$. This means that the generator tries to generate realistic data $G(z)$ so that the discriminator cannot distinguish them from real data.

In real training process, the formula (3.3) will replace by (3.4):

$$\min_G V(D, G) = \mathbb{E}_{z \sim p_z(z)}[-\log(D(G(z)))]. \tag{3.4}$$

The reason for replacing $\log(1-D(G(z)))$ with $-\log(D(G(z)))$ in practice is to address the vanishing gradient problem. As seen in the figure 3.1, $\log(1 - D)$ (blue curve) rapidly decays towards negative infinity when D approaches 1, resulting in very small gradients for the generator when the discriminator is strong. This significantly slows down the generator's learning process. In contrast, $-\log(D)$ (red curve) provides larger gradients even when D is close to 0, allowing the generator to continue learning effectively even against a strong discriminator. Therefore, the modified formula ensures that the generator receives significant gradient updates and avoids stagnation during training.

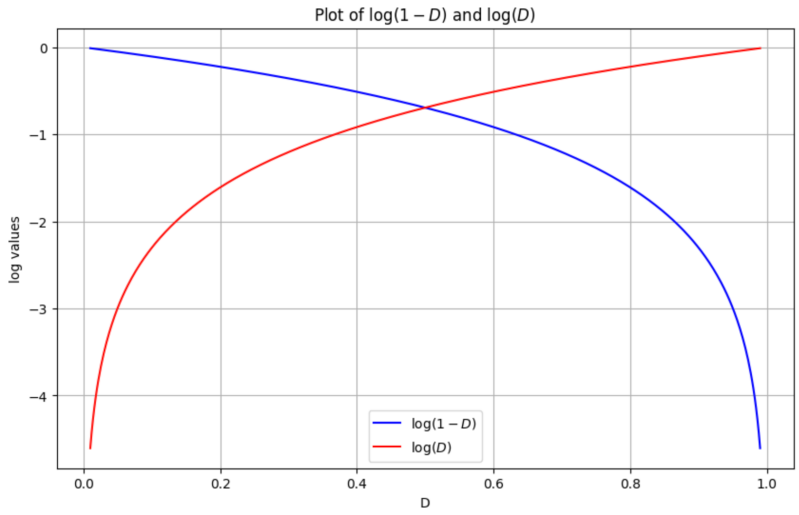


Figure 3.1: Comparison of $\log(1 - D)$ and $-\log(D)$.

The GAN network parameters play a crucial role in determining the quality and diversity of the generated samples $G(x)$ [19]. The optimization process in GANs is a min-max game. The generator aims to minimize the loss function, attempting to generate data that can fool the discriminator, while the discriminator aims to maximize its ability to correctly classify real and generated data. This min-max dynamic is critical for balancing the training of both networks. [20]. Maintaining this balance during training is essential for achieving high-quality sample generation [19]. The D's role is to provide feedback to the generator by acting as a critic, guiding the G towards producing more realistic samples [21].

3.2 Training Process of GAN

The training process of Generative Adversarial Networks (GANs) is inherently adversarial. The generator endeavors to create increasingly realistic fake samples to deceive the discriminator, while the discriminator seeks to more accurately distinguish between real and fake samples. To understand how this process works, in the following sections, I will describe the GAN training process from the perspectives of data distribution and mathematical formulation.

3.2.1 Distribution Changes During GAN Training

A simaple diagram show how distribution change in GAN training.

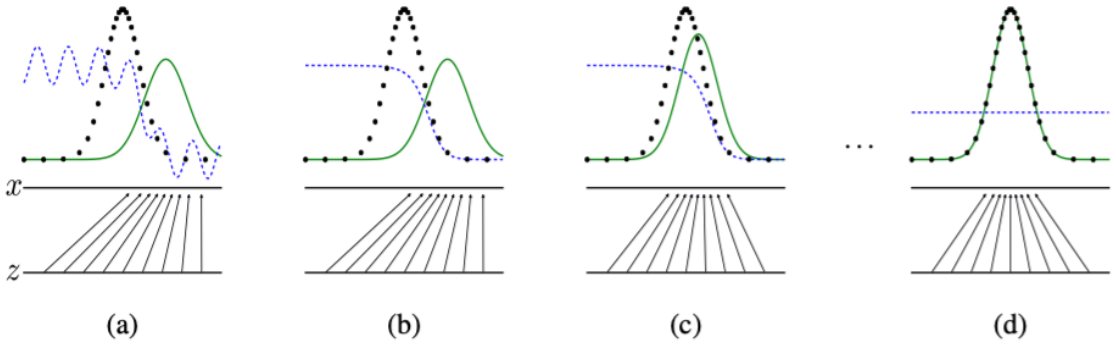


Figure 3.2: Diagram of GAN's training process. The green curve represents the distribution of generated samples. Initially, the generated samples may differ significantly from the real samples. As training progresses, the generated samples' distribution gradually approaches the real samples. The black dots represent the distribution of real samples, which remains unchanged throughout the training process and represents the target distribution. The blue dashed line represents the discriminator's output probability distribution. At the beginning of training, the discriminator can easily distinguish between real and generated data, resulting in a strong classification boundary. As training progresses and the generated data becomes more realistic, the discriminator's ability to differentiate between the two distributions weakens. Eventually, the discriminator's output approaches 0.5, indicating it can no longer effectively distinguish between real and generated data. The lines labeled x and z below represent the distribution of samples in the latent space. During GAN training, samples from the latent space z are mapped to the data space x through the generator.

Source: [22]

During the training process of the GAN model, the generator and the discriminator fight against each other. The generator tries to produce realistic samples to fool the discriminator, while the discriminator strives to distinguish real samples from generated samples. Initially, the generator generates samples with poor quality and large errors, similar to the blue sine wave in Figure (a). However, as training proceeds and the generator continues to improve, the error in generated samples gradually decreases, similar

to the blue lines in figures (b) and (c) becoming flat. At the same time, the distribution of real samples and generated samples gradually becomes consistent, and finally reaches the optimal state in Figure (d). At this moment $p_{data}(x) = p_g(x)$, the samples generated by the generator are almost the same as the real samples, and the error is minimized. During the entire process, the sample distribution gradually converges from the initial noise and discrete state to a state that highly coincides with the real distribution, reflecting a significant improvement in the quality of the generated samples.

3.2.2 Mathematical Formulation Changes During GAN Training

1. Problem Setup

In a Generative Adversarial Network (GAN), the objective is to train a generator G and a discriminator D to generate data that resembles the real data distribution. The objective function to be maximized is given by:

$$V(D, G) = \mathbb{E}_{x \sim p_{\text{data}}(x)}[\log D(x)] + \mathbb{E}_{z \sim p_z(z)}[\log(1 - D(G(z)))]. \tag{3.5}$$

2. Rewriting the Objective Function

First, the objective function is rewritten in integral form:

$$V(D, G) = \int p_{\text{data}}(x) \log D(x)\, dx + \int p_z(z) \log(1 - D(G(z)))\, dz. \tag{3.6}$$

By changing variables $x' = G(z)$, the generated data is represented as x', which corresponds to samples produced by the generator G. This allows the second term to be rewritten as an integral over the generated data distribution $p_g(x)$. The integral now reflects the contribution of the generated samples to the objective function.

$$\int p_g(x) \log(1 - D(x))\, dx. \tag{3.7}$$

Thus, the objective function becomes:

$$V(D,G) = \int \left[p_{\text{data}}(x) \log D(x) + p_g(x) \log(1 - D(x)) \right] dx. \tag{3.8}$$

3. Deriving the Optimal Discriminator

To find the optimal discriminator D^*, it needs to take the derivative of the objective function with respect to $D(x)$ and set it to zero.

Let:

$$f(D(x)) = p_{\text{data}}(x) \log D(x) + p_g(x) \log(1 - D(x)). \tag{3.9}$$

Taking the derivative with respect to $D(x)$:

$$\frac{d}{dD(x)} f(D(x)) = \frac{p_{\text{data}}(x)}{D(x)} - \frac{p_g(x)}{1 - D(x)}. \tag{3.10}$$

Setting the derivative to zero:

$$\frac{p_{\text{data}}(x)}{D(x)} = \frac{p_g(x)}{1 - D(x)} \tag{3.11}$$

Solving equation (3.11):

$$D(x) = \frac{p_{\text{data}}(x)}{p_{\text{data}}(x) + p_g(x)}. \tag{3.12}$$

4. Optimal Discriminator Formula

Therefore, the optimal discriminator D^* is given by:

$$D^*(x) = \frac{p_{\text{data}}(x)}{p_{\text{data}}(x) + p_g(x)}. \tag{3.13}$$

- When $p_{\text{data}}(x)$ is much larger than $p_g(x)$, $D^*(x) \approx 1$, indicating that the data point is almost certainly from the real data.
- When $p_{\text{data}}(x)$ is much smaller than $p_g(x)$, $D^*(x) \approx 0$, indicating that the data point is almost certainly from the generated data.

- When $p_{\text{data}}(x)$ is close to $p_g(x)$, $D^*(x) \approx 0.5$, indicating that the discriminator cannot confidently determine whether the data point is real or generated, giving each a 50% probability.

5. Verifying the Optimal Discriminator

To verify that this D^* maximizes the objective function, substitute D^* back into the objective function:

$$V(D^*, G) = \int \left[p_{\text{data}}(x) \log \left(\frac{p_{\text{data}}(x)}{p_{\text{data}}(x) + p_g(x)} \right) + p_g(x) \log \left(1 - \frac{p_{\text{data}}(x)}{p_{\text{data}}(x) + p_g(x)} \right) \right] dx. \tag{3.14}$$

Since:

$$1 - D^*(x) = 1 - \frac{p_{\text{data}}(x)}{p_{\text{data}}(x) + p_g(x)} = \frac{p_g(x)}{p_{\text{data}}(x) + p_g(x)}, \tag{3.15}$$

substituting this in:

$$V(D^*, G) = \int \left[p_{\text{data}}(x) \log \left(\frac{p_{\text{data}}(x)}{p_{\text{data}}(x) + p_g(x)} \right) + p_g(x) \log \left(\frac{p_g(x)}{p_{\text{data}}(x) + p_g(x)} \right) \right] dx. \tag{3.16}$$

This objective function represents the negative of the cross-entropy, which is maximized when $D(x) = D^*(x)$. When $D(x) = 0.5$, the discriminator cannot distinguish between real and generated data, indicating that the generator has produced samples that closely resemble the real data. Maximizing the negative cross-entropy aligns the generated data distribution with the real data distribution.

6. Conclusion

Through the above derivation, it has shown that given the generator G, the optimal form of the discriminator D is:

$$D^*(x) = \frac{p_{\text{data}}(x)}{p_{\text{data}}(x) + p_g(x)}. \tag{3.17}$$

This demonstrates that the optimal discriminator D^* outputs the probability that the input data comes from the real data distribution. This formula

provides a theoretical foundation for training GANs, guiding the updates to the generator G so that its generated data gradually approaches the real data distribution.

3.3 Evaluating GAN Performance

Fréchet Inception Distance (FID) is a metric commonly used in Generative Adversarial Network (GAN) models to quantify the dissimilarity between two image distributions [23]. It measures the distance between the distributions of real images and generated images, providing a numerical assessment of the quality of generated images. FID has gained prominence in evaluating the performance of GANs due to its ability to capture both the quality and diversity of generated images [24]. The following is the objective function for FID:

$$\text{FID} = ||\mu_r - \mu_g||^2 + \text{Tr}(\Sigma_r + \Sigma_g - 2(\Sigma_r\Sigma_g)^{1/2}) \tag{3.18}$$

$$\mu_r = \frac{1}{N}\sum_{i=1}^{N} f(x_i), \quad \Sigma_r = \frac{1}{N}\sum_{i=1}^{N}(f(x_i) - \mu_r)(f(x_i) - \mu_r)^T \tag{3.19}$$

$$\mu_g = \frac{1}{M}\sum_{i=1}^{M} f(G(z_i)), \quad \Sigma_g = \frac{1}{M}\sum_{i=1}^{M}(f(G(z_i)) - \mu_g)(f(G(z_i)) - \mu_g)^T \tag{3.20}$$

where:

- μ_r and μ_g: The feature means of the real and generated images, respectively.
- Σ_r and Σ_g: The feature covariance matrices of the real and generated images, respectively.
- Tr: The trace (the sum of the diagonal elements of the matrix).

- f: The feature extraction function, which extracts feature vectors from images using the Inception network. These feature vectors are used to compute the mean and covariance for both real and generated images.

A lower FID value indicates that the distribution of the generated images is closer to that of real images, reflecting higher quality and diversity in the generated images [25]. Specifically, a FID value below 10 is considered to represent very high-quality generated images, while values between 10 and 50 indicate good quality, and values above 50 suggest average or poor quality [25].

3.4 Limitations of Accuracy in Evaluating GANs

Accuracy is not suitable for evaluating GANs because accuracy is an indicator of classification tasks, which is used to measure the prediction accuracy of the model in classification tasks, and cannot measure the quality and diversity of generated data. In generation tasks, there is no clear "correct answer" and the generated data has no "real label", so it is impossible to directly compare the correspondence between the generated data and a real sample.

The following is a result for a standard GANs model with high accuracy but generate low quality images.

```
1900 [D loss: 9.02800047697383e-06 | D accuracy: 100.0] [G loss: 0.0005307616665959358] [FID: -1.49196970922936e+87]
2/2 [==============================] - 0s 28ms/step
1901 [D loss: 7.511995590903098e-06 | D accuracy: 100.0] [G loss: 0.0006589822005480528] [Epoch time: 0.51 seconds]
2/2 [==============================] - 0s 29ms/step
1902 [D loss: 9.099765065911924e-06 | D accuracy: 100.0] [G loss: 0.0008528590551577508] [Epoch time: 0.48 seconds]
2/2 [==============================] - 0s 29ms/step
1903 [D loss: 7.7156341831141 62e-06 | D accuracy: 100.0] [G loss: 0.0008156942203640938] [Epoch time: 0.48 seconds]
2/2 [==============================] - 0s 29ms/step
1904 [D loss: 7.386817742371932e-06 | D accuracy: 100.0] [G loss: 0.0005978870904073119] [Epoch time: 0.49 seconds]
2/2 [==============================] - 0s 29ms/step
1905 [D loss: 1.5066923879203387e-05 | D accuracy: 100.0] [G loss: 0.0014342099893838167] [Epoch time: 0.48 seconds]
2/2 [==============================] - 0s 29ms/step
1906 [D loss: 1.9851730939990375e-05 | D accuracy: 100.0] [G loss: 0.0007973920437507331] [Epoch time: 0.49 seconds]
2/2 [==============================] - 0s 29ms/step
1907 [D loss: 1.692915657258709e-05 | D accuracy: 100.0] [G loss: 0.0009673223830759525] [Epoch time: 0.49 seconds]
2/2 [==============================] - 0s 29ms/step
1908 [D loss: 1.5433080079674255e-05 | D accuracy: 100.0] [G loss: 0.0010331417433917522] [Epoch time: 0.49 seconds]
2/2 [==============================] - 0s 29ms/step
1909 [D loss: 3.74487547105673e-06 | D accuracy: 100.0] [G loss: 0.0008501751581206918] [Epoch time: 0.52 seconds]
```

Figure 3.3: GAN Training Accuracy Over Epochs

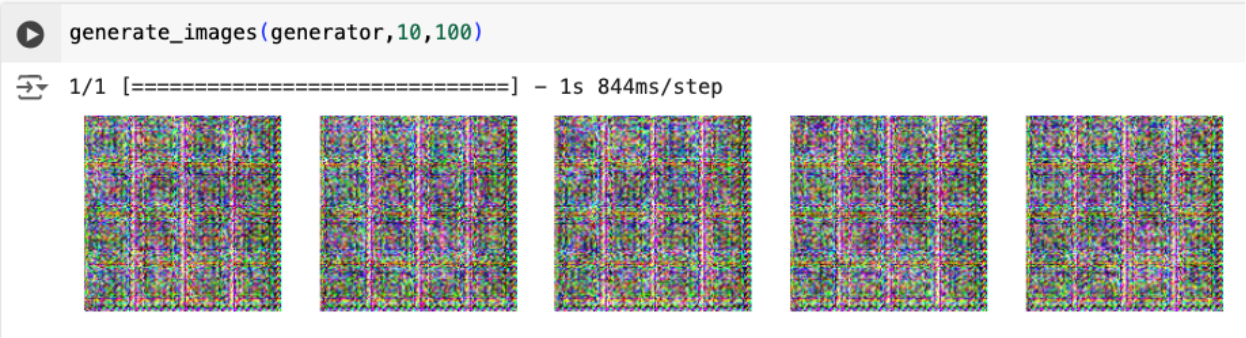


Figure 3.4: Generated Images from GAN

FID (Fréchet Inception Distance) quantifies the difference between generated data and real data by comparing their distribution in the Inception network feature space. FID takes into account the overall distribution of generated data and real data, and can reflect the quality and diversity of generated data. A low FID value indicates that the distribution of generated data is very close to the distribution of real data, that is, the generated data is both realistic and diverse, so FID is a more suitable indicator for evaluating the performance of generative models.

Chapter 4

Results

This chapter details the numerical processes undertaken to evaluate the performance of various GAN architectures. It includes model selection, structural adjustments, exploration of layer depth, and the impact of data augmentation on model performance. Additionally, the chapter discusses the application of the chosen model to a new dataset, providing insights into the practical effectiveness of the models in generating high-quality images.

4.1 Standard GAN Versus Other GAN Realizations

Approximately a decade has passed since Goodfellow introduced Generative Adversarial Networks (GANs), during which numerous variants of GAN models have been developed. For the purpose of this study, I have selected seven distinct GAN models for examination and minimal implementation: Standard GANs, Conditional GANs, Auxiliary Classifier GANs, Cycle GANs, Domain Transfer Network GANs, Coupled GANs, and Style GANs. Based on their foundational nature, extensive research and documentation, training and implementation efficiency, and flexibility and versatility, Standard GANs were selected for this study. The following outlines the reasons for this choice.

1. Foundational Nature: Standard GANs, introduced by Ian Goodfel-

low et al. in 2014, serve as the foundational model for all subsequent GAN variants. Understanding the principles and mechanics of Standard GANs is crucial for comprehending more complex versions like Conditional GANs, Cycle GANs, and Style GANs. By focusing on the Standard GAN, this study lays a solid foundation for exploring more advanced models.

2. Widely Studied and Well-Documented: Standard GANs have been extensively researched, with a vast amount of literature available. This wealth of resources provides a robust theoretical background and a variety of implementation strategies, facilitating a more thorough and well-supported analysis. This also means that there is ample precedent for common challenges and solutions, making it easier to troubleshoot and refine the model during the study.

3. Training and Implementation Efficiency: Compared to more complex GAN variants, Standard GANs typically require less computational power and shorter training times, making them more accessible for experimentation and analysis. This efficiency allows for multiple experiments and parameter tuning within the constraints of the study, leading to more reliable and reproducible results.

4. Flexibility and Versatility: Standard GANs are highly versatile and can be adapted to a wide range of tasks and datasets. This flexibility makes them an excellent choice for a detailed study that may involve exploring various applications or extending the model to new domains.

4.2 GAN With Convolutional or Dense layers

On the MNIST dataset for 3000 epochs, I compared the generated images from two GAN architectures: one using dense layers and the other using convolutional layers (CNN). It was observed that the GAN with the CNN

architecture outperformed the dense layer architecture in terms of image quality, producing clearer and more realistic images.

As shown in Figures 4.1 and 4.2, the generator and discriminator architectures for both dense and convolutional layers are compared. Figure 4.1a illustrates the generator with dense layers, which consists of a series of fully connected layers, while Figure 4.1b shows the generator with convolutional layers, where convolutional and transpose convolutional layers are used to capture spatial features more effectively. The same comparison applies to the discriminator architectures shown in Figure 4.2. The dense-layer discriminator (Figure 4.2a) is built with fully connected layers, whereas the convolutional discriminator (Figure 4.2b) leverages convolutional layers to better identify patterns in the data.

Layer (type)	Output Shape	Param #
dense_30 (Dense)	(None, 256)	25856
leaky_re_lu_22 (LeakyReLU)	(None, 256)	0
batch_normalization_18 (BatchNormalization)	(None, 256)	1024
dense_31 (Dense)	(None, 512)	131584
leaky_re_lu_23 (LeakyReLU)	(None, 512)	0
batch_normalization_19 (BatchNormalization)	(None, 512)	2048
dense_32 (Dense)	(None, 1024)	525312
leaky_re_lu_24 (LeakyReLU)	(None, 1024)	0
batch_normalization_20 (BatchNormalization)	(None, 1024)	4096
dense_33 (Dense)	(None, 784)	803600
reshape_6 (Reshape)	(None, 28, 28, 1)	0

Total params: 1493520 (5.70 MB)
Trainable params: 1489936 (5.68 MB)
Non-trainable params: 3584 (14.00 KB)

(a) Generator with dense layer

Layer (type)	Output Shape	Param #
dense_38 (Dense)	(None, 6272)	633472
leaky_re_lu_28 (LeakyReLU)	(None, 6272)	0
reshape_8 (Reshape)	(None, 7, 7, 128)	0
batch_normalization_24 (BatchNormalization)	(None, 7, 7, 128)	512
conv2d_transpose (Conv2DTranspose)	(None, 14, 14, 128)	262272
leaky_re_lu_29 (LeakyReLU)	(None, 14, 14, 128)	0
batch_normalization_25 (BatchNormalization)	(None, 14, 14, 128)	512
conv2d_transpose_1 (Conv2DTranspose)	(None, 28, 28, 64)	131136
leaky_re_lu_30 (LeakyReLU)	(None, 28, 28, 64)	0
batch_normalization_26 (BatchNormalization)	(None, 28, 28, 64)	256
conv2d (Conv2D)	(None, 28, 28, 1)	3137

Total params: 1031297 (3.93 MB)
Trainable params: 1030657 (3.93 MB)
Non-trainable params: 640 (2.50 KB)

(b) Generator with convolution layer

Figure 4.1: Generator Architecture with Dense and Convolutional Layers

Layer (type)	Output Shape	Param #
flatten_1 (Flatten)	(None, 784)	0
dense_23 (Dense)	(None, 512)	401920
leaky_re_lu_17 (LeakyReLU)	(None, 512)	0
dropout_2 (Dropout)	(None, 512)	0
dense_24 (Dense)	(None, 256)	131328
leaky_re_lu_18 (LeakyReLU)	(None, 256)	0
dropout_3 (Dropout)	(None, 256)	0
dense_25 (Dense)	(None, 1)	257

Total params: 533505 (2.04 MB)
Trainable params: 533505 (2.04 MB)
Non-trainable params: 0 (0.00 Byte)

(a) Discriminator with dense layer

Layer (type)	Output Shape	Param #
conv2d_2 (Conv2D)	(None, 14, 14, 64)	640
leaky_re_lu_39 (LeakyReLU)	(None, 14, 14, 64)	0
dropout_6 (Dropout)	(None, 14, 14, 64)	0
conv2d_3 (Conv2D)	(None, 7, 7, 128)	73856
leaky_re_lu_40 (LeakyReLU)	(None, 7, 7, 128)	0
dropout_7 (Dropout)	(None, 7, 7, 128)	0
flatten_3 (Flatten)	(None, 6272)	0
dense_47 (Dense)	(None, 1)	6273

Total params: 80769 (315.50 KB)
Trainable params: 80769 (315.50 KB)
Non-trainable params: 0 (0.00 Byte)

(b) Discriminator with convolution layer

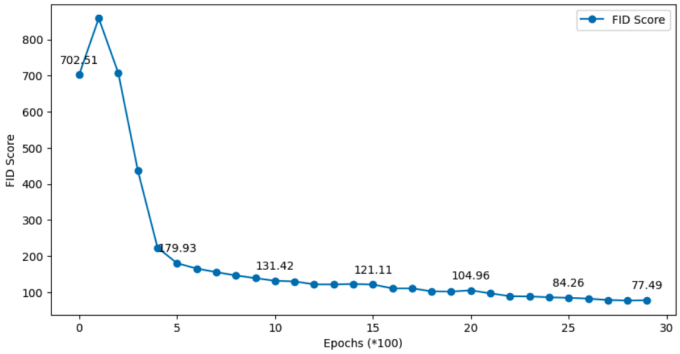
Figure 4.2: Discriminator Architecture with Dense and Convolutional Layers

Notably, the dense-layer models contain significantly more parameters compared to the convolutional models. The generator with dense layers has approximately 1.5 million parameters, whereas the convolutional generator has only about 1 million parameters. Similarly, the dense-layer discriminator has over 500,000 parameters, while the convolutional discriminator has only 80,000. Despite the larger parameter size, the dense-layer architectures were less effective in producing high-quality images, highlighting the advantage of convolutional layers in capturing spatial dependencies in the data.

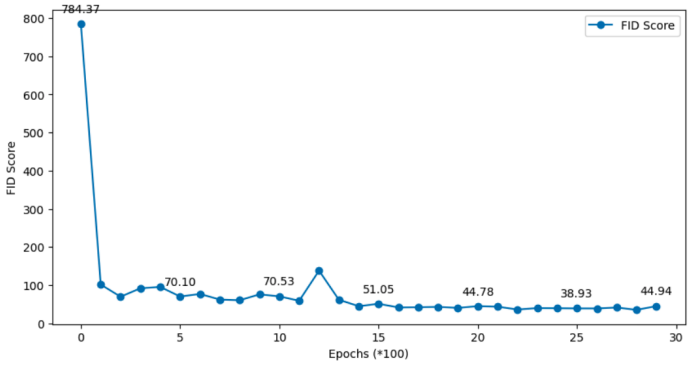
(a) Images generated by GAN with dense layers

(b) Images generated by GAN with convolution layers

Figure 4.3: Comparison of GAN Performance at 3000 Epochs



(a) FID score for dense layer



(b) FID score for convolutional layer

Figure 4.4: FID Scores Across 3000 Epochs

Based on the results, the GAN model with convolutional layers demonstrated superior performance in generating high-quality images compared to the dense layer model. As shown in the generated images (Figure 4.3), the convolutional GAN produced more coherent and realistic outputs, whereas the dense layer GAN struggled to generate clear and recognizable digits. Additionally, the FID scores, which measure the dissimilarity between generated and real images, further validate this observation. The FID scores for the convolutional GAN (Figure 4.4b) are consistently lower across training epochs compared to the dense layer model (Figure 4.4a), indicating better quality in the generated images.

Given these results, I selected the standard GAN model with convolutional layers for further experiments, as it not only generated higher-quality

images but also used fewer parameters, making it a more efficient and effective choice.

4.3 Exploring Layer Depth

In this section, I explored the impact of changing convolutional layers in the generator and discriminator of a GAN.

Firstly, I trained the basic convolutional GAN model three times and recorded the FID scores. Secondly, based on the basic structure, I gradually added convolutional layers to the generator, from one to three layers, while keeping the discriminator structure fixed. I trained the model three times for each configuration and recorded the FID scores, as shown in Table 4.1.

Finally, using the base GAN with three additional convolutional layers in the generator, I gradually added convolutional layers to the discriminator, from one to three layers, while keeping the generator structure fixed. I trained the model three times for each configuration and recorded the FID scores.

Table 4.1: Average FID Scores for Different GAN Architectures (Lower is Better)

Layers in G \ Layers in D	**3**	**4**	**5**	**6**
3	68.03	72.45	78.56	85.67
4	78.95	70.89	74.23	79.45
5	99.17	85.36	37.09	65.54
6	194.66	71.54	55.21	35.78

Note: A FID value below 10 is considered to represent very high-quality generated images, while values between 10 and 50 indicate good quality, and values above 50 suggest average or poor quality.

Upon comparing, I found that increasing the number of layers in either the generator or the discriminator alone worsened the model's performance. This imbalance disrupts the dynamic equilibrium between the generator and discriminator in the GAN framework. However, simultaneously increasing the layers in both the generator and the discriminator improved the model's

performance, as demonstrated by the decreasing FID scores with higher layer counts. For example, Table 4.1 shows that increasing both the generator and discriminator layers to five results in a significantly lower FID score of 37.09 compared to the unbalanced configurations.

This finding aligns with the concept of maintaining a balance between the generator and discriminator during training, as highlighted in the literature [26]. The importance of this balance is crucial for the effective operation of GANs, ensuring stable learning and improved image quality [27].

4.4 Impact of Data Augmentation on Model Performance

The impact of various data augmentation techniques on model performance was evaluated by comparing the average FID scores across different augmentation configurations (Table 4.2). The results indicate that data augmentation generally led to poorer performance in terms of image quality compared to training without any augmentation.

Table 4.2: Average FID Scores for Different Data Augmentation Techniques

Data Augmentation Technique	**Average FID Score**
Rotation 10° + Shifting 0.1 + Flipping	103.49
Rotation 10° + Shifting 0.1	76.04
Shifting 0.1	70.74
Shifting 0.05	66.06
Without Data Augmentation	58.94

Note: A FID value below 10 is considered to represent very high-quality generated images, while values between 10 and 50 indicate good quality, and values above 50 suggest average or poor quality.

Among the techniques tested, applying a combination of rotation, shifting, and flipping resulted in the highest FID score (103.49), suggesting a significant negative impact on the generated image quality. Rotation and

shifting alone slightly improved the FID score to 76.04, while reducing the shift range to 0.1 and 0.05 further improved performance, yielding FID scores of 70.74 and 66.06, respectively. The best performance was observed with no data augmentation, achieving an FID score of 58.94.

These findings suggest that, while data augmentation is commonly employed to improve model generalization, it can potentially disrupt the data distribution and hinder the optimization process when not carefully selected or overused.

4.5 Applying the Model to the Animal Faces-HQ Dataset

In this section, the Animal Faces-HQ (AFHQ) dataset was used to train a standard GAN focused on generating realistic cat images. The dataset consists of 16,130 high-quality images with a resolution of 512x512 pixels. Due to the high resolution of the original images, I downscaled them to 128x128 pixels to avoid GPU memory issues during training.

The GAN was then trained for 4000 epochs. The following results showcase the images generated by the model's generator. Figure 4.5 displays the cat faces created by the GAN.

4.5.1 Model Structure and Training

The GAN model consists of two main components: a generator and a discriminator. The generator takes a random noise vector and progressively transforms it into a full-resolution image, while the discriminator is trained to distinguish between real and generated images.

The generator architecture starts by expanding the noise vector (100 dimensions) into a 16x16x256 feature map. Several transpose convolutional layers are then used to upsample this feature map to 128x128 pixels. The discriminator processes these images using convolutional layers, which progressively downsample the input. The final output of the discriminator is a classification (real or fake).

Below is a brief summary of the generator and discriminator models:

```
def build_generator():
    model = Sequential()

    # Expand noise vector and reshape
    model.add(Dense(16*16*256, input_dim=100))
    model.add(LeakyReLU(alpha=0.2))
    model.add(Reshape((16, 16, 256)))
    model.add(BatchNormalization(momentum=0.8))

    # Upsample to 128x128
    model.add(Conv2DTranspose(256, kernel_size=4, strides=2, padding='same'))
    model.add(LeakyReLU(alpha=0.2))
    model.add(BatchNormalization(momentum=0.8))

    model.add(Conv2DTranspose(128, kernel_size=4, strides=2, padding='same'))
    model.add(LeakyReLU(alpha=0.2))
    model.add(BatchNormalization(momentum=0.8))

    model.add(Conv2DTranspose(64, kernel_size=4, strides=2, padding='same'))
    model.add(LeakyReLU(alpha=0.2))
    model.add(BatchNormalization(momentum=0.8))

    model.add(Conv2D(3, kernel_size=7, activation='tanh', padding='same'))

    return model
```

The discriminator processes the generated images through a series of convolutional layers and finally classifies them as real or fake.

```
def build_discriminator():
    model = Sequential()
```

```
    model.add(Conv2D(64, kernel_size=3, strides=2, input_shape=(128, 128, 3), pa
    model.add(LeakyReLU(alpha=0.2))
    model.add(Dropout(0.25))

    model.add(Conv2D(128, kernel_size=3, strides=2, padding='same'))
    model.add(LeakyReLU(alpha=0.2))
    model.add(Dropout(0.25))

    model.add(Conv2D(256, kernel_size=3, strides=2, padding='same'))
    model.add(LeakyReLU(alpha=0.2))
    model.add(Dropout(0.25))

    model.add(Conv2D(512, kernel_size=3, strides=2, padding='same'))
    model.add(LeakyReLU(alpha=0.2))
    model.add(Dropout(0.25))

    model.add(Flatten())
    model.add(Dense(1, activation='sigmoid'))

    return model
```

The training was conducted over 4000 epochs with a batch size of 128. Both real and generated images were used to train the discriminator, while the generator was trained to produce images that could fool the discriminator into classifying them as real.

```
def train(generator, discriminator, gan, x_train, epochs, batch_size=128):
    valid = np.ones((batch_size, 1))
    fake = np.zeros((batch_size, 1))

    for epoch in range(epochs):
        start_time = time.time()

        # Train the discriminator
```

```
idx = np.random.randint(0, x_train.shape[0], batch_size)
imgs = x_train[idx]

noise = np.random.normal(0, 1, (batch_size, 100))
gen_imgs = generator.predict(noise)

d_loss_real = discriminator.train_on_batch(imgs, valid)
d_loss_fake = discriminator.train_on_batch(gen_imgs, fake)
d_loss = 0.5 * np.add(d_loss_real, d_loss_fake)

# Train the generator
noise = np.random.normal(0, 1, (batch_size, 100))
g_loss = gan.train_on_batch(noise, valid)

end_time = time.time()
epoch_time = end_time - start_time

print(f"{epoch} [D loss: {d_loss[0]} | D accuracy: "
      f"{100 * d_loss[1]}] [G loss: {g_loss}] "
      f"[Epoch time: {epoch_time:.2f} seconds]")
```

4.5.2 Generated Results

Figure 4.5 shows examples of the cat images generated by the GAN after training for 4000 epochs. While the model was able to capture many of the essential features of cat faces, the generated outputs still show some variation compared to the original dataset.

Figure 4.5: Cat Faces Generated by GAN

The GAN's architecture and training details are included in the appendix. Additionally, the full Python code for this implementation is available to provide a detailed understanding of how the model was trained and evaluated.

4.6 Discussion

In this study, I conducted a series of experiments to evaluate the performance of different GAN architectures and explored various factors influencing the quality of generated images. Specifically, the experiments compared the effects of using dense layers versus convolutional layers in a standard GAN, examined the impact of varying the number of convolutional layers in both the generator and discriminator, analyzed the effects of data augmentation on GAN training, and applied a GAN model to a dataset to generate high-quality cat face images.

The experiments revealed that GAN models utilizing convolutional layers outperformed those using dense layers in terms of image quality, confirming the strength of convolutional layers in capturing spatial features. Additionally, I found that increasing the number of layers in the generator or discriminator independently led to a decline in performance, suggesting that the balance between these two components is critical for stable GAN training. Simultaneously increasing the number of layers in both components improved the model's output.

Regarding data augmentation, the results indicated that certain augmentation techniques negatively impacted GAN training, likely by introducing noise or disrupting the data distribution. This suggests that when applying data augmentation to GANs, careful selection and parameter tuning are necessary to avoid adverse effects on training stability and performance.

When applying the model to the Animal Faces-HQ dataset, I successfully trained the GAN to generate cat images. However, due to hardware constraints, I had to downscale the images from 512x512 to 128x128 pixels, which may have limited the generated images' resolution and quality.

The limitations of this study include computational restrictions, which prevented experiments on higher-resolution images, and a focus on standard GANs rather than more advanced architectures such as StyleGAN or BigGAN. Future work could explore these more complex GAN architectures and improve data augmentation strategies to enhance training performance and image quality. Additionally, training GANs on higher-resolution datasets

and experimenting with advanced GAN models, such as Conditional GANs or Adaptive GANs, could provide further insights into enhancing image diversity and fidelity.

In conclusion, this study demonstrates the effectiveness of GAN models in generating high-quality images, while highlighting areas for future improvements in architecture and training methods.

Appendix A

Code

Listing A.1: GAN model with dense layers

```
import tensorflow as tf
from tensorflow.keras.layers import Dense, Reshape, Flatten,
    Dropout, LeakyReLU, BatchNormalization
from tensorflow.keras.models import Sequential
from tensorflow.keras.optimizers import Adam
from scipy.linalg import sqrtm
import numpy as np
import time
import matplotlib.pyplot as plt

np.random.seed(1000)
tf.random.set_seed(1000)

# input 100
# output 28*28*1

def build_generator():
    model = Sequential()

    # increase the dimension
    model.add(Dense(256, input_dim=100))
    model.add(LeakyReLU(alpha=0.2))
    model.add(BatchNormalization(momentum=0.8))

```

```
    model.add(Dense(512))
    model.add(LeakyReLU(alpha=0.2))
    model.add(BatchNormalization(momentum=0.8))

    model.add(Dense(1024))
    model.add(LeakyReLU(alpha=0.2))
    model.add(BatchNormalization(momentum=0.8))

    model.add(Dense(28*28, activation='tanh'))
    model.add(Reshape((28, 28, 1)))

    return model

# input 28*28*1
# output 1

def build_discriminator():
    model = Sequential()

    model.add(Flatten(input_shape=(28, 28, 1)))

    model.add(Dense(512))
    model.add(LeakyReLU(alpha=0.2))
    model.add(Dropout(0.25))

    model.add(Dense(256))
    model.add(LeakyReLU(alpha=0.2))
    model.add(Dropout(0.25))

    model.add(Dense(1, activation='sigmoid'))

    return model

# Load and preprocess the MNIST dataset
(x_train, _), (_, _) = tf.keras.datasets.mnist.load_data()
x_train = (x_train - 127.5) / 127.5
x_train = np.expand_dims(x_train, axis=3)

# Calculate FID function
```

```
63 def calculate_fid(real_images, fake_images):
64     act1 = real_images.reshape((real_images.shape[0], -1))
65     mu1, sigma1 = act1.mean(axis=0), np.cov(act1, rowvar=
           False)
66
67     act2 = fake_images.reshape((fake_images.shape[0], -1))
68     mu2, sigma2 = act2.mean(axis=0), np.cov(act2, rowvar=
           False)
69
70     ssdiff = np.sum((mu1 - mu2)**2.0)
71     covmean = sqrtm(sigma1.dot(sigma2))
72
73     if np.iscomplexobj(covmean):
74         covmean = covmean.real
75
76     fid = ssdiff + np.trace(sigma1 + sigma2 - 2.0 * covmean)
77
78     return fid
79
80 def train_gan(epochs=1000, batch_size=64, p_epoch=100):
81     generator = build_generator()
82     discriminator = build_discriminator()
83
84     discriminator.compile(loss='binary_crossentropy',
           optimizer=Adam(0.0002, 0.5), metrics=['accuracy'])
85     discriminator.trainable = False
86
87     gan_input = tf.keras.Input(shape=(100,))
88     gan_output = discriminator(generator(gan_input))
89     gan = tf.keras.Model(gan_input, gan_output)
90     gan.compile(loss='binary_crossentropy', optimizer=Adam
           (0.0002, 0.5))
91
92     half_batch = int(batch_size / 2)
93
94     d_losses = []
95     g_losses = []
96     d_acc = []
97     fid_scores = []  # List to store FID scores
```

```
98
99      start_time = time.time()  # Record the start time
100
101     for epoch in range(epochs):
102         # Select a random half batch of real images
103         idx = np.random.randint(0, x_train.shape[0],
                half_batch)
104         real_images = x_train[idx]
105
106         # Generate a half batch of new fake images
107         noise = np.random.normal(0, 1, (half_batch, 100))
108         fake_images = generator.predict(noise)
109
110         # Train the discriminator
111         real_labels = np.ones((half_batch, 1))
112         fake_labels = np.zeros((half_batch, 1))
113
114         d_loss_real = discriminator.train_on_batch(
                real_images, real_labels)
115         d_loss_fake = discriminator.train_on_batch(
                fake_images, fake_labels)
116         d_loss = 0.5 * np.add(d_loss_real, d_loss_fake)
117
118         # Train the generator
119         noise = np.random.normal(0, 1, (batch_size, 100))
120         valid_y = np.ones((batch_size, 1))
121
122         g_loss = gan.train_on_batch(noise, valid_y)
123
124         # Record the losses
125         d_losses.append(d_loss[0])
126         g_losses.append(g_loss)
127         d_acc.append(d_loss[1] * 100)
128
129         # Calculate and print FID every p_epoch epochs
130         if epoch % p_epoch == 0:
131             noise = np.random.normal(0, 1, (1000, 100))
132             fake_images = generator.predict(noise)
133             fid = calculate_fid(x_train[:1000], fake_images)
```

```
            fid_scores.append(fid)
            print(f"{epoch} [D loss: {d_loss[0]}, acc.: {100
                *d_loss[1]}%] [G loss: {g_loss}] [FID: {fid}]
                ")

    end_time = time.time()  # Record the end time
    total_time = end_time - start_time
    print(f"Total training time: {total_time:.2f} seconds")
    return generator, d_losses, g_losses, d_acc, fid_scores
```

Listing A.2: GAN model with Convolutional layers

```
import tensorflow as tf
from tensorflow.keras.layers import Dense, Reshape,
    Flatten, Dropout, LeakyReLU, Conv2D, Conv2DTranspose,
    BatchNormalization
from tensorflow.keras.models import Sequential
from tensorflow.keras.optimizers import Adam
from scipy.linalg import sqrtm
import numpy as np
import time
import matplotlib.pyplot as plt

np.random.seed(1000)
tf.random.set_seed(1000)

# input 100
# output 28*28*1

def build_generator():
    model = Sequential()

    # increase the dimension
    model.add(Dense(7*7*128, input_dim=100))
    model.add(LeakyReLU(alpha=0.2))
    model.add(Reshape((7, 7, 128)))
    model.add(BatchNormalization(momentum=0.8))

    model.add(Conv2DTranspose(128, kernel_size=4, strides
        =2, padding='same'))
```

```
26          model.add(LeakyReLU(alpha=0.2))
27          model.add(BatchNormalization(momentum=0.8))
28
29          model.add(Conv2DTranspose(64, kernel_size=4, strides
                =2, padding='same'))
30          model.add(LeakyReLU(alpha=0.2))
31          model.add(BatchNormalization(momentum=0.8))
32
33          model.add(Conv2D(1, kernel_size=7, activation='tanh',
                 padding='same'))
34
35          return model
36
37      # input 28*28*1
38      # output 1
39
40      def build_discriminator():
41          model = Sequential()
42
43          model.add(Conv2D(64, kernel_size=3, strides=2,
                input_shape=(28, 28, 1), padding='same'))
44          model.add(LeakyReLU(alpha=0.2))
45          model.add(Dropout(0.25))
46
47          model.add(Conv2D(128, kernel_size=3, strides=2,
                padding='same'))
48          model.add(LeakyReLU(alpha=0.2))
49          model.add(Dropout(0.25))
50
51          model.add(Flatten())
52          model.add(Dense(1, activation='sigmoid'))
53
54          return model
55
56      # Load and preprocess the MNIST dataset
57      (x_train, _), (_, _) = tf.keras.datasets.mnist.load_data
            ()
58      x_train = (x_train - 127.5) / 127.5
59      x_train = np.expand_dims(x_train, axis=3)
```

```
60
61      # Calculate FID function
62      def calculate_fid(real_images, fake_images):
63          act1 = real_images.reshape((real_images.shape[0], -1)
                )
64          mu1, sigma1 = act1.mean(axis=0), np.cov(act1, rowvar=
                False)
65
66          act2 = fake_images.reshape((fake_images.shape[0], -1)
                )
67          mu2, sigma2 = act2.mean(axis=0), np.cov(act2, rowvar=
                False)
68
69          ssdiff = np.sum((mu1 - mu2)**2.0)
70          covmean = sqrtm(sigma1.dot(sigma2))
71
72          if np.iscomplexobj(covmean):
73              covmean = covmean.real
74
75          fid = ssdiff + np.trace(sigma1 + sigma2 - 2.0 *
                covmean)
76
77          return fid
78
79      def train_gan(epochs=1000, batch_size=64, p_epoch=100):
80          generator = build_generator()
81          discriminator = build_discriminator()
82
83          discriminator.compile(loss='binary_crossentropy',
                optimizer=Adam(0.0002, 0.5), metrics=['accuracy'])
84          discriminator.trainable = False
85
86          gan_input = tf.keras.Input(shape=(100,))
87          gan_output = discriminator(generator(gan_input))
88          gan = tf.keras.Model(gan_input, gan_output)
89          gan.compile(loss='binary_crossentropy', optimizer=
                Adam(0.0002, 0.5))
90
91          # using half batches for the discriminator ensures
```

```
            balanced and efficient training,
92          # better memory management, and more stable training
            dynamics in GANs.
93          half_batch = int(batch_size / 2)
94
95          d_losses = []
96          g_losses = []
97          d_acc = []
98          fid_scores = []  # List to store FID scores
99
100         start_time = time.time()  # Record the start time
101
102         for epoch in range(epochs):
103             # Select a random half batch of real images
104             idx = np.random.randint(0, x_train.shape[0],
                    half_batch)
105             real_images = x_train[idx]
106
107             # Generate a half batch of new fake images
108             noise = np.random.normal(0, 1, (half_batch, 100))
109             fake_images = generator.predict(noise)
110
111             # Train the discriminator
112             real_labels = np.ones((half_batch, 1))
113             fake_labels = np.zeros((half_batch, 1))
114
115             d_loss_real = discriminator.train_on_batch(
                    real_images, real_labels)
116             d_loss_fake = discriminator.train_on_batch(
                    fake_images, fake_labels)
117             d_loss = 0.5 * np.add(d_loss_real, d_loss_fake)
118
119             # Train the generator
120             noise = np.random.normal(0, 1, (batch_size, 100))
121             valid_y = np.ones((batch_size, 1))
122
123             g_loss = gan.train_on_batch(noise, valid_y)
124
125             # Record the losses
```

```
            d_losses.append(d_loss[0])
            g_losses.append(g_loss)
            d_acc.append(d_loss[1] * 100)

            # Calculate FID every p_epoch epochs
            if epoch % p_epoch == 0:
                noise = np.random.normal(0, 1, (1000, 100))
                fake_images = generator.predict(noise)
                fid = calculate_fid(x_train[:1000],
                    fake_images)
                fid_scores.append(fid)
                print(f"{epoch} [D loss: {d_loss[0]}, acc.:
                    {100 * d_loss[1]}%] [G loss: {g_loss}] [
                    FID: {fid}]")

        end_time = time.time()  # Record the end time
        total_time = end_time - start_time
        print(f"Total training time: {total_time:.2f} seconds
            ")
        return generator, d_losses, g_losses, d_acc,
            fid_scores
```

Listing A.3: Explore data augmentaion 1

```
    import tensorflow as tf
    from tensorflow.keras.layers import Dense, Reshape,
        Flatten, Dropout, LeakyReLU, Conv2D, Conv2DTranspose,
        BatchNormalization
    from tensorflow.keras.models import Sequential
    from tensorflow.keras.optimizers import Adam
    from scipy.linalg import sqrtm
    import numpy as np
    import time
    import matplotlib.pyplot as plt

    np.random.seed(1000)
    tf.random.set_seed(1000)

    # input 100
    # output 28*28*1
```

```
15
16      def build_generator():
17          model = Sequential()
18
19          # increase the dimension
20          model.add(Dense(7*7*128, input_dim=100))
21          model.add(LeakyReLU(alpha=0.2))
22          model.add(Reshape((7, 7, 128)))
23          model.add(BatchNormalization(momentum=0.8))
24
25          model.add(Conv2DTranspose(128, kernel_size=4, strides
                =2, padding='same'))
26          model.add(LeakyReLU(alpha=0.2))
27          model.add(BatchNormalization(momentum=0.8))
28
29          model.add(Conv2DTranspose(64, kernel_size=4, strides
                =2, padding='same'))
30          model.add(LeakyReLU(alpha=0.2))
31          model.add(BatchNormalization(momentum=0.8))
32
33          model.add(Conv2D(1, kernel_size=7, activation='tanh',
                 padding='same'))
34
35          return model
36
37      # input 28*28*1
38      # output 1
39
40      def build_discriminator():
41          model = Sequential()
42
43          model.add(Conv2D(64, kernel_size=3, strides=2,
                input_shape=(28, 28, 1), padding='same'))
44          model.add(LeakyReLU(alpha=0.2))
45          model.add(Dropout(0.25))
46
47          model.add(Conv2D(128, kernel_size=3, strides=2,
                padding='same'))
48          model.add(LeakyReLU(alpha=0.2))
```

```
        model.add(Dropout(0.25))

        model.add(Flatten())
        model.add(Dense(1, activation='sigmoid'))

        return model

    (x_train, _), (_, _) = tf.keras.datasets.mnist.load_data()
    x_train = (x_train - 127.5) / 127.5
    x_train = np.expand_dims(x_train, axis=3)

    datagen = tf.keras.preprocessing.image.ImageDataGenerator(
        rotation_range=10,
        width_shift_range=0.1,
        height_shift_range=0.1,
        horizontal_flip=True
    )

    def calculate_fid(real_images, fake_images):
        act1 = real_images.reshape((real_images.shape[0], -1))
        mu1, sigma1 = act1.mean(axis=0), np.cov(act1, rowvar=False)

        act2 = fake_images.reshape((fake_images.shape[0], -1))
        mu2, sigma2 = act2.mean(axis=0), np.cov(act2, rowvar=False)

        ssdiff = np.sum((mu1 - mu2)**2.0)
        covmean = sqrtm(sigma1.dot(sigma2))

        if np.iscomplexobj(covmean):
            covmean = covmean.real

        fid = ssdiff + np.trace(sigma1 + sigma2 - 2.0 * covmean)
```

```
81
82          return fid
83
84      def train_gan(epochs=1000, batch_size=64, p_epoch=100):
85          generator = build_generator()
86          discriminator = build_discriminator()
87
88          discriminator.compile(loss='binary_crossentropy',
                optimizer=Adam(0.0002, 0.5), metrics=['accuracy'])
89          discriminator.trainable = False
90
91          gan_input = tf.keras.Input(shape=(100,))
92          gan_output = discriminator(generator(gan_input))
93          gan = tf.keras.Model(gan_input, gan_output)
94          gan.compile(loss='binary_crossentropy', optimizer=
                Adam(0.0002, 0.5))
95
96          half_batch = int(batch_size / 2)
97
98          d_losses = []
99          g_losses = []
100         d_acc = []
101         fid_scores = []
102
103         start_time = time.time()  # Record the start time
104
105         for epoch in range(epochs):
106             # Select a random half batch of real images
107             idx = np.random.randint(0, x_train.shape[0],
                    half_batch)
108             real_images = x_train[idx]
109
110             real_images_augmented = next(datagen.flow(
                    real_images, batch_size=half_batch))
111
112             # Generate a half batch of new fake images
113             noise = np.random.normal(0, 1, (half_batch, 100))
114             fake_images = generator.predict(noise)
115
```

```
            # Train the discriminator
            real_labels = np.ones((half_batch, 1))
            fake_labels = np.zeros((half_batch, 1))

            d_loss_real = discriminator.train_on_batch(
                real_images_augmented, real_labels)
            d_loss_fake = discriminator.train_on_batch(
                fake_images, fake_labels)
            d_loss = 0.5 * np.add(d_loss_real, d_loss_fake)

            # Train the generator
            noise = np.random.normal(0, 1, (batch_size, 100))
            valid_y = np.ones((batch_size, 1))

            g_loss = gan.train_on_batch(noise, valid_y)

            # Record the losses
            d_losses.append(d_loss[0])
            g_losses.append(g_loss)
            d_acc.append(d_loss[1] * 100)

            # Calculate FID every p_epoch epochs
            if epoch % p_epoch == 0:
                noise = np.random.normal(0, 1, (1000, 100))
                fake_images = generator.predict(noise)
                fid = calculate_fid(x_train[:1000],
                    fake_images)
                fid_scores.append(fid)
                print(f"{epoch} [D loss: {d_loss[0]}, acc.: 
                    {100 * d_loss[1]}%] [G loss: {g_loss}] [
                    FID: {fid}]")

        end_time = time.time()  # Record the end time
        total_time = end_time - start_time
        print(f"Total training time: {total_time:.2f} seconds
            ")
        return generator, d_losses, g_losses, d_acc,
            fid_scores

```

```
148 # Training the GAN with data augmentation and FID
        calculation
149 generator, d_losses, g_losses, d_acc, fid_scores =
        train_gan(epochs=1000, batch_size=64, p_epoch=100)
```

Listing A.4: Explore data augmentation 2

```
1  import tensorflow as tf
2  from tensorflow.keras.layers import Dense, Reshape,
       Flatten, Dropout, LeakyReLU, Conv2D, Conv2DTranspose,
       BatchNormalization
3  from tensorflow.keras.models import Sequential
4  from tensorflow.keras.optimizers import Adam
5  from scipy.linalg import sqrtm
6  import numpy as np
7  import time
8  import matplotlib.pyplot as plt
9
10 np.random.seed(1000)
11 tf.random.set_seed(1000)
12
13 # input 100
14 # output 28*28*1
15
16 def build_generator():
17     model = Sequential()
18
19     # increase the dimension
20     model.add(Dense(7*7*128, input_dim=100))
21     model.add(LeakyReLU(alpha=0.2))
22     model.add(Reshape((7, 7, 128)))
23     model.add(BatchNormalization(momentum=0.8))
24
25     model.add(Conv2DTranspose(128, kernel_size=4, strides
           =2, padding='same'))
26     model.add(LeakyReLU(alpha=0.2))
27     model.add(BatchNormalization(momentum=0.8))
28
29     model.add(Conv2DTranspose(64, kernel_size=4, strides
           =2, padding='same'))
```

```
30          model.add(LeakyReLU(alpha=0.2))
31          model.add(BatchNormalization(momentum=0.8))
32
33          model.add(Conv2D(1, kernel_size=7, activation='tanh',
                padding='same'))
34
35          return model
36
37      # input 28*28*1
38      # output 1
39
40      def build_discriminator():
41          model = Sequential()
42
43          model.add(Conv2D(64, kernel_size=3, strides=2,
                input_shape=(28, 28, 1), padding='same'))
44          model.add(LeakyReLU(alpha=0.2))
45          model.add(Dropout(0.25))
46
47          model.add(Conv2D(128, kernel_size=3, strides=2,
                padding='same'))
48          model.add(LeakyReLU(alpha=0.2))
49          model.add(Dropout(0.25))
50
51          model.add(Flatten())
52          model.add(Dense(1, activation='sigmoid'))
53
54          return model
55
56      (x_train, _), (_, _) = tf.keras.datasets.mnist.load_data
            ()
57      x_train = (x_train - 127.5) / 127.5
58      x_train = np.expand_dims(x_train, axis=3)
59
60      datagen = tf.keras.preprocessing.image.ImageDataGenerator
            (
61          rotation_range=10,
62          width_shift_range=0.1,
63          height_shift_range=0.1,
```

```
    )

    def calculate_fid(real_images, fake_images):
        act1 = real_images.reshape((real_images.shape[0], -1))
        mu1, sigma1 = act1.mean(axis=0), np.cov(act1, rowvar=False)

        act2 = fake_images.reshape((fake_images.shape[0], -1))
        mu2, sigma2 = act2.mean(axis=0), np.cov(act2, rowvar=False)

        ssdiff = np.sum((mu1 - mu2)**2.0)
        covmean = sqrtm(sigma1.dot(sigma2))

        if np.iscomplexobj(covmean):
            covmean = covmean.real

        fid = ssdiff + np.trace(sigma1 + sigma2 - 2.0 * covmean)

        return fid

    def train_gan(epochs=1000, batch_size=64, p_epoch=100):
        generator = build_generator()
        discriminator = build_discriminator()

        discriminator.compile(loss='binary_crossentropy', optimizer=Adam(0.0002, 0.5), metrics=['accuracy'])
        discriminator.trainable = False

        gan_input = tf.keras.Input(shape=(100,))
        gan_output = discriminator(generator(gan_input))
        gan = tf.keras.Model(gan_input, gan_output)
        gan.compile(loss='binary_crossentropy', optimizer=Adam(0.0002, 0.5))

        half_batch = int(batch_size / 2)
```

```
96
97          d_losses = []
98          g_losses = []
99          d_acc = []
100         fid_scores = []
101
102         start_time = time.time()  # Record the start time
103
104         for epoch in range(epochs):
105             # Select a random half batch of real images
106             idx = np.random.randint(0, x_train.shape[0],
                    half_batch)
107             real_images = x_train[idx]
108
109             real_images_augmented = next(datagen.flow(
                    real_images, batch_size=half_batch))
110
111             # Generate a half batch of new fake images
112             noise = np.random.normal(0, 1, (half_batch, 100))
113             fake_images = generator.predict(noise)
114
115             # Train the discriminator
116             real_labels = np.ones((half_batch, 1))
117             fake_labels = np.zeros((half_batch, 1))
118
119             d_loss_real = discriminator.train_on_batch(
                    real_images_augmented, real_labels)
120             d_loss_fake = discriminator.train_on_batch(
                    fake_images, fake_labels)
121             d_loss = 0.5 * np.add(d_loss_real, d_loss_fake)
122
123             # Train the generator
124             noise = np.random.normal(0, 1, (batch_size, 100))
125             valid_y = np.ones((batch_size, 1))
126
127             g_loss = gan.train_on_batch(noise, valid_y)
128
129             # Record the losses
130             d_losses.append(d_loss[0])
```

```
            g_losses.append(g_loss)
            d_acc.append(d_loss[1] * 100)

            # Calculate FID every p_epoch epochs
            if epoch % p_epoch == 0:
                noise = np.random.normal(0, 1, (1000, 100))
                fake_images = generator.predict(noise)
                fid = calculate_fid(x_train[:1000],
                    fake_images)
                fid_scores.append(fid)
                print(f"{epoch} [D loss: {d_loss[0]}, acc.: 
                    {100 * d_loss[1]}%] [G loss: {g_loss}] [
                    FID: {fid}]")

        end_time = time.time()  # Record the end time
        total_time = end_time - start_time
        print(f"Total training time: {total_time:.2f} seconds
            ")
        return generator, d_losses, g_losses, d_acc,
            fid_scores

    # Training the GAN with data augmentation and FID
        calculation
    generator, d_losses, g_losses, d_acc, fid_scores =
        train_gan(epochs=1000, batch_size=64, p_epoch=100)
```

Listing A.5: Explore data augmetation 3

```
    import tensorflow as tf
    from tensorflow.keras.layers import Dense, Reshape,
        Flatten, Dropout, LeakyReLU, Conv2D, Conv2DTranspose,
        BatchNormalization
    from tensorflow.keras.models import Sequential
    from tensorflow.keras.optimizers import Adam
    from scipy.linalg import sqrtm
    import numpy as np
    import time
    import matplotlib.pyplot as plt

    np.random.seed(1000)
```

```
    tf.random.set_seed(1000)

    # input 100
    # output 28*28*1

    def build_generator():
        model = Sequential()

        # increase the dimension
        model.add(Dense(7*7*128, input_dim=100))
        model.add(LeakyReLU(alpha=0.2))
        model.add(Reshape((7, 7, 128)))
        model.add(BatchNormalization(momentum=0.8))

        model.add(Conv2DTranspose(128, kernel_size=4, strides
            =2, padding='same'))
        model.add(LeakyReLU(alpha=0.2))
        model.add(BatchNormalization(momentum=0.8))

        model.add(Conv2DTranspose(64, kernel_size=4, strides
            =2, padding='same'))
        model.add(LeakyReLU(alpha=0.2))
        model.add(BatchNormalization(momentum=0.8))

        model.add(Conv2D(1, kernel_size=7, activation='tanh',
             padding='same'))

        return model

    # input 28*28*1
    # output 1

    def build_discriminator():
        model = Sequential()

        model.add(Conv2D(64, kernel_size=3, strides=2,
            input_shape=(28, 28, 1), padding='same'))
        model.add(LeakyReLU(alpha=0.2))
        model.add(Dropout(0.25))
```

```
46
47          model.add(Conv2D(128, kernel_size=3, strides=2,
                padding='same'))
48          model.add(LeakyReLU(alpha=0.2))
49          model.add(Dropout(0.25))
50
51          model.add(Flatten())
52          model.add(Dense(1, activation='sigmoid'))
53
54          return model
55
56      (x_train, _), (_, _) = tf.keras.datasets.mnist.load_data
            ()
57      x_train = (x_train - 127.5) / 127.5
58      x_train = np.expand_dims(x_train, axis=3)
59
60      datagen = tf.keras.preprocessing.image.ImageDataGenerator
            (
61          width_shift_range=0.1,
62          height_shift_range=0.1,
63      )
64
65      def calculate_fid(real_images, fake_images):
66          act1 = real_images.reshape((real_images.shape[0], -1)
                )
67          mu1, sigma1 = act1.mean(axis=0), np.cov(act1, rowvar=
                False)
68
69          act2 = fake_images.reshape((fake_images.shape[0], -1)
                )
70          mu2, sigma2 = act2.mean(axis=0), np.cov(act2, rowvar=
                False)
71
72          ssdiff = np.sum((mu1 - mu2)**2.0)
73          covmean = sqrtm(sigma1.dot(sigma2))
74
75          if np.iscomplexobj(covmean):
76              covmean = covmean.real
77
```

```
78          fid = ssdiff + np.trace(sigma1 + sigma2 - 2.0 *
                covmean)
79
80          return fid
81
82      def train_gan(epochs=1000, batch_size=64, p_epoch=100):
83          generator = build_generator()
84          discriminator = build_discriminator()
85
86          discriminator.compile(loss='binary_crossentropy',
                optimizer=Adam(0.0002, 0.5), metrics=['accuracy'])
87          discriminator.trainable = False
88
89          gan_input = tf.keras.Input(shape=(100,))
90          gan_output = discriminator(generator(gan_input))
91          gan = tf.keras.Model(gan_input, gan_output)
92          gan.compile(loss='binary_crossentropy', optimizer=
                Adam(0.0002, 0.5))
93
94          half_batch = int(batch_size / 2)
95
96          d_losses = []
97          g_losses = []
98          d_acc = []
99          fid_scores = []
100
101         start_time = time.time()  # Record the start time
102
103         for epoch in range(epochs):
104             # Select a random half batch of real images
105             idx = np.random.randint(0, x_train.shape[0],
                    half_batch)
106             real_images = x_train[idx]
107
108             real_images_augmented = next(datagen.flow(
                    real_images, batch_size=half_batch))
109
110             # Generate a half batch of new fake images
111             noise = np.random.normal(0, 1, (half_batch, 100))
```

```
            fake_images = generator.predict(noise)

            # Train the discriminator
            real_labels = np.ones((half_batch, 1))
            fake_labels = np.zeros((half_batch, 1))

            d_loss_real = discriminator.train_on_batch(
                real_images_augmented, real_labels)
            d_loss_fake = discriminator.train_on_batch(
                fake_images, fake_labels)
            d_loss = 0.5 * np.add(d_loss_real, d_loss_fake)

            # Train the generator
            noise = np.random.normal(0, 1, (batch_size, 100))
            valid_y = np.ones((batch_size, 1))

            g_loss = gan.train_on_batch(noise, valid_y)

            # Record the losses
            d_losses.append(d_loss[0])
            g_losses.append(g_loss)
            d_acc.append(d_loss[1] * 100)

            # Calculate FID every p_epoch epochs
            if epoch % p_epoch == 0:
                noise = np.random.normal(0, 1, (1000, 100))
                fake_images = generator.predict(noise)
                fid = calculate_fid(x_train[:1000],
                    fake_images)
                fid_scores.append(fid)
                print(f"{epoch} [D loss: {d_loss[0]}, acc.: 
                    {100 * d_loss[1]}%] [G loss: {g_loss}] [
                    FID: {fid}]")

        end_time = time.time()  # Record the end time
        total_time = end_time - start_time
        print(f"Total training time: {total_time:.2f} seconds
            ")
        return generator, d_losses, g_losses, d_acc,
```

```
            fid_scores

    # Training the GAN with data augmentation and FID
        calculation
    generator, d_losses, g_losses, d_acc, fid_scores =
        train_gan(epochs=1000, batch_size=64, p_epoch=100)
```

Listing A.6: Explore GAN with more convolutional layers 1

```
import tensorflow as tf
from tensorflow.keras.layers import Dense, Reshape,
    Flatten, Dropout, LeakyReLU, Conv2D, Conv2DTranspose,
    BatchNormalization
from tensorflow.keras.models import Sequential
from tensorflow.keras.optimizers import Adam
from scipy.linalg import sqrtm
import numpy as np
import time
import matplotlib.pyplot as plt

np.random.seed(1000)
tf.random.set_seed(1000)

# input 100
# output 28*28*1

def build_generator():
    model = Sequential()

    model.add(Dense(7*7*128, input_dim=100))
    model.add(LeakyReLU(alpha=0.2))
    model.add(Reshape((7, 7, 128)))
    model.add(BatchNormalization(momentum=0.8))

    # add 1 convolution layer
    model.add(Conv2D(128, kernel_size=3, strides=1,
        padding='same'))
    model.add(LeakyReLU(alpha=0.2))
    model.add(BatchNormalization(momentum=0.8))

```

```
29          model.add(Conv2DTranspose(128, kernel_size=4, strides
                =2, padding='same'))
30          model.add(LeakyReLU(alpha=0.2))
31          model.add(BatchNormalization(momentum=0.8))
32
33          model.add(Conv2DTranspose(64, kernel_size=4, strides
                =2, padding='same'))
34          model.add(LeakyReLU(alpha=0.2))
35          model.add(BatchNormalization(momentum=0.8))
36
37          model.add(Conv2D(1, kernel_size=7, activation='tanh',
                 padding='same'))
38
39          return model
40
41      # input 28*28*1
42      # output 1
43
44      def build_discriminator():
45          model = Sequential()
46
47          model.add(Conv2D(64, kernel_size=3, strides=2,
                input_shape=(28, 28, 1), padding='same'))
48          model.add(LeakyReLU(alpha=0.2))
49          model.add(Dropout(0.25))
50
51          model.add(Conv2D(128, kernel_size=3, strides=2,
                padding='same'))
52          model.add(LeakyReLU(alpha=0.2))
53          model.add(Dropout(0.25))
54
55          model.add(Flatten())
56          model.add(Dense(1, activation='sigmoid'))
57
58          return model
59
60      (x_train, _), (_, _) = tf.keras.datasets.mnist.load_data
            ()
61      x_train = (x_train - 127.5) / 127.5
```

```
62      x_train = np.expand_dims(x_train, axis=3)
63
64      def calculate_fid(real_images, fake_images):
65          act1 = real_images.reshape((real_images.shape[0], -1)
                )
66          mu1, sigma1 = act1.mean(axis=0), np.cov(act1, rowvar=
                False)
67
68          act2 = fake_images.reshape((fake_images.shape[0], -1)
                )
69          mu2, sigma2 = act2.mean(axis=0), np.cov(act2, rowvar=
                False)
70
71          ssdiff = np.sum((mu1 - mu2)**2.0)
72          covmean = sqrtm(sigma1.dot(sigma2))
73
74          if np.iscomplexobj(covmean):
75              covmean = covmean.real
76
77          fid = ssdiff + np.trace(sigma1 + sigma2 - 2.0 *
                covmean)
78
79          return fid
80
81      def train_gan(epochs=1000, batch_size=64, p_epoch=100):
82          generator = build_generator()
83          discriminator = build_discriminator()
84
85          discriminator.compile(loss='binary_crossentropy',
                optimizer=Adam(0.0002, 0.5), metrics=['accuracy'])
86          discriminator.trainable = False
87
88          gan_input = tf.keras.Input(shape=(100,))
89          gan_output = discriminator(generator(gan_input))
90          gan = tf.keras.Model(gan_input, gan_output)
91          gan.compile(loss='binary_crossentropy', optimizer=
                Adam(0.0002, 0.5))
92
93          half_batch = int(batch_size / 2)
```

```
94
95          d_losses = []
96          g_losses = []
97          d_acc = []
98          fid_scores = []
99
100         start_time = time.time()  # Record the start time
101
102         for epoch in range(epochs):
103             # Select a random half batch of real images
104             idx = np.random.randint(0, x_train.shape[0],
                    half_batch)
105             real_images = x_train[idx]
106
107             # Generate a half batch of new fake images
108             noise = np.random.normal(0, 1, (half_batch, 100))
109             fake_images = generator.predict(noise)
110
111             # Train the discriminator
112             real_labels = np.ones((half_batch, 1))
113             fake_labels = np.zeros((half_batch, 1))
114
115             d_loss_real = discriminator.train_on_batch(
                    real_images, real_labels)
116             d_loss_fake = discriminator.train_on_batch(
                    fake_images, fake_labels)
117             d_loss = 0.5 * np.add(d_loss_real, d_loss_fake)
118
119             # Train the generator
120             noise = np.random.normal(0, 1, (batch_size, 100))
121             valid_y = np.ones((batch_size, 1))
122
123             g_loss = gan.train_on_batch(noise, valid_y)
124
125             # Record the losses
126             d_losses.append(d_loss[0])
127             g_losses.append(g_loss)
128             d_acc.append(d_loss[1] * 100)
129
```

```
            # Calculate FID every p_epoch epochs
            if epoch % p_epoch == 0:
                noise = np.random.normal(0, 1, (1000, 100))
                fake_images = generator.predict(noise)
                fid = calculate_fid(x_train[:1000],
                    fake_images)
                fid_scores.append(fid)
                print(f"{epoch} [D loss: {d_loss[0]}, acc.: 
                    {100 * d_loss[1]}%] [G loss: {g_loss}] [
                    FID: {fid}]")

        end_time = time.time()  # Record the end time
        total_time = end_time - start_time
        print(f"Total training time: {total_time:.2f} seconds
            ")
        return generator, d_losses, g_losses, d_acc,
            fid_scores
```

Listing A.7: Explore GAN with more convolutional layers 2

```
import tensorflow as tf
from tensorflow.keras.layers import Dense, Reshape,
    Flatten, Dropout, LeakyReLU, Conv2D, Conv2DTranspose,
    BatchNormalization
from tensorflow.keras.models import Sequential
from tensorflow.keras.optimizers import Adam
from scipy.linalg import sqrtm
import numpy as np
import time
import matplotlib.pyplot as plt

np.random.seed(1000)
tf.random.set_seed(1000)

# input 100
# output 28*28*1

def build_generator():
    model = Sequential()

```

```
19          model.add(Dense(7*7*128, input_dim=100))
20          model.add(LeakyReLU(alpha=0.2))
21          model.add(Reshape((7, 7, 128)))
22          model.add(BatchNormalization(momentum=0.8))
23
24          # add the first convolution layer
25          model.add(Conv2D(128, kernel_size=3, strides=1,
                padding='same'))
26          model.add(LeakyReLU(alpha=0.2))
27          model.add(BatchNormalization(momentum=0.8))
28
29          # add the 2nd convolution layer
30          model.add(Conv2D(128, kernel_size=3, strides=1,
                padding='same'))
31          model.add(LeakyReLU(alpha=0.2))
32          model.add(BatchNormalization(momentum=0.8))
33
34          model.add(Conv2DTranspose(128, kernel_size=4, strides
                =2, padding='same'))
35          model.add(LeakyReLU(alpha=0.2))
36          model.add(BatchNormalization(momentum=0.8))
37
38          model.add(Conv2DTranspose(64, kernel_size=4, strides
                =2, padding='same'))
39          model.add(LeakyReLU(alpha=0.2))
40          model.add(BatchNormalization(momentum=0.8))
41
42          model.add(Conv2D(1, kernel_size=7, activation='tanh',
                 padding='same'))
43
44          return model
45
46      # input 28*28*1
47      # output 1
48
49      def build_discriminator():
50          model = Sequential()
51
52          model.add(Conv2D(64, kernel_size=3, strides=2,
```

```
            input_shape=(28, 28, 1), padding='same'))
        model.add(LeakyReLU(alpha=0.2))
        model.add(Dropout(0.25))

        model.add(Conv2D(128, kernel_size=3, strides=2,
            padding='same'))
        model.add(LeakyReLU(alpha=0.2))
        model.add(Dropout(0.25))

        model.add(Flatten())
        model.add(Dense(1, activation='sigmoid'))

        return model

    (x_train, _), (_, _) = tf.keras.datasets.mnist.load_data
        ()
    x_train = (x_train - 127.5) / 127.5
    x_train = np.expand_dims(x_train, axis=3)

    def calculate_fid(real_images, fake_images):
        act1 = real_images.reshape((real_images.shape[0], -1)
            )
        mu1, sigma1 = act1.mean(axis=0), np.cov(act1, rowvar=
            False)

        act2 = fake_images.reshape((fake_images.shape[0], -1)
            )
        mu2, sigma2 = act2.mean(axis=0), np.cov(act2, rowvar=
            False)

        ssdiff = np.sum((mu1 - mu2)**2.0)
        covmean = sqrtm(sigma1.dot(sigma2))

        if np.iscomplexobj(covmean):
            covmean = covmean.real

        fid = ssdiff + np.trace(sigma1 + sigma2 - 2.0 *
            covmean)

```

```
84          return fid
85
86      def train_gan(epochs=1000, batch_size=64, p_epoch=100):
87          generator = build_generator()
88          discriminator = build_discriminator()
89
90          discriminator.compile(loss='binary_crossentropy',
                optimizer=Adam(0.0002, 0.5), metrics=['accuracy'])
91          discriminator.trainable = False
92
93          gan_input = tf.keras.Input(shape=(100,))
94          gan_output = discriminator(generator(gan_input))
95          gan = tf.keras.Model(gan_input, gan_output)
96          gan.compile(loss='binary_crossentropy', optimizer=
                Adam(0.0002, 0.5))
97
98          half_batch = int(batch_size / 2)
99
100         d_losses = []
101         g_losses = []
102         d_acc = []
103         fid_scores = []
104
105         start_time = time.time()  # Record the start time
106
107         for epoch in range(epochs):
108             # Select a random half batch of real images
109             idx = np.random.randint(0, x_train.shape[0],
                    half_batch)
110             real_images = x_train[idx]
111
112             # Generate a half batch of new fake images
113             noise = np.random.normal(0, 1, (half_batch, 100))
114             fake_images = generator.predict(noise)
115
116             # Train the discriminator
117             real_labels = np.ones((half_batch, 1))
118             fake_labels = np.zeros((half_batch, 1))
119
```

```
            d_loss_real = discriminator.train_on_batch(
                real_images, real_labels)
            d_loss_fake = discriminator.train_on_batch(
                fake_images, fake_labels)
            d_loss = 0.5 * np.add(d_loss_real, d_loss_fake)

            # Train the generator
            noise = np.random.normal(0, 1, (batch_size, 100))
            valid_y = np.ones((batch_size, 1))

            g_loss = gan.train_on_batch(noise, valid_y)

            # Record the losses
            d_losses.append(d_loss[0])
            g_losses.append(g_loss)
            d_acc.append(d_loss[1] * 100)

            # Calculate FID every p_epoch epochs
            if epoch % p_epoch == 0:
                noise = np.random.normal(0, 1, (1000, 100))
                fake_images = generator.predict(noise)
                fid = calculate_fid(x_train[:1000],
                    fake_images)
                fid_scores.append(fid)
                print(f"{epoch} [D loss: {d_loss[0]}, acc.: 
                    {100 * d_loss[1]}%] [G loss: {g_loss}] [
                    FID: {fid}]")

        end_time = time.time()  # Record the end time
        total_time = end_time - start_time
        print(f"Total training time: {total_time:.2f} seconds
            ")
        return generator, d_losses, g_losses, d_acc,
            fid_scores
```

Listing A.8: Explore GAN with more convolutional layers 3

```
    import tensorflow as tf
    from tensorflow.keras.layers import Dense, Reshape,
        Flatten, Dropout, LeakyReLU, Conv2D, Conv2DTranspose,
```

```
    BatchNormalization
from tensorflow.keras.models import Sequential
from tensorflow.keras.optimizers import Adam
from scipy.linalg import sqrtm
import numpy as np
import time
import matplotlib.pyplot as plt

np.random.seed(1000)
tf.random.set_seed(1000)

# input 100
# output 28*28*1

def build_generator():
    model = Sequential()

    model.add(Dense(7*7*128, input_dim=100))
    model.add(LeakyReLU(alpha=0.2))
    model.add(Reshape((7, 7, 128)))
    model.add(BatchNormalization(momentum=0.8))

    # add the 1st convolution layer
    model.add(Conv2D(128, kernel_size=3, strides=1,
        padding='same'))
    model.add(LeakyReLU(alpha=0.2))
    model.add(BatchNormalization(momentum=0.8))

    # add the 2nd convolution layer
    model.add(Conv2D(128, kernel_size=3, strides=1,
        padding='same'))
    model.add(LeakyReLU(alpha=0.2))
    model.add(BatchNormalization(momentum=0.8))

    # add the 3rd convolution layer
    model.add(Conv2D(128, kernel_size=3, strides=1,
        padding='same'))
    model.add(LeakyReLU(alpha=0.2))
    model.add(BatchNormalization(momentum=0.8))
```

```
38
39          model.add(Conv2DTranspose(128, kernel_size=4, strides
                =2, padding='same'))
40          model.add(LeakyReLU(alpha=0.2))
41          model.add(BatchNormalization(momentum=0.8))
42
43          model.add(Conv2DTranspose(64, kernel_size=4, strides
                =2, padding='same'))
44          model.add(LeakyReLU(alpha=0.2))
45          model.add(BatchNormalization(momentum=0.8))
46
47          model.add(Conv2D(1, kernel_size=7, activation='tanh',
                 padding='same'))
48
49          return model
50
51      # input 28*28*1
52      # output 1
53
54      def build_discriminator():
55          model = Sequential()
56
57          model.add(Conv2D(64, kernel_size=3, strides=2,
                input_shape=(28, 28, 1), padding='same'))
58          model.add(LeakyReLU(alpha=0.2))
59          model.add(Dropout(0.25))
60
61          model.add(Conv2D(128, kernel_size=3, strides=2,
                padding='same'))
62          model.add(LeakyReLU(alpha=0.2))
63          model.add(Dropout(0.25))
64
65          model.add(Flatten())
66          model.add(Dense(1, activation='sigmoid'))
67
68          return model
69
70      (x_train, _), (_, _) = tf.keras.datasets.mnist.load_data
            ()
```

```
    x_train = (x_train - 127.5) / 127.5
    x_train = np.expand_dims(x_train, axis=3)

    def calculate_fid(real_images, fake_images):
        act1 = real_images.reshape((real_images.shape[0], -1)
            )
        mu1, sigma1 = act1.mean(axis=0), np.cov(act1, rowvar=
            False)

        act2 = fake_images.reshape((fake_images.shape[0], -1)
            )
        mu2, sigma2 = act2.mean(axis=0), np.cov(act2, rowvar=
            False)

        ssdiff = np.sum((mu1 - mu2)**2.0)
        covmean = sqrtm(sigma1.dot(sigma2))

        if np.iscomplexobj(covmean):
            covmean = covmean.real

        fid = ssdiff + np.trace(sigma1 + sigma2 - 2.0 *
            covmean)

        return fid

    def train_gan(epochs=1000, batch_size=64, p_epoch=100):
        generator = build_generator()
        discriminator = build_discriminator()

        discriminator.compile(loss='binary_crossentropy',
            optimizer=Adam(0.0002, 0.5), metrics=['accuracy'])
        discriminator.trainable = False

        gan_input = tf.keras.Input(shape=(100,))
        gan_output = discriminator(generator(gan_input))
        gan = tf.keras.Model(gan_input, gan_output)
        gan.compile(loss='binary_crossentropy', optimizer=
            Adam(0.0002, 0.5))

```

```
        half_batch = int(batch_size / 2)

        d_losses = []
        g_losses = []
        d_acc = []
        fid_scores = []

        start_time = time.time()  # Record the start time

        for epoch in range(epochs):
            # Select a random half batch of real images
            idx = np.random.randint(0, x_train.shape[0],
                half_batch)
            real_images = x_train[idx]

            # Generate a half batch of new fake images
            noise = np.random.normal(0, 1, (half_batch, 100))
            fake_images = generator.predict(noise)

            # Train the discriminator
            real_labels = np.ones((half_batch, 1))
            fake_labels = np.zeros((half_batch, 1))

            d_loss_real = discriminator.train_on_batch(
                real_images, real_labels)
            d_loss_fake = discriminator.train_on_batch(
                fake_images, fake_labels)
            d_loss = 0.5 * np.add(d_loss_real, d_loss_fake)

            # Train the generator
            noise = np.random.normal(0, 1, (batch_size, 100))
            valid_y = np.ones((batch_size, 1))

            g_loss = gan.train_on_batch(noise, valid_y)

            # Record the losses
            d_losses.append(d_loss[0])
            g_losses.append(g_loss)
            d_acc.append(d_loss[1] * 100)
```

```
139
140             # Calculate FID every p_epoch epochs
141             if epoch % p_epoch == 0:
142                 noise = np.random.normal(0, 1, (1000, 100))
143                 fake_images = generator.predict(noise)
144                 fid = calculate_fid(x_train[:1000],
                        fake_images)
145                 fid_scores.append(fid)
146                 print(f"{epoch} [D loss: {d_loss[0]}, acc.: 
                        {100 * d_loss[1]}%] [G loss: {g_loss}] [
                        FID: {fid}]")
147
148         end_time = time.time()  # Record the end time
149         total_time = end_time - start_time
150         print(f"Total training time: {total_time:.2f} seconds
                ")
151         return generator, d_losses, g_losses, d_acc,
                fid_scores
```

Listing A.9: Explore GAN with more convolutional layers 4

```
1   import tensorflow as tf
2   from tensorflow.keras.layers import Dense, Reshape,
        Flatten, Dropout, LeakyReLU, Conv2D, Conv2DTranspose,
        BatchNormalization
3   from tensorflow.keras.models import Sequential
4   from tensorflow.keras.optimizers import Adam
5   from scipy.linalg import sqrtm
6   import numpy as np
7   import time
8   import matplotlib.pyplot as plt
9
10  np.random.seed(1000)
11  tf.random.set_seed(1000)
12
13  # input 100
14  # output 28*28*1
15
16  def build_generator():
17      model = Sequential()
```

```
18
19          model.add(Dense(7*7*128, input_dim=100))
20          model.add(LeakyReLU(alpha=0.2))
21          model.add(Reshape((7, 7, 128)))
22          model.add(BatchNormalization(momentum=0.8))
23
24          # add the 1st convolution layer
25          model.add(Conv2D(128, kernel_size=3, strides=1,
                padding='same'))
26          model.add(LeakyReLU(alpha=0.2))
27          model.add(BatchNormalization(momentum=0.8))
28
29          # add the 2nd convolution layer
30          model.add(Conv2D(128, kernel_size=3, strides=1,
                padding='same'))
31          model.add(LeakyReLU(alpha=0.2))
32          model.add(BatchNormalization(momentum=0.8))
33
34          # add the 3rd convolution layer
35          model.add(Conv2D(128, kernel_size=3, strides=1,
                padding='same'))
36          model.add(LeakyReLU(alpha=0.2))
37          model.add(BatchNormalization(momentum=0.8))
38
39          model.add(Conv2DTranspose(128, kernel_size=4, strides
                =2, padding='same'))
40          model.add(LeakyReLU(alpha=0.2))
41          model.add(BatchNormalization(momentum=0.8))
42
43          model.add(Conv2DTranspose(64, kernel_size=4, strides
                =2, padding='same'))
44          model.add(LeakyReLU(alpha=0.2))
45          model.add(BatchNormalization(momentum=0.8))
46
47          model.add(Conv2D(1, kernel_size=7, activation='tanh',
                 padding='same'))
48
49          return model
50
```

```
    # input 28*28*1
    # output 1

    def build_discriminator():
        model = Sequential()

        model.add(Conv2D(64, kernel_size=3, strides=2,
            input_shape=(28, 28, 1), padding='same'))
        model.add(LeakyReLU(alpha=0.2))
        model.add(Dropout(0.25))

        model.add(Conv2D(128, kernel_size=3, strides=2,
            padding='same'))
        model.add(LeakyReLU(alpha=0.2))
        model.add(Dropout(0.25))

        # add the 1st convolution layer
        model.add(Conv2D(256, kernel_size=3, strides=2,
            padding='same'))
        model.add(LeakyReLU(alpha=0.2))
        model.add(Dropout(0.25))

        model.add(Flatten())
        model.add(Dense(1, activation='sigmoid'))

        return model

    (x_train, _), (_, _) = tf.keras.datasets.mnist.load_data
        ()
    x_train = (x_train - 127.5) / 127.5
    x_train = np.expand_dims(x_train, axis=3)

    def calculate_fid(real_images, fake_images):
        act1 = real_images.reshape((real_images.shape[0], -1)
            )
        mu1, sigma1 = act1.mean(axis=0), np.cov(act1, rowvar=
            False)

        act2 = fake_images.reshape((fake_images.shape[0], -1)
```

```
            )
        mu2, sigma2 = act2.mean(axis=0), np.cov(act2, rowvar=
            False)

        ssdiff = np.sum((mu1 - mu2)**2.0)
        covmean = sqrtm(sigma1.dot(sigma2))

        if np.iscomplexobj(covmean):
            covmean = covmean.real

        fid = ssdiff + np.trace(sigma1 + sigma2 - 2.0 *
            covmean)

        return fid

    def train_gan(epochs=1000, batch_size=64, p_epoch=100):
        generator = build_generator()
        discriminator = build_discriminator()

        discriminator.compile(loss='binary_crossentropy',
            optimizer=Adam(0.0002, 0.5), metrics=['accuracy'])
        discriminator.trainable = False

        gan_input = tf.keras.Input(shape=(100,))
        gan_output = discriminator(generator(gan_input))
        gan = tf.keras.Model(gan_input, gan_output)
        gan.compile(loss='binary_crossentropy', optimizer=
            Adam(0.0002, 0.5))

        half_batch = int(batch_size / 2)

        d_losses = []
        g_losses = []
        d_acc = []
        fid_scores = []

        start_time = time.time()  # Record the start time

        for epoch in range(epochs):
```

```
            # Select a random half batch of real images
            idx = np.random.randint(0, x_train.shape[0],
                half_batch)
            real_images = x_train[idx]

            # Generate a half batch of new fake images
            noise = np.random.normal(0, 1, (half_batch, 100))
            fake_images = generator.predict(noise)

            # Train the discriminator
            real_labels = np.ones((half_batch, 1))
            fake_labels = np.zeros((half_batch, 1))

            d_loss_real = discriminator.train_on_batch(
                real_images, real_labels)
            d_loss_fake = discriminator.train_on_batch(
                fake_images, fake_labels)
            d_loss = 0.5 * np.add(d_loss_real, d_loss_fake)

            # Train the generator
            noise = np.random.normal(0, 1, (batch_size, 100))
            valid_y = np.ones((batch_size, 1))

            g_loss = gan.train_on_batch(noise, valid_y)

            # Record the losses
            d_losses.append(d_loss[0])
            g_losses.append(g_loss)
            d_acc.append(d_loss[1] * 100)

            # Calculate FID every p_epoch epochs
            if epoch % p_epoch == 0:
                noise = np.random.normal(0, 1, (1000, 100))
                fake_images = generator.predict(noise)
                fid = calculate_fid(x_train[:1000],
                    fake_images)
                fid_scores.append(fid)
                print(f"{epoch} [D loss: {d_loss[0]}, acc.: 
                    {100 * d_loss[1]}%] [G loss: {g_loss}] [
```

```
                    FID: {fid}]")

        end_time = time.time()  # Record the end time
        total_time = end_time - start_time
        print(f"Total training time: {total_time:.2f} seconds
            ")
        return generator, d_losses, g_losses, d_acc,
            fid_scores
```

Listing A.10: Explore GAN with more convolutional layers 5

```
import tensorflow as tf
from tensorflow.keras.layers import Dense, Reshape,
    Flatten, Dropout, LeakyReLU, Conv2D, Conv2DTranspose,
    BatchNormalization
from tensorflow.keras.models import Sequential
from tensorflow.keras.optimizers import Adam
from scipy.linalg import sqrtm
import numpy as np
import time
import matplotlib.pyplot as plt

np.random.seed(1000)
tf.random.set_seed(1000)

# input 100
# output 28*28*1

def build_generator():
    model = Sequential()

    model.add(Dense(7*7*128, input_dim=100))
    model.add(LeakyReLU(alpha=0.2))
    model.add(Reshape((7, 7, 128)))
    model.add(BatchNormalization(momentum=0.8))

    # add the 1st convolution layer
    model.add(Conv2D(128, kernel_size=3, strides=1,
        padding='same'))
    model.add(LeakyReLU(alpha=0.2))
```

```
27          model.add(BatchNormalization(momentum=0.8))
28
29          # add the 2nd convolution layer
30          model.add(Conv2D(128, kernel_size=3, strides=1,
                padding='same'))
31          model.add(LeakyReLU(alpha=0.2))
32          model.add(BatchNormalization(momentum=0.8))
33
34          # add the 3rd convolution layer
35          model.add(Conv2D(128, kernel_size=3, strides=1,
                padding='same'))
36          model.add(LeakyReLU(alpha=0.2))
37          model.add(BatchNormalization(momentum=0.8))
38
39          model.add(Conv2DTranspose(128, kernel_size=4, strides
                =2, padding='same'))
40          model.add(LeakyReLU(alpha=0.2))
41          model.add(BatchNormalization(momentum=0.8))
42
43          model.add(Conv2DTranspose(64, kernel_size=4, strides
                =2, padding='same'))
44          model.add(LeakyReLU(alpha=0.2))
45          model.add(BatchNormalization(momentum=0.8))
46
47          model.add(Conv2D(1, kernel_size=7, activation='tanh',
                 padding='same'))
48
49          return model
50
51      # input 28*28*1
52      # output 1
53
54      def build_discriminator():
55          model = Sequential()
56
57          model.add(Conv2D(64, kernel_size=3, strides=2,
                input_shape=(28, 28, 1), padding='same'))
58          model.add(LeakyReLU(alpha=0.2))
59          model.add(Dropout(0.25))
```

```
60
61          model.add(Conv2D(128, kernel_size=3, strides=2,
                padding='same'))
62          model.add(LeakyReLU(alpha=0.2))
63          model.add(Dropout(0.25))
64
65          model.add(Conv2D(256, kernel_size=3, strides=2,
                padding='same'))
66          model.add(LeakyReLU(alpha=0.2))
67          model.add(Dropout(0.25))
68
69          # add the 1st convolution layer
70          model.add(Conv2D(512, kernel_size=3, strides=2,
                padding='same'))
71          model.add(LeakyReLU(alpha=0.2))
72          model.add(Dropout(0.25))
73
74          # add the 2nd convolution layer
75          model.add(Conv2D(1024, kernel_size=3, strides=2,
                padding='same'))
76          model.add(LeakyReLU(alpha=0.2))
77          model.add(Dropout(0.25))
78
79          model.add(Flatten())
80          model.add(Dense(1, activation='sigmoid'))
81
82          return model
83
84      (x_train, _), (_, _) = tf.keras.datasets.mnist.load_data
            ()
85      x_train = (x_train - 127.5) / 127.5
86      x_train = np.expand_dims(x_train, axis=3)
87
88      def calculate_fid(real_images, fake_images):
89          act1 = real_images.reshape((real_images.shape[0], -1)
                )
90          mu1, sigma1 = act1.mean(axis=0), np.cov(act1, rowvar=
                False)
91
```

```
92          act2 = fake_images.reshape((fake_images.shape[0], -1)
                )
93          mu2, sigma2 = act2.mean(axis=0), np.cov(act2, rowvar=
                False)
94
95          ssdiff = np.sum((mu1 - mu2)**2.0)
96          covmean = sqrtm(sigma1.dot(sigma2))
97
98          if np.iscomplexobj(covmean):
99              covmean = covmean.real
100
101         fid = ssdiff + np.trace(sigma1 + sigma2 - 2.0 *
                covmean)
102
103         return fid
104
105     def train_gan(epochs=1000, batch_size=64, p_epoch=100):
106         generator = build_generator()
107         discriminator = build_discriminator()
108
109         discriminator.compile(loss='binary_crossentropy',
                optimizer=Adam(0.0002, 0.5), metrics=['accuracy'])
110         discriminator.trainable = False
111
112         gan_input = tf.keras.Input(shape=(100,))
113         gan_output = discriminator(generator(gan_input))
114         gan = tf.keras.Model(gan_input, gan_output)
115         gan.compile(loss='binary_crossentropy', optimizer=
                Adam(0.0002, 0.5))
116
117         half_batch = int(batch_size / 2)
118
119         d_losses = []
120         g_losses = []
121         d_acc = []
122         fid_scores = []
123
124         start_time = time.time()  # Record the start time
125
```

```
126         for epoch in range(epochs):
127             # Select a random half batch of real images
128             idx = np.random.randint(0, x_train.shape[0],
                    half_batch)
129             real_images = x_train[idx]
130 
131             # Generate a half batch of new fake images
132             noise = np.random.normal(0, 1, (half_batch, 100))
133             fake_images = generator.predict(noise)
134 
135             # Train the discriminator
136             real_labels = np.ones((half_batch, 1))
137             fake_labels = np.zeros((half_batch, 1))
138 
139             d_loss_real = discriminator.train_on_batch(
                    real_images, real_labels)
140             d_loss_fake = discriminator.train_on_batch(
                    fake_images, fake_labels)
141             d_loss = 0.5 * np.add(d_loss_real, d_loss_fake)
142 
143             # Train the generator
144             noise = np.random.normal(0, 1, (batch_size, 100))
145             valid_y = np.ones((batch_size, 1))
146 
147             g_loss = gan.train_on_batch(noise, valid_y)
148 
149             # Record the losses
150             d_losses.append(d_loss[0])
151             g_losses.append(g_loss)
152             d_acc.append(d_loss[1] * 100)
153 
154             # Calculate FID every p_epoch epochs
155             if epoch % p_epoch == 0:
156                 noise = np.random.normal(0, 1, (1000, 100))
157                 fake_images = generator.predict(noise)
158                 fid = calculate_fid(x_train[:1000],
                        fake_images)
159                 fid_scores.append(fid)
160                 print(f"{epoch} [D loss: {d_loss[0]}, acc.: 
```

```
                    {100 * d_loss[1]}%] [G loss: {g_loss}] [
                    FID: {fid}]")

        end_time = time.time()  # Record the end time
        total_time = end_time - start_time
        print(f"Total training time: {total_time:.2f} seconds
            ")
        return generator, d_losses, g_losses, d_acc,
            fid_scores
```

Listing A.11: Explore GAN with more convolutional layers 6

```
def build_generator():
model = Sequential()

model.add(Dense(7*7*128, input_dim=100))
model.add(LeakyReLU(alpha=0.2))
model.add(Reshape((7, 7, 128)))
model.add(BatchNormalization(momentum=0.8))

# add the 1st convolution layer
model.add(Conv2D(128, kernel_size=3, strides=1, padding='
    same'))
model.add(LeakyReLU(alpha=0.2))
model.add(BatchNormalization(momentum=0.8))

# add the 2nd convolution layer
model.add(Conv2D(128, kernel_size=3, strides=1, padding='
    same'))
model.add(LeakyReLU(alpha=0.2))
model.add(BatchNormalization(momentum=0.8))

# add the 3rd convolution layer
model.add(Conv2D(128, kernel_size=3, strides=1, padding='
    same'))
model.add(LeakyReLU(alpha=0.2))
model.add(BatchNormalization(momentum=0.8))

model.add(Conv2DTranspose(128, kernel_size=4, strides=2,
    padding='same'))
```

```
25      model.add(LeakyReLU(alpha=0.2))
26      model.add(BatchNormalization(momentum=0.8))
27
28      model.add(Conv2DTranspose(64, kernel_size=4, strides=2,
            padding='same'))
29      model.add(LeakyReLU(alpha=0.2))
30      model.add(BatchNormalization(momentum=0.8))
31
32      model.add(Conv2D(1, kernel_size=7, activation='tanh',
            padding='same'))
33
34      return model
35
36  def build_discriminator():
37      model = Sequential()
38
39      model.add(Conv2D(64, kernel_size=3, strides=2,
            input_shape=(28, 28, 1), padding='same'))
40      model.add(LeakyReLU(alpha=0.2))
41      model.add(Dropout(0.25))
42
43      model.add(Conv2D(128, kernel_size=3, strides=2, padding='
            same'))
44      model.add(LeakyReLU(alpha=0.2))
45      model.add(Dropout(0.25))
46
47      model.add(Conv2D(256, kernel_size=3, strides=2, padding='
            same'))
48      model.add(LeakyReLU(alpha=0.2))
49      model.add(Dropout(0.25))
50
51     # add the 1st convolution layer
52      model.add(Conv2D(512, kernel_size=3, strides=2, padding='
            same'))
53      model.add(LeakyReLU(alpha=0.2))
54      model.add(Dropout(0.25))
55
56      # add the 2nd convolution layer
57      model.add(Conv2D(1024, kernel_size=3, strides=2, padding=
```

```
        'same'))
    model.add(LeakyReLU(alpha=0.2))
    model.add(Dropout(0.25))

    # add the 3rd convolution layer
    model.add(Conv2D(2048, kernel_size=3, strides=2, padding=
        'same'))
    model.add(LeakyReLU(alpha=0.2))
    model.add(Dropout(0.25))

    model.add(Flatten())
    model.add(Dense(1, activation='sigmoid'))

    return model




import tensorflow as tf
from tensorflow.keras.layers import Dense, Reshape, Flatten,
    Dropout, LeakyReLU, Conv2D, Conv2DTranspose,
    BatchNormalization
from tensorflow.keras.models import Sequential
from tensorflow.keras.optimizers import Adam
from scipy.linalg import sqrtm
import numpy as np
import time
import matplotlib.pyplot as plt

np.random.seed(1000)
tf.random.set_seed(1000)

(x_train, _), (_, _) = tf.keras.datasets.mnist.load_data()
x_train = (x_train - 127.5) / 127.5
x_train = np.expand_dims(x_train, axis=3)

def calculate_fid(real_images, fake_images):
    act1 = real_images.reshape((real_images.shape[0], -1))
    mu1, sigma1 = act1.mean(axis=0), np.cov(act1, rowvar=
```

```
        False)

    act2 = fake_images.reshape((fake_images.shape[0], -1))
    mu2, sigma2 = act2.mean(axis=0), np.cov(act2, rowvar=
        False)

    ssdiff = np.sum((mu1 - mu2)**2.0)
    covmean = sqrtm(sigma1.dot(sigma2))

    if np.iscomplexobj(covmean):
        covmean = covmean.real

    fid = ssdiff + np.trace(sigma1 + sigma2 - 2.0 * covmean)

    return fid

def train_gan(epochs=1000, batch_size=64, p_epoch=100):
    generator = build_generator()
    discriminator = build_discriminator()

    discriminator.compile(loss='binary_crossentropy',
        optimizer=Adam(0.0002, 0.5), metrics=['accuracy'])
    discriminator.trainable = False

    gan_input = tf.keras.Input(shape=(100,))
    gan_output = discriminator(generator(gan_input))
    gan = tf.keras.Model(gan_input, gan_output)
    gan.compile(loss='binary_crossentropy', optimizer=Adam
        (0.0002, 0.5))

    half_batch = int(batch_size / 2)

    d_losses = []
    g_losses = []
    d_acc = []
    fid_scores = []

    start_time = time.time()  # Record the start time

```

```
    for epoch in range(epochs):
        # Select a random half batch of real images
        idx = np.random.randint(0, x_train.shape[0],
            half_batch)
        real_images = x_train[idx]

        # Generate a half batch of new fake images
        noise = np.random.normal(0, 1, (half_batch, 100))
        fake_images = generator.predict(noise)

        # Train the discriminator
        real_labels = np.ones((half_batch, 1))
        fake_labels = np.zeros((half_batch, 1))

        d_loss_real = discriminator.train_on_batch(
            real_images, real_labels)
        d_loss_fake = discriminator.train_on_batch(
            fake_images, fake_labels)
        d_loss = 0.5 * np.add(d_loss_real, d_loss_fake)

        # Train the generator
        noise = np.random.normal(0, 1, (batch_size, 100))
        valid_y = np.ones((batch_size, 1))

        g_loss = gan.train_on_batch(noise, valid_y)

        # Record the losses
        d_losses.append(d_loss[0])
        g_losses.append(g_loss)
        d_acc.append(d_loss[1] * 100)

        # Calculate FID every p_epoch epochs
        if epoch % p_epoch == 0:
            noise = np.random.normal(0, 1, (1000, 100))
            fake_images = generator.predict(noise)
            fid = calculate_fid(x_train[:1000], fake_images)
            fid_scores.append(fid)
            print(f"{epoch} [D loss: {d_loss[0]}, acc.: {100
                *d_loss[1]}%] [G loss: {g_loss}] [FID: {fid}]
```

```
                ")
163
164     end_time = time.time()  # Record the end time
165     total_time = end_time - start_time
166     print(f"Total training time: {total_time:.2f} seconds")
167     return generator, d_losses, g_losses, d_acc, fid_scores
```

Listing A.12: Apply new dataset

```
1
2     import os
3     import time
4     import numpy as np
5     import tensorflow as tf
6     import matplotlib.pyplot as plt
7     from tensorflow.keras.models import Sequential, Model
8     from tensorflow.keras.layers import Dense, Reshape,
          BatchNormalization, LeakyReLU, Conv2D, Conv2DTranspose
          , Flatten, Dropout, Input
9     from tensorflow.keras.optimizers import Adam
10    from tensorflow.keras.preprocessing.image import load_img
          , img_to_array
11
12    def load_images_as_rgb_matrices(path, size):
13    images = []
14    for img_name in os.listdir(path):
15        img_path = os.path.join(path, img_name)
16        img = load_img(img_path, target_size=size)
17        img_array = img_to_array(img)
18        images.append(img_array)
19    images = np.array(images)
20    images = (images - 127.5) / 127.5  # Normalize images to
          [-1, 1]
21    return images
22
23    from google.colab import drive
24
25    # mount Google Drive
26    drive.mount('/content/drive')
27
```

```
28      cat_path = "/content/drive/My␣Drive/gan/afhq/train/cat"
29      size = (128, 128)
30      x_train = load_images_as_rgb_matrices(cat_path, size)
31
32      # generator
33      # input 100
34      # output 128*128*3
35
36      def build_generator():
37          model = Sequential()
38
39          # Increase the dimension
40          model.add(Dense(16*16*256, input_dim=100))
41          model.add(LeakyReLU(alpha=0.2))
42          model.add(Reshape((16, 16, 256)))
43          model.add(BatchNormalization(momentum=0.8))
44
45          model.add(Conv2DTranspose(256, kernel_size=4, strides
                =2, padding='same'))  # 32x32
46          model.add(LeakyReLU(alpha=0.2))
47          model.add(BatchNormalization(momentum=0.8))
48
49          model.add(Conv2DTranspose(128, kernel_size=4, strides
                =2, padding='same'))  # 64x64
50          model.add(LeakyReLU(alpha=0.2))
51          model.add(BatchNormalization(momentum=0.8))
52
53          model.add(Conv2DTranspose(64, kernel_size=4, strides
                =2, padding='same'))  # 128x128
54          model.add(LeakyReLU(alpha=0.2))
55          model.add(BatchNormalization(momentum=0.8))
56
57          model.add(Conv2D(3, kernel_size=7, activation='tanh',
                padding='same'))  # 128x128x3
58
59      return model
60
61      # discriminator
62      # input 128*128*3
```

```
    # output 1

    def build_discriminator():
        model = Sequential()

        model.add(Conv2D(64, kernel_size=3, strides=2,
            input_shape=(128, 128, 3), padding='same'))
        model.add(LeakyReLU(alpha=0.2))
        model.add(Dropout(0.25))

        model.add(Conv2D(128, kernel_size=3, strides=2,
            padding='same'))
        model.add(LeakyReLU(alpha=0.2))
        model.add(Dropout(0.25))

        model.add(Conv2D(256, kernel_size=3, strides=2,
            padding='same'))
        model.add(LeakyReLU(alpha=0.2))
        model.add(Dropout(0.25))

        model.add(Conv2D(512, kernel_size=3, strides=2,
            padding='same'))
        model.add(LeakyReLU(alpha=0.2))
        model.add(Dropout(0.25))

        model.add(Flatten())
        model.add(Dense(1, activation='sigmoid'))

    return model

    # initial GAN
    def build_gan(generator, discriminator):
    discriminator.trainable = False
    gan_input = Input(shape=(100,))
    x = generator(gan_input)
    gan_output = discriminator(x)
    gan = Model(gan_input, gan_output)
    gan.compile(loss='binary_crossentropy', optimizer=tf.
        keras.optimizers.legacy.Adam(0.0002, 0.5))
```

```
    return gan

    # define training step
    def train(generator, discriminator, gan, x_train, epochs,
        batch_size=128):
        valid = np.ones((batch_size, 1))
        fake = np.zeros((batch_size, 1))

        for epoch in range(epochs):
            start_time = time.time()


            idx = np.random.randint(0, x_train.shape[0],
               batch_size)
            imgs = x_train[idx]

            noise = np.random.normal(0, 1, (batch_size, 100))
            gen_imgs = generator.predict(noise)

            d_loss_real = discriminator.train_on_batch(imgs,
               valid)
            d_loss_fake = discriminator.train_on_batch(
               gen_imgs, fake)
            d_loss = 0.5 * np.add(d_loss_real, d_loss_fake)


            noise = np.random.normal(0, 1, (batch_size, 100))
            g_loss = gan.train_on_batch(noise, valid)

            end_time = time.time()
            epoch_time = end_time - start_time

            print(f"{epoch} [D loss: {d_loss[0]} | D accuracy
               : {100*d_loss[1]}] [G loss: {g_loss}] [Epoch
               time: {epoch_time:.2f} seconds]")


    generator = build_generator()
    discriminator = build_discriminator()
```

```
130     discriminator.compile(loss='binary_crossentropy',
            optimizer=tf.keras.optimizers.legacy.Adam(0.0002, 0.5)
            , metrics=['accuracy'])
131     gan = build_gan(generator, discriminator)
132
133     train(generator, discriminator, gan, x_train, epochs
            =4000, batch_size=64)
134
135     # gen images
136     def show_generated_images(generator, num_images=25, dim
            =(5, 5), figsize=(10, 10)):
137         noise = np.random.normal(0, 1, (num_images, 100))
138         gen_imgs = generator.predict(noise)
139         gen_imgs = 0.5 * gen_imgs + 0.5
140
141         plt.figure(figsize=figsize)
142         for i in range(num_images):
143             plt.subplot(dim[0], dim[1], i+1)
144             plt.imshow(gen_imgs[i])
145             plt.axis('off')
146         plt.tight_layout()
147         plt.show()
148
149     show_generated_images(generator)
```

Bibliography

[1] I. Gulrajani, F. Ahmed, M. Arjovsky, V. Dumoulin, and A. Courville. Improved training of wasserstein gans. 2017.

[2] Y. Jiang. Applications of generative adversarial networks in materials science. *Materials Genome Engineering Advances*, 2, 2024.

[3] Y. Xin, E. Walia, and P. Babyn. Generative adversarial network in medical imaging: a review. *Medical Image Analysis*, 58:101552, 2019.

[4] S. Kazeminia, C. Baur, A. Kuijper, B. Ginneken, N. Navab, S. Albarqouni, and A. Mukhopadhyay. Gans for medical image analysis. *Artificial Intelligence in Medicine*, 109:101938, 2020.

[5] M. Frid-Adar, I. Diamant, E. Klang, M. M. Amitai, J. Goldberger, and H. Greenspan. Gan-based synthetic medical image augmentation for increased cnn performance in liver lesion classification. *Neurocomputing*, 321:321–331, 2018.

[6] R. Abdal, Y. Qin, and P. Wonka. Image2stylegan: how to embed images into the stylegan latent space? *2019 IEEE/CVF International Conference on Computer Vision (ICCV)*, 2019.

[7] L. Han, Y. Huang, and T. Zhang. Candidates vs. noises estimation for large multi-class classification problem. 2017.

[8] B. Liu, E. Rosenfeld, P. Ravikumar, and A. Risteski. Analyzing and improving the optimization landscape of noise-contrastive estimation. 2021.

[9] M. Labeau and A. Allauzen. An experimental analysis of noise-contrastive estimation: the noise distribution matters. 2017.

[10] B. Damavandi, S. Kumar, N. Shazeer, and A. Bruguier. Nn-grams: unifying neural network and n-gram language models for speech recognition. 2016.

[11] Y. Song. How to train your energy-based models. 2021.

[12] D. Kingma and M. Welling. An introduction to variational autoencoders. *Foundations and Trends® in Machine Learning*, 12:307–392, 2019.

[13] B. Kiran, D. Thomas, and R. Parakkal. An overview of deep learning based methods for unsupervised and semi-supervised anomaly detection in videos. *Journal of Imaging*, 4:36, 2018.

[14] Y. Varolgunes, T. Bereau, and J. Rudzinski. Interpretable embeddings from molecular simulations using gaussian mixture variational autoencoders. *Machine Learning Science and Technology*, 1:015012, 2020.

[15] S. Portillo. Dimensionality reduction of sdss spectra with variational autoencoders. 2020.

[16] C. Guo, J. Zhou, H. Chen, N. Ying, J. Zhang, and D. Zhou. Variational autoencoder with optimizing gaussian mixture model priors. *Ieee Access*, 8:43992–44005, 2020.

[17] P. Munjal, A. Paul, and N. Krishnan. Implicit discriminator in variational autoencoder. 2019.

[18] X. Bie, L. Girin, S. Leglaive, T. Hueber, and X. Alameda-Pineda. A benchmark of dynamical variational autoencoders applied to speech spectrogram modeling. 2021.

[19] J. Parikh, T. Rumbell, X. Butova, T. Myachina, J. Acero, S. Khamzin, O. Solovyova, J. Kozloski, A. Khokhlova, and V. Gurev. Generative adversarial networks for construction of virtual populations of mechanistic

models: simulations to study omecamtiv mecarbil action. *Journal of Pharmacokinetics and Pharmacodynamics*, 49:51–64, 2021.

[20] Y. Saito, S. Takamichi, and H. Saruwatari. Statistical parametric speech synthesis incorporating generative adversarial networks. *Ieee/Acm Transactions on Audio Speech and Language Processing*, 26:84–96, 2018.

[21] T. Miyato. Cgans with projection discriminator. 2018.

[22] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial networks. 2014.

[23] T. Kynkäänniemi, T. Karras, M. Aittala, T. Aila, and J. Lehtinen. The role of imagenet classes in fréchet inception distance. 2022.

[24] Y. Xu, T. Wu, J. Charlton, and K. Bennett. Gan training acceleration using fréchet descriptor-based coreset. *Applied Sciences*, 12:7599, 2022.

[25] R. Xu, J. Wang, J. Liu, F. Ni, and B. Cao. Thermal infrared face image data enhancement method based on deep learning. 2023.

[26] D. Berthelot, T. Schumm, and L. Metz. Began: boundary equilibrium generative adversarial networks. 2017.

[27] H. Hyungrok, T. Jun, and D. Kim. Unbalanced gans: pre-training the generator of generative adversarial network using variational autoencoder. 2020.