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: 12/09/2024

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 gained attention for their ability to generate realistic data in various fields. In this thesis, I explore the performance of GANs in generating high-quality cat images, utilizing the Animal Faces-HQ dataset, which includes 16,130 high-resolution images. The focus is on comparing different architectural designs, specifically convolutional and dense layers, and investigating how data augmentation techniques influence model performance.

The findings suggest that convolutional architectures offer advantages in capturing spatial features, contributing to improved image quality. While data augmentation introduces diversity into the dataset, it also presents optimization challenges that can complicate the training process. These results highlight important considerations for future studies aiming to enhance GAN-generated images. The insights gained from this work may guide further exploration of more advanced architectures and larger datasets.

Contents

1 Introduction **5**

2 Historical Models of Image Generation **7**

2.1 Noise Contrastive Estimation (NCEs) 7

2.1.1 NCE Architecture 8

2.1.2 NCE Objective Function 9

2.1.3 Comparison with GANs 10

2.1.4 Applications of NCE 11

2.1.5 Limitations of NCE 12

2.2 Variational Autoencoders (VAEs) 12

2.2.1 VAE Architecture 13

2.2.2 VAE Objective Function 13

2.2.3 Comparison with GANs 14

2.2.4 Applications of VAEs 15

2.2.5 Limitations of VAEs 15

2.3 Diffusion Models 16

2.3.1 Diffusion Model Architecture 16

2.3.2 Diffusion Model Objective Function 17

2.3.3 Comparison with GANs 17

2.3.4 Applications of Diffusion Models 18

2.3.5 Limitations of Diffusion Models 19

3 Theoretical Background **21**

3.1 Generative Adversarial Networks (GANs) 21

3.2 GAN Architecture 22

3.3 Objective Function of GAN 22
3.3.1 Discriminator's Loss Function 23
3.3.2 Generator's Loss Function 23
3.3.3 Modification for Vanishing Gradient Problem 23
3.4 GAN Training Process 24
3.4.1 Distribution Changes During GAN Training 25
3.4.2 Mathematical Formulation during GAN Training . . . 26
3.5 Evaluating GAN Performance 29
3.6 Limitations of Accuracy in Evaluating GANs 30

4 Results and Disscussion 32
4.1 Standard GAN Versus Other GAN Realizations 32
4.2 GAN With Convolutional or Dense layers 33
4.3 Exploring Layer Depth 37
4.4 Impact of Data Augmentation on Model Performance 38
4.5 Applying the Model to the Animal Faces-HQ Dataset 39
4.5.1 Model Structure and Training 39
4.5.2 Generated Results 42
4.6 Discussion 44

A Appendix 46

List of Figures

2.1 NCE Architecture. 9
2.2 VAE Architecture. 13
2.3 Diffusion Model Architecture. 17

3.1 GAN Architecture. 22
3.2 Comparison of $\log(1 - D)$ and $-\log(D)$. 24

3.3 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. 25
3.4 GAN Training Accuracy over Epochs 30
3.5 Generated Images from GAN 31

4.1 Generator Architecture with Dense and Convolutional Layers . 34
4.2 Discriminator Architecture with Dense and Convolutional Layers 35
4.3 Comparison of GAN Performance at 3000 Epochs 35
4.4 FID Scores Across 3000 Epochs 36
4.5 Cat Faces Generated by GAN 43

Chapter 1

Introduction

Generative Adversarial Networks (GANs) have emerged as a transformative tool in generative modeling, framing the problem as a competition between two networks: a generator that creates synthetic data from noise, and a discriminator that differentiates between real and generated data [1]. Since their introduction in 2014, GANs have found applications in fields such as materials science, radiology, and computer vision [2], [3], [4]. For instance, CycleGAN has been applied in medical imaging, enhancing tasks like liver lesion classification through synthetic image augmentation, outperforming traditional methods in sensitivity and specificity [5]. Similarly, StyleGAN has shown effectiveness in image deformation and style transfer [6], further expanding the reach of GANs in the computer vision field, where they generate data without explicitly modeling probability density functions [3].

However, despite their success, training GANs presents notable challenges, including issues like mode collapse, training instability, and the high computational demands required for effective performance. Evaluating GANs is also complex, as traditional metrics such as accuracy are insufficient to measure the quality and diversity of generated data. These difficulties have driven the development of various architectures and training methods aimed at improving the stability and effectiveness of GANs.

The objective of this thesis is to contribute to a clearer understanding of the factors that influence GAN performance, particularly in generating realistic images. For this purpose, I use the Animal Faces-HQ (AFHQ) dataset, containing 16,130 high-resolution images at 512×512 pixels, to train a GAN model specifically for generating realistic cat images. The high resolution presents both opportunities and challenges, as it requires careful attention to the model's architecture and training to avoid overfitting or underfitting.

This thesis is structured as follows: Chapter 2 offers an overview of previous models for image generation, including methods such as Noise Contrastive Estimation, Variational Autoencoders, and Diffusion Models, emphasizing their architectures, objectives, and applications. Chapter 3 focuses on Generative Adversarial Networks (GANs), discussing their theoretical foundation, including key concepts such as objective functions and training dynamics. Chapter 4 describes the experimental work, covering model selection, architecture exploration, the effects of data augmentation, and the application of the GAN model to the Animal Faces-HQ dataset. Lastly, it presents the results, their implications, and suggests directions for future research.

Chapter 2

Historical Models of Image Generation

In this chapter, I aim to provide an overview of three important generative modeling techniques: Noise Contrastive Estimation (NCE), Variational Autoencoders (VAEs), and Diffusion Models. These models have each played a significant role in the evolution of generative modeling and continue to influence modern approaches in the field.

2.1 Noise Contrastive Estimation (NCEs)

Noise Contrastive Estimation (NCE) was introduced in 2010 by Gutmann and Hyvärinen as a method for estimating parameters in unnormalized probabilistic models. It offers an efficient alternative to Maximum Likelihood Estimation (MLE), particularly in cases where MLE can become computationally expensive, especially with large-scale models. NCE reframes the challenge of normalizing probability distributions into a more manageable binary classification problem [7].

The core idea behind NCE is to treat MLE as a binary classification task. Traditionally, training models for unnormalized probability distributions using MLE involves calculating the partition function, which can be computationally infeasible for large datasets. NCE mitigates this difficulty by incorporating noise samples drawn from a known distribution. The model is then trained to differentiate between real data and noise samples, with higher probabilities assigned to real data and lower probabilities to noise. This approach allows the model to learn the data distribution without the need for explicit normalization [8].

2.1.1 NCE Architecture

In NCE, the likelihood of a data point x is reformulated as the probability that it comes from the real data distribution rather than from the noise distribution. As depicted in Figure 2.1, the architecture of a typical NCE model includes an input layer, a hidden layer, and an output layer. The input layer receives both real and noise samples, while the hidden layer learns representations of these samples. The output layer functions as a binary classifier, producing probabilities that indicate whether a given sample is real or noise.

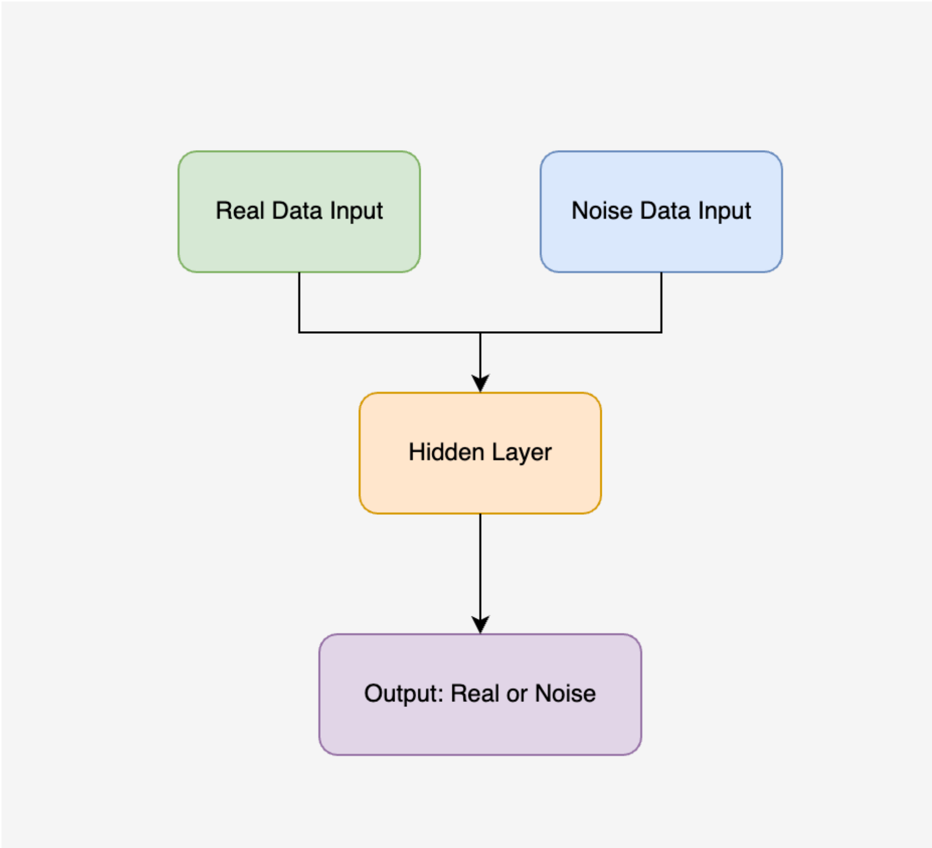


Figure 2.1: NCE Architecture.

- **Data and Noise Samples**: NCE introduces noise samples from a known distribution to compare with actual data. These noise samples act as negative examples in the classification task, while the real data serves as positive examples [7].
- **Binary Classification**: The task of differentiating between real and noise samples is framed as a binary classification problem, which can be optimized using standard logistic regression techniques [9].

2.1.2 NCE Objective Function

In NCE, the probability $P(y = 1|x)$, where $y = 1$ indicates that x is a real sample, is defined as:

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

Where $p_\theta(x)$ is the unnormalized probability assigned to the sample x by the model, $p_{\text{noise}}(x)$ is the probability assigned to the noise sample, and k is the ratio of noise samples to real samples.

The corresponding probability that a sample x is from the noise distribution 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 seeks to maximize the log-probabilities of correctly classifying real and noise samples:

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

This method circumvents the need for computing the partition function, which is typically required in Maximum Likelihood Estimation (MLE), making NCE particularly useful for large-scale models [8].

2.1.3 Comparison with GANs

Noise Contrastive Estimation (NCE) and Generative Adversarial Networks (GANs) are both prominent methods in generative modeling, but they take different approaches to training and model estimation.

- **Training Stability**: One of NCE's key advantages over GANs is the stability it provides during training. GANs frequently suffer from challenges such as mode collapse and instability due to the adversarial nature of the training process, where the generator and discriminator are pitted against each other. NCE, by contrast, frames the training process as a straightforward binary classification task, which tends to be more stable and can be optimized with logistic regression [9].

- **Computational Efficiency**: Training GANs requires simultaneously optimizing two models—the generator and the discriminator—which can lead to higher computational costs, particularly when tuning their interactions for better performance. NCE, on the other hand, simplifies the problem by reducing it to a comparison between real data and noise samples, avoiding the adversarial framework and offering a more computationally efficient solution, especially for large-scale models [10].

- **Handling Unnormalized Models**: NCE is particularly suited for training unnormalized probabilistic models, where computing the partition function is impractical or impossible. GANs, however, are focused on generating realistic samples and, while they do not require explicit normalization, they do not address the problem of unnormalized models as directly as NCE does [11].

- **Model Interpretability**: In NCE, the model explicitly learns to estimate the probability of real data versus noise, providing insights into the data distribution. In comparison, GANs focus on generating realistic samples without explicitly modeling probabilities, which can make their internal latent space less interpretable than that of NCE-based models.

- **Applications in Language Models and Word Embeddings**: NCE is particularly beneficial in tasks such as word embeddings and large-scale language models, where normalizing the likelihood function over vast vocabularies is computationally prohibitive. GANs have been applied in text generation, but NCE's efficiency in handling large-scale vocabulary models makes it more suitable for these problems [11].

2.1.4 Applications of NCE

NCE is highly effective in a variety of machine learning tasks, particularly those that involve large datasets and unnormalized models:

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

- **Language Models**: Beyond word embeddings, NCE has been applied to training large-scale language models, where traditional likelihood-based methods may become computationally expensive.

- **Energy-Based Models**: NCE is also effective in training energy-based models, which typically require an intractable partition function for normalization.

NCE allows models to scale efficiently, making it an invaluable tool in areas ranging from natural language processing to computer vision.

2.1.5 Limitations of NCE

While NCE has proven to be a powerful estimation technique, it is not without limitations. A key challenge lies in selecting an appropriate noise distribution. Poor choices in this regard can lead to suboptimal parameter estimates and slower convergence during training [8].

- **Noise Distribution Sensitivity**: The success of NCE relies heavily on choosing a noise distribution that is sufficiently different from the real data distribution. If the noise distribution is poorly chosen, the model may struggle to differentiate between real and noise samples [8].

- **Handling Complex Data Distributions**: NCE may encounter difficulties with highly complex data distributions, particularly in cases where defining an appropriate noise distribution is challenging [8].

2.2 Variational Autoencoders (VAEs)

Variational Autoencoders (VAEs), introduced in 2013, represent one of the foundational approaches to generative modeling. The primary goal of VAEs is to model the underlying distribution of data by learning a compressed latent representation, denoted as z, from the input data x.

2.2.1 VAE Architecture

VAEs consist of two main components: an encoder and a decoder. The encoder maps input data into a latent space, where the latent variables are generally assumed to follow a Gaussian distribution. This assumption simplifies the training process and enables techniques such as the reparameterization trick, which facilitates backpropagation through stochastic layers [12]. The decoder then reconstructs the input data from the latent variables, ensuring that the essential characteristics of the data distribution are captured.

As illustrated in Figure 2.2, the VAE architecture comprises:

- **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 aids in simplifying the optimization process.

- **Decoder**: The decoder reconstructs the original data x' from the latent representation z, aiming to produce data that closely resembles the input.

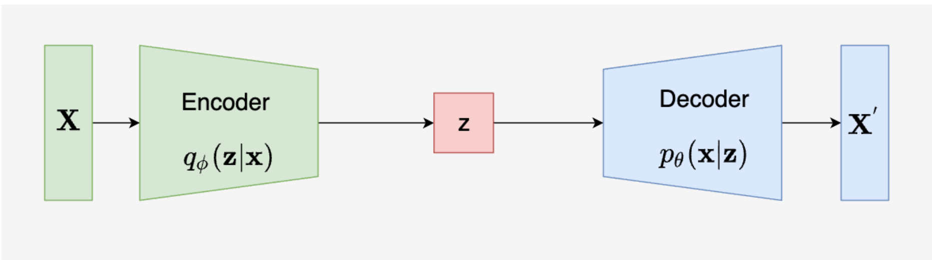


Figure 2.2: VAE Architecture.

2.2.2 VAE Objective Function

The VAE loss function is designed to balance two objectives: accurate data reconstruction and regularization of the latent space. The total loss combines a reconstruction loss with a Kullback-Leibler (KL) divergence term, which helps the model learn a smooth, continuous representation of the latent space [13].

The VAE aims to maximize the variational lower bound, encouraging both accurate reconstruction and a well-structured latent space. This, in turn, enables the model to generate new data by sampling from the latent space and reconstructing it using the decoder.

The loss function of a VAE is expressed 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 seeks to minimize.
- $\mathbb{E}_{q_\phi(z|x)}[\log p_\theta(x|z)]$: This term represents the expected log-likelihood of the reconstructed data x' given the latent variable z. The distribution $q_\phi(z|x)$ corresponds to the encoder's approximation of the posterior, while $p_\theta(x|z)$ represents the decoder's likelihood of reconstructing the input data.
- $D_{KL}(q_\phi(z|x) \| p(z))$: The KL divergence measures the difference between the encoder's learned latent distribution $q_\phi(z|x)$ and the prior distribution $p(z)$, which is typically Gaussian.

2.2.3 Comparison with GANs

When compared to Generative Adversarial Networks (GANs), VAEs exhibit several advantages. One of the most notable strengths of VAEs is their training stability. In contrast to GANs, which involve training both a generator and a discriminator—a process that can lead to issues such as mode collapse—VAEs have a single objective function. This objective combines reconstruction loss and KL divergence, making the training process more straightforward and less prone to instability [12].

Another key strength of VAEs is their ability to generate smooth transitions between data points. This capability is particularly valuable for applications that require meaningful interpolation in the latent space, such as image generation, anomaly detection, and data imputation [14][15]. GANs, on the other hand, do not explicitly model the latent space, which can limit their interpretability and flexibility in certain tasks.

However, GANs are often favored for generating sharper and more realistic images, particularly in high-resolution tasks. The adversarial loss in GANs encourages the generator to produce outputs that closely resemble real data, while VAEs, due to their reliance on Gaussian priors, tend to generate slightly blurrier images [16]. Nonetheless, VAEs offer greater flexibility and scalability, as they can be trained using standard gradient descent methods and do not require the complex adversarial framework inherent to GANs.

2.2.4 Applications of VAEs

The versatility of VAEs extends to a wide range of applications, many of which benefit from the structured latent space that VAEs provide:

- **Image Generation**: VAEs can generate new images by sampling from the latent space and decoding the latent vectors into realistic image representations.

- **Anomaly Detection**: VAEs are often used to identify outliers in data by examining reconstruction errors. Data points with high reconstruction loss may be flagged as anomalies.

- **Data Imputation**: VAEs are capable of filling in missing data by reconstructing the incomplete data from latent representations, making them useful for handling datasets with gaps or missing values.

2.2.5 Limitations of VAEs

While VAEs have proven effective in many applications, they also have certain limitations:

- **Blurred Outputs**: Due to the Gaussian prior assumption in the latent space, VAEs often generate blurrier outputs compared to GANs, which may affect performance in tasks requiring high-resolution and sharp images [16].

- **Difficulty with Discrete Data**: VAEs struggle to effectively model discrete data, as backpropagation through continuous latent variables is not well-suited to handle discrete structures [17].

- **Limited Diversity**: VAEs may not capture the full complexity and diversity of the data distribution as effectively as GANs, particularly when generating high-resolution images [18].

2.3 Diffusion Models

Diffusion models, emerging in the early 2020s, represent a significant advancement in generative modeling. These models progressively add noise to data and then learn to reverse this process, effectively "denoising" it back to its original form. This iterative process distinguishes diffusion models from traditional approaches like GANs and VAEs [19].

2.3.1 Diffusion Model Architecture

The core idea behind diffusion models relies on a Markov chain where noise is added in the forward process, starting from a simple distribution (e.g., Gaussian), and reversed through a learned denoising mechanism [20], [21]. This approach has proven effective in generating high-quality samples across various domains, such as image synthesis, audio generation, and medical imaging [22], [23]. In several benchmarks, diffusion models have outperformed GANs [19], [24].

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 noise level, 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.

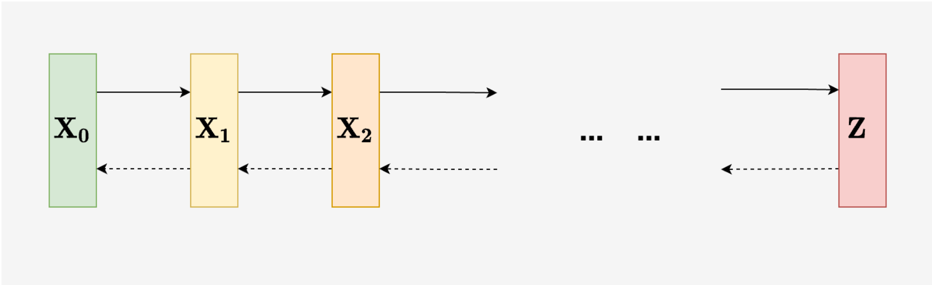


Figure 2.3: Diffusion Model Architecture.

2.3.2 Diffusion Model Objective 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:

$$L = \sum_{t=1}^{T} \mathbb{E}_{x_0,\epsilon} \left[\|\epsilon - \epsilon_\theta(x_t, t)\|^2 \right] \tag{2.5}$$

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.

2.3.3 Comparison with GANs

Diffusion models and Generative Adversarial Networks (GANs) represent two distinct approaches in generative modeling, each with its strengths and weaknesses. A key advantage of diffusion models is their training stability.

Unlike GANs, which often suffer from unstable dynamics due to the adversarial competition between the generator and discriminator, diffusion models follow a simpler, noise-reversal process that avoids these issues [25]. This stability helps diffusion models avoid problems like mode collapse, where GANs fail to capture the diversity of the data distribution [26], [27].

Diffusion models also excel in generating diverse, high-quality samples through a gradual denoising process, allowing for controlled generation [28]. In contrast, GANs tend to produce sharper but less diverse images, often overfitting to specific modes of the data [29], [30]. This trade-off between sharpness and diversity is a known limitation of GANs [31].

However, GANs remain preferred for tasks requiring extremely high-resolution and photorealistic images, such as human face generation, where they outperform diffusion models in terms of detail and realism [32], [33].

2.3.4 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. Recent innovations, such as classifier-free guidance, have further enhanced the quality of generated samples by allowing the model to generate data without relying on an explicit classifier [34].

- **Text-to-Image Generation**: Advancements in diffusion models have also been applied 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. Additionally, techniques like Denoising Diffusion Implicit Models (DDIMs) have accelerated the sampling process, making diffusion models more practical for real-time applications [35], [36].

- **Speech Synthesis**: Diffusion models have been applied to generating high-quality audio data. For example, they are used in text-to-speech systems to generate realistic human speech. These models have demonstrated remarkable performance in generating high-fidelity audio outputs [22], [20].

- **Anomaly Detection and Medical Imaging**: 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. Diffusion models have also been applied in medical image segmentation tasks, such as the MedSegDiff model, which enhances segmentation by leveraging diffusion processes [23]. These models have shown their ability to handle complex data structures, proving their versatility across different domains.

Recent innovations in diffusion models have further enhanced their applicability and efficiency. By reducing the computational burden associated with the iterative sampling process, models like DDIMs maintain the generative capabilities of traditional diffusion models while improving their practicality for real-time applications [37].

2.3.5 Limitations of Diffusion Models

Despite the significant advancements that diffusion models have brought to generative modeling, they are not without limitations, which can be categorized into three primary areas: computational intensity, generation speed, and sample sharpness.

- **Computationally Intensive**: Diffusion models are computationally expensive due to their iterative nature, which involves progressively adding and removing noise from data. This process requires significantly more computational resources compared to GANs, which generate data in a single forward pass through the generator [38], [39].

This computational burden can limit their practical use, especially in scenarios requiring rapid generation [40].

- **Generation Speed**: Diffusion models are slower than GANs in terms of sample generation. Producing a single sample often requires hundreds or even thousands of time steps, significantly increasing the generation time compared to the near-instantaneous output of GANs [41], [35]. This can be a critical drawback in real-time applications.

- **Sample Sharpness**: While diffusion models generate diverse outputs, they may not always match the photorealism and fine detail achieved by GANs, especially in tasks requiring intricate details [19], [42]. This can affect their suitability for applications such as high-resolution image generation or medical imaging [40].

Chapter 3

Theoretical Background

This chapter provides an overview of Generative Adversarial Networks (GANs), focusing on their core architecture, objective function, and training dynamics. It introduces the interplay between the generator and discriminator in the adversarial training process, as well as the Fréchet Inception Distance (FID) metric for evaluating the quality and diversity of generated images.

3.1 Generative Adversarial Networks (GANs)

Generative Adversarial Networks (GANs) were introduced by Goodfellow et al. in 2014 and have become a widely used approach in generative modeling. GANs consist of two neural networks: a generator (G) and a discriminator (D), trained simultaneously through an adversarial process. The generator is tasked with producing synthetic data samples, while the discriminator tries to distinguish between real and generated samples. This architecture leads to a game-like dynamic between the two networks, where the generator improves its ability to create realistic data while the discriminator becomes better at distinguishing real from fake data [43], [44] and provide feedback to the generator, helping it to improve and generate more realistic samples over time [45].

3.2 GAN Architecture

The structure of GANs consists of two key components:

- **Generator (G)**: The generator maps random noise z from a prior distribution, such as Gaussian or uniform, to synthetic data samples $G(z)$ that aim to replicate real data distributions.
- **Discriminator (D)**: The discriminator receives input data, which can be either real data x or generated data $G(z)$, and outputs the probability that the data is real. The discriminator is trained to classify real samples as real and generated samples as fake [45].

The following figure illustrates the GAN architecture, where the generator produces synthetic data, and the discriminator learns to differentiate real from fake data.

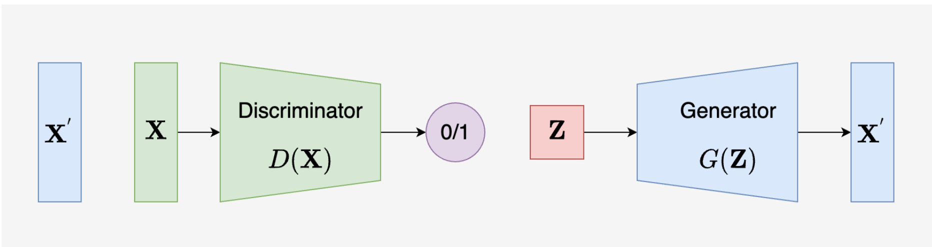


Figure 3.1: GAN Architecture.

3.3 Objective Function of GAN

The training of GANs involves optimizing two neural networks simultaneously through a minimax game. The objective function is defined as follows:

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

where :

- $z \sim p_z(z)$: The distribution from which the noise z is sampled. The generator transforms this noise into complex, high-dimensional data.

- $x \sim p_{data}(x)$: A sample x drawn from the true data distribution $p_{data}(x)$.
- $D(G(z))$: The discriminator's prediction for generated data $G(z)$.
- $D(x)$: The discriminator's prediction for real data x.
- $\mathbb{E}_{x \sim p_{data}(x)}[\log D(x)]$: The expectation of $\log D(x)$ over the real data distribution.
- $\mathbb{E}_{z \sim p_z(z)}[\log(1 - D(G(z)))]$: The expectation of $\log(1 - D(G(z)))$ over the noise distribution.

3.3.1 Discriminator's Loss Function

The discriminator's loss function measures its ability to distinguish between real and generated data. It is expressed as:

$$\mathcal{L}_D = -\mathbb{E}_{x \sim p_{\mathrm{data}}(x)}[\log D(x)] - \mathbb{E}_{z \sim p_z(z)}[\log(1 - D(G(z)))]. \tag{3.2}$$

The discriminator minimizes this loss function to improve its ability to classify real and generated data correctly.

3.3.2 Generator's Loss Function

The generator's loss function is designed to encourage it to generate data that can fool the discriminator. It is given by:

$$\mathcal{L}_G = -\mathbb{E}_{z \sim p_z(z)}[\log D(G(z))]. \tag{3.3}$$

The generator minimizes this loss function to produce data that the discriminator classifies as real.

3.3.3 Modification for Vanishing Gradient Problem

To address the vanishing gradient problem during GAN training, the generator's loss is often modified. The original form $\log(1 - D(G(z)))$ can result in small gradients when the discriminator accurately identifies fake data. Therefore, the alternative loss function for the generator is:

$$\mathcal{L}_G = \mathbb{E}_{z \sim p_z(z)}[-\log(D(G(z)))]. \tag{3.4}$$

This modification ensures larger gradients and more stable training.

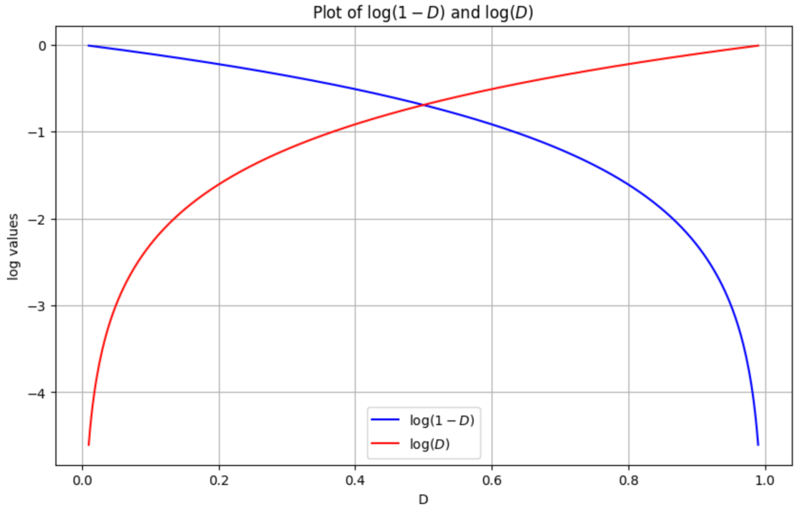


Figure 3.2: Comparison of $\log(1 - D)$ and $-\log(D)$.

3.4 GAN Training Process

During the training process of the GAN model, the generator and the discriminator are in constant competition. The generator aims to produce increasingly realistic samples to deceive the discriminator, while the discriminator works to accurately distinguish between real and generated samples. Initially, the generator produces samples that are of low quality and deviate significantly from the real data, as illustrated by the blue sine wave in Figure (a). As training progresses, the generator improves, and the generated samples become more refined, as shown by the smoother blue lines in Figures (b) and (c). Simultaneously, the distribution of the real samples and generated samples gradually align, and finally, in the optimal state represented by Figure (d), $p_{data}(x) = p_g(x)$, meaning that the generated samples closely match the real samples. Throughout the training process, the distribution of the generated samples transitions from an initially noisy and scattered state to one that closely mirrors the real data distribution, indicating a significant enhancement in the quality of the generated samples.

3.4.1 Distribution Changes During GAN Training

A simaple diagram show how distribution change in GAN training.

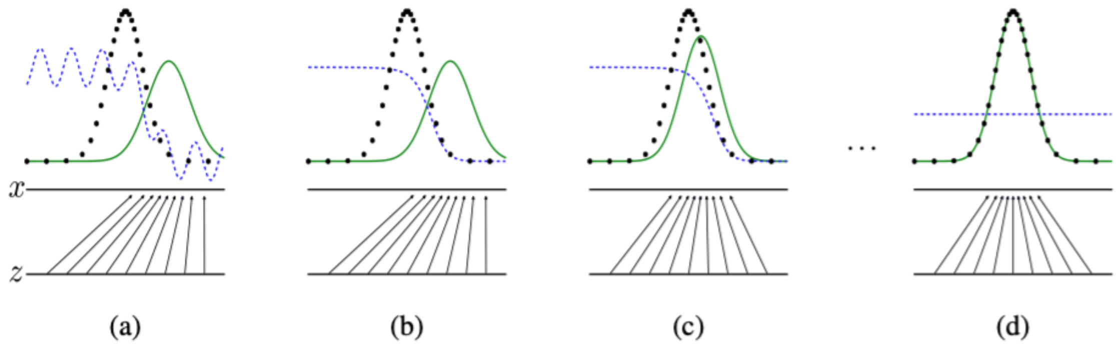


Figure 3.3: Diagram of GAN's Training Process. The green curve represents the distribution of generated samples. Initially, the generated samples may differ significantly from the real samples. As training progresses, the generated samples' distribution gradually approaches the real samples. The black 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: [46]

As training progresses, the samples generated by the generator gradually become indistinguishable from the real data, leading to a minimized error between the two distributions. Initially, the generated samples exhibit significant differences from the real data, starting from a noisy and dispersed state. However, over time, the generator learns to map the latent space to the real data distribution more accurately. This results in the generated samples converging toward the real data distribution, ultimately aligning closely with the target distribution.

3.4.2 Mathematical Formulation 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.

3.5 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 [47]. 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 [48]. 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 [49]. 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 [49].

3.6 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: 7.715634183114162e-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 3.4: GAN Training Accuracy over Epochs

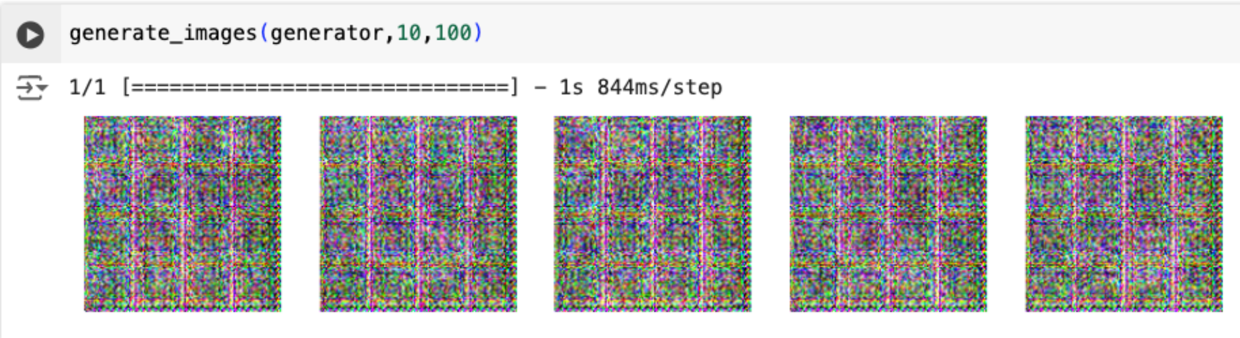


Figure 3.5: Generated Images from GAN

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 and Disscussion

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

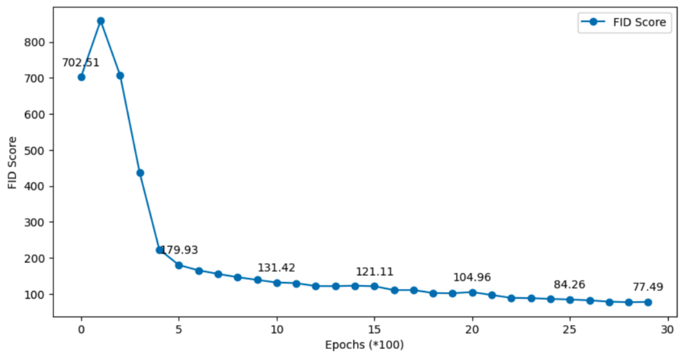
Figure 4.2: Discriminator Architecture with Dense and Convolutional Layers

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.

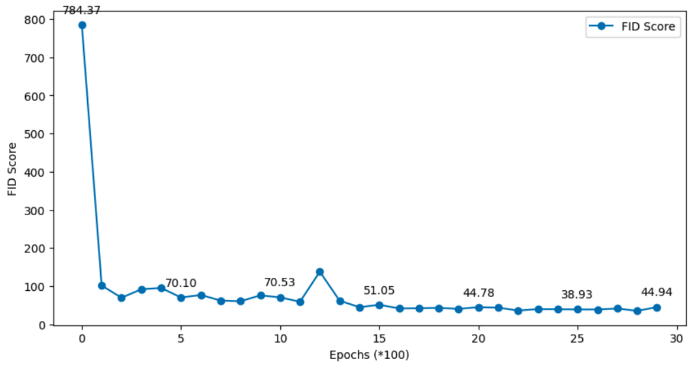
(a) Images Generated by GAN with Dense Layers

(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 D / Layers in G	**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 [50]. The importance of this balance is crucial for the effective operation of GANs, ensuring stable learning and improved image quality [51].

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), 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
```

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

Appendix

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
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 = []  # List to store 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 and print 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.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'))
        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))

        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

    # 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
    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))
```

```python

        # using half batches for the discriminator ensures balanced and efficient training,
        # better memory management, and more stable training dynamics in GANs.
        half_batch = int(batch_size / 2)

        d_losses = []
        g_losses = []
        d_acc = []
        fid_scores = []  # List to store 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.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

    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))

        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)

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)

        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]

            real_images_augmented = next(datagen.flow(real_images, batch_size=half_batch))

            # 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_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.4: Explore data augmentation 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()

        # 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))

        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)

    datagen = tf.keras.preprocessing.image.ImageDataGenerator(
        rotation_range=10,
```

```
        width_shift_range=0.1,
        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)

    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]

        real_images_augmented = next(datagen.flow(
            real_images, batch_size=half_batch))

        # 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_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.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
```

```python

    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'))
```

```python
    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)

datagen = tf.keras.preprocessing.image.ImageDataGenerator(
    width_shift_range=0.1,
    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)

        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]

            real_images_augmented = next(datagen.flow(
                real_images, batch_size=half_batch))

```

```
            # 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_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))

    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))

    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)

        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.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():
```

```python
    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 first 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))

    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))

        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)

        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.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))

        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))

        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)

        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.9: Explore GAN with more convolutional layers 4

```
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

```

```python
    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'))
        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'))

```

```python
    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))

    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
```

```python

        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))
        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'))
        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))

        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))

        # add the 1st convolution layer
        model.add(Conv2D(512, kernel_size=3, strides=2, padding='same'))
        model.add(LeakyReLU(alpha=0.2))
        model.add(Dropout(0.25))

        # add the 2nd convolution layer
        model.add(Conv2D(1024, 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 = []

```

```python
        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.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'))
    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

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))

    model.add(Conv2D(256, 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(512, kernel_size=3, strides=2, padding='same'))
    model.add(LeakyReLU(alpha=0.2))
    model.add(Dropout(0.25))

```

```
    # add the 2nd convolution layer
    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):
```

```python
    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.12: Apply new dataset

```

import os
import time
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt
from tensorflow.keras.models import Sequential, Model
from tensorflow.keras.layers import Dense, Reshape, BatchNormalization, LeakyReLU, Conv2D, Conv2DTranspose, Flatten, Dropout, Input
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.preprocessing.image import load_img, img_to_array

def load_images_as_rgb_matrices(path, size):
images = []
for img_name in os.listdir(path):
    img_path = os.path.join(path, img_name)
    img = load_img(img_path, target_size=size)
    img_array = img_to_array(img)
    images.append(img_array)
images = np.array(images)
images = (images - 127.5) / 127.5  # Normalize images to [-1, 1]
return images

from google.colab import drive

# mount Google Drive
```

```python
drive.mount('/content/drive')

cat_path = "/content/drive/My Drive/gan/afhq/train/cat"
size = (128, 128)
x_train = load_images_as_rgb_matrices(cat_path, size)

# generator
# input 100
# output 128*128*3

def build_generator():
    model = Sequential()

    # Increase the dimension
    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))

    model.add(Conv2DTranspose(256, kernel_size=4, strides=2, padding='same'))  # 32x32
    model.add(LeakyReLU(alpha=0.2))
    model.add(BatchNormalization(momentum=0.8))

    model.add(Conv2DTranspose(128, kernel_size=4, strides=2, padding='same'))  # 64x64
    model.add(LeakyReLU(alpha=0.2))
    model.add(BatchNormalization(momentum=0.8))

    model.add(Conv2DTranspose(64, kernel_size=4, strides=2, padding='same'))  # 128x128
    model.add(LeakyReLU(alpha=0.2))
    model.add(BatchNormalization(momentum=0.8))

    model.add(Conv2D(3, kernel_size=7, activation='tanh', padding='same'))  # 128x128x3

return model

```

```
# discriminator
# 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]")


```

```python
    generator = build_generator()
    discriminator = build_discriminator()
    discriminator.compile(loss='binary_crossentropy', optimizer=tf.keras.optimizers.legacy.Adam(0.0002, 0.5), metrics=['accuracy'])
    gan = build_gan(generator, discriminator)

    train(generator, discriminator, gan, x_train, epochs=4000, batch_size=64)

    # gen images
    def show_generated_images(generator, num_images=25, dim=(5, 5), figsize=(10, 10)):
        noise = np.random.normal(0, 1, (num_images, 100))
        gen_imgs = generator.predict(noise)
        gen_imgs = 0.5 * gen_imgs + 0.5

        plt.figure(figsize=figsize)
        for i in range(num_images):
            plt.subplot(dim[0], dim[1], i+1)
            plt.imshow(gen_imgs[i])
            plt.axis('off')
        plt.tight_layout()
        plt.show()

    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] P. Dhariwal. Diffusion models beat gans on image synthesis. 2021.

[20] Z. Kong. Diffwave: a versatile diffusion model for audio synthesis. 2020.

[21] Q. Zhang, M. Tao, and Y. Chen. Gddim: generalized denoising diffusion implicit models. 2022.

[22] S. Liu, D. Su, and D. Yu. Diffgan-tts: high-fidelity and efficient text-to-speech with denoising diffusion gans. 2022.

[23] J. Wu, H. Fang, Y. Yang, and Y. Xu. Medsegdiff: medical image segmentation with diffusion probabilistic model. 2022.

[24] K. Pandey, A. Mukherjee, P. Rai, and A. Kumar. Diffusevae: efficient, controllable and high-fidelity generation from low-dimensional latents. 2022.

[25] M. Fathallah, M. Sakr, and S. El-etriby. Stabilizing and improving training of generative adversarial networks through identity blocks and modified loss function. *Ieee Access*, 11:43276–43285, 2023.

[26] J. Mu, C. Chen, W. Zhu, S. Li, and Y. Zhou. Taming mode collapse in generative adversarial networks using cooperative realness discriminators. *Iet Image Processing*, 16:2240–2262, 2022.

[27] Y. Zou. Auto-encoding generative adversarial networks towards mode collapse reduction and feature representation enhancement. *Entropy*, 25:1657, 2023.

[28] Y. Gong, Z. Ming, J. Yang, M. Xie, and X. Ma. Distribution constraining for combating mode collapse in generative adversarial networks. *Journal of Electronic Imaging*, 32, 2023.

[29] A. Dieng. Prescribed generative adversarial networks. 2019.

[30] J. Dubinski, K. Deja, S. Wenzel, P. Rokita, and T. Trzciński. Selectively increasing the diversity of gan-generated samples. 2022.

[31] Y. Chen, Q. Gao, and W. Xiao. Inferential wasserstein generative adversarial networks. *Journal of the Royal Statistical Society Series B (Statistical Methodology)*, 84:83–113, 2021.

[32] H. Zhu. Comparison of deep convolutional gan and progressive gan for facial image generation. *Applied and Computational Engineering*, 18:165–171, 2023.

[33] H. Zhao, T. Li, Y. Xiao, and Y. Wang. Improving multi-agent generative adversarial nets with variational latent representation. *Entropy*, 22:1055, 2020.

[34] J. Ho and T. Salimans. Classifier-free diffusion guidance. 2022.

[35] J. Song, C. Meng, and S. Ermon. Denoising diffusion implicit models. 2020.

[36] K. Preechakul, N. Chatthee, S. Wizadwongsa, and S. Suwajanakorn. Diffusion autoencoders: toward a meaningful and decodable representation. 2021.

[37] E. Luhman. Knowledge distillation in iterative generative models for improved sampling speed. 2021.

[38] A. Creswell, T. White, V. Dumoulin, K. Arulkumaran, B. Sengupta, and A. A. Bharath. Generative adversarial networks: an overview. *IEEE Signal Processing Magazine*, 35:53–65, 2018.

[39] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio. Generative adversarial networks. *Communications of the Acm*, 63:139–144, 2020.

[40] A. Kazerouni, E. Aghdam, M. Heidari, R. Azad, M. Fayyaz, I. Hacihaliloglu, and D. Merhof. Diffusion models for medical image analysis: a comprehensive survey. 2022.

[41] Y. Song. Score-based generative modeling through stochastic differential equations. 2020.

[42] A. Lugmayr, M. Danelljan, A. Romero, F. Yu, R. Timofte, and L. Gool. Repaint: inpainting using denoising diffusion probabilistic models. 2022.

[43] 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.

[44] 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.

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

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

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

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

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

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

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