SpringerBriefs in Computer Science

Sanaa Kaddoura

A Primer on Generative Adversarial Networks

Springer

SpringerBriefs in Computer Science

SpringerBriefs present concise summaries of cutting-edge research and practical applications across a wide spectrum of fields. Featuring compact volumes of 50 to 125 pages, the series covers a range of content from professional to academic.

Typical topics might include:

- A timely report of state-of-the art analytical techniques
- A bridge between new research results, as published in journal articles, and a contextual literature review
- A snapshot of a hot or emerging topic
- An in-depth case study or clinical example
- A presentation of core concepts that students must understand in order to make independent contributions

Briefs allow authors to present their ideas and readers to absorb them with minimal time investment. Briefs will be published as part of Springer's eBook collection, with millions of users worldwide. In addition, Briefs will be available for individual print and electronic purchase. Briefs are characterized by fast, global electronic dissemination, standard publishing contracts, easy-to-use manuscript preparation and formatting guidelines, and expedited production schedules. We aim for publication 8–12 weeks after acceptance. Both solicited and unsolicited manuscripts are considered for publication in this series.

**Indexing: This series is indexed in Scopus, Ei-Compendex, and zbMATH **

Sanaa Kaddoura

A Primer on Generative Adversarial Networks

Springer

Sanaa Kaddoura
Zayed University
Abu Dhabi, United Arab Emirates

ISSN 2191-5768 ISSN 2191-5776 (electronic)
SpringerBriefs in Computer Science
ISBN 978-3-031-32660-8 ISBN 978-3-031-32661-5 (eBook)
https://doi.org/10.1007/978-3-031-32661-5

This Springer imprint is published by the registered company Springer Nature Switzerland AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

Preface

Generative Adversarial Networks, or GANs, are a type of artificial neural network that has taken the field of machine learning to another level. Ian Goodfellow and his colleagues invented it in June 2014. GANs have the unique ability to generate new, realistic data that closely mimics the distribution of the original dataset they were trained on it. GANs can create images, videos, and even music. It revolutionized many domains, such as computer vision, natural language processing, and robotics. GANs can produce photos of human faces that cannot be differentiated from the photo taken by a camera.

This book, *A Primer on Generative Adversarial Networks*, is designed for readers who want to learn GAN without going into their mathematical background. Also, the book considers the straightforward applications of GANs, which allows a beginner reader in this topic to learn it. The book suits researchers, developers, students, and anyone wanting to practice GANs. The author assumes the reader is familiar with machine learning and neural networks. The book will include some ready-to-run scripts that can be used for further research. The programming language used in the book is python. So, it is assumed that the reader is familiar with the basics of this language.

The book starts with an overview of GAN architecture, explaining the idea of generative models. Then it goes into the most straightforward GAN architecture that is utilized to explain how GANs work, including the generator and discriminator concepts. Next, the book delves into more advanced applications of GANs from the real world such as human faces generation, deep fake, CycleGANs, and others.

By the end of this book, the readers will be able to write their own GAN code after understanding how GAN works and its potential applications. The reader can employ GANs as a solution in their projects. Whether the reader is a beginner or a seasoned machine learning practitioner, I hope this book will suit you.

All the codes of the book can be downloaded from: https://github.com/sn-code-inside/A-Primer-on-GNAs

Zayed University, Abu Dhabi, United Arab Emirates Sanaa Kaddoura
April 2023

Acknowledgments

I am grateful to Ms. Reem Nassar who supported the technical part of this book. She also contributed in writing the codes of the book.

Contents

Chapter 1
Overview of GAN Structure

Abstract This chapter will introduce generative adversarial networks (GANs), a type of neural network that can generate new data samples that resemble a given dataset. The chapter will discuss the difference between generative and discriminative models, an overview of the basic architecture of GANs, which consists of a generator and a discriminator, and how they are trained using an adversarial process. We also discuss the weaknesses of GANs and highlight some of their current challenges and limitations.

1.1 Introduction

Machine learning is an ever-evolving research area under artificial intelligence (AI). It allows the machine to imitate the human learning process to predict unseen data without being explicitly programmed accurately. Machine learning algorithms usually find patterns in the data to make predictions. The machine can learn from experience to improve accuracy by recognizing these patterns. Exploring specific patterns in data and making accurate decisions without being explicitly programmed has made it one of the most exciting fields of study in recent years.

It is important to note that the field of machine learning is vast, and there are different types of machine learning algorithms. For example, generative learning is the most recent innovation that focuses on creating new data similar to the existing one. Here is where generative adversarial networks (GANs) come into play.

GANs have been AI's most exciting idea in the last decade. They have revolutionized the deep learning paradigm leading to some of the most significant technological advances in AI history. GANs are considered a significant breakthrough in AI because they can generate new data that is not just a copy of the training data.

GANs were first introduced by J. Goodfellow and other co-authors [1]. They are composed of two neural networks, a generator and a discriminator, competing against each other in a zero-sum game. This process is called adversarial training, where the generator or the discriminator wins, and the other loses.

S. Kaddoura, *A Primer on Generative Adversarial Networks*, SpringerBriefs in Computer Science, https://doi.org/10.1007/978-3-031-32661-5_1

Fig. 1.1 Generated human faces by GAN

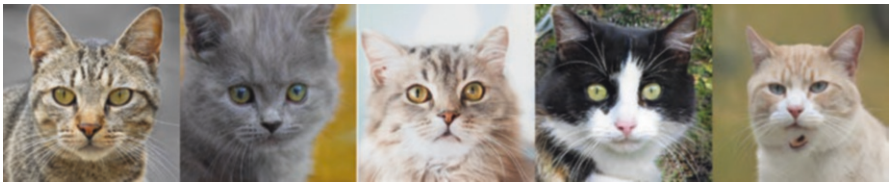

Fig. 1.2 Generated cats by GAN

GANs have a broad scope of applications in several areas. Below is one of the most popular and well-known GANs based on the research work in [2]. They used GANs to generate images through a website created by the researchers.[1] The proposed GAN model generates real human faces for people who do not exist. The generated images are almost identical to real faces. All faces in Fig. 1.1 are GAN-generated images and do not belong to any actual person. Similarly, the cats in Fig. 1.2 are not real; a GAN model generates them. The website can also generate houses, vessels, resumes, and others.

GANs can reach these highly realistic images by utilizing two networks: a generator and a discriminator. The generator generates image instances from a random noise vector in this case. However, the discriminator is responsible for recognizing whether the image is actual or generated by the generator. Throughout this process, the generator will work on deceiving the discriminator, and the discriminator will try not to be deceived.

GANs have experienced substantial enhancements and progress since their inception in 2014. These developments include improving GAN stability which authors achieve in [3] through utilizing different loss functions. Now, it can generate more realistic photos with high resolution. As shown in Fig. 1.3, the generated image quality was enhanced over time until it reached an acceptable quality in 2017. Now, it has been more potent in generating better-quality images. The photos from 2014 to 2017 in Fig. 1.3 are taken from the papers in [1, 4–6], respectively [7]. The photo of 2018 was taken from [8]. The photos of 2019 and 2021 are taken from [9]. The author generated the photo of 2023.

This book introduces GANs and their applications. After preparing the environment for coding, the reader will learn to implement a basic GAN from scratch. Then, real-world applications of GANs are introduced along with their

[1] Kashish Hora, "This x Does Not Exist", https://thisxdoesnotexist.com/. [Accessed: 13-Dec-2022].

Fig. 1.3 Development of GANs from 2014 to 2023

implementation. The reader is expected to know the basics of machine learning algorithms and their implementations.

1.2 Generative Models

Understanding the difference between the two types of machine learning models is important before explaining GANs. Machine learning models are either discriminative or generative. One can distinguish how generative models differ from discriminative ones. Discriminative is a term used for models that are used for classification; generative is a term used for models that generate new data. As shown in Fig. 1.4, the discriminative model is a classifier. It is uncertain about the input data and does not consider how the data was generated. A discriminative model takes the data instances to learn a function that maps input data to classes and then predicts the class of new instances. However, a generative model can generate new data compared to the discriminative model.

Assume that a convolutional neural network (CNN) model is built to classify images of dogs and cats. The CNN model will input cat and dog images, each having a class label to differentiate between them. By assigning weights and biases to each image, the CNN model gives the class an output, whether a dog or a cat. What if the problem was in the opposite direction? The model generates images based on a class label input; this is where generative models come. These models generate output based on input given to the machine. They are based on unsupervised machine learning algorithms. The generative model will be first trained using a training dataset. Then it learns the patterns in the input data to generate new data

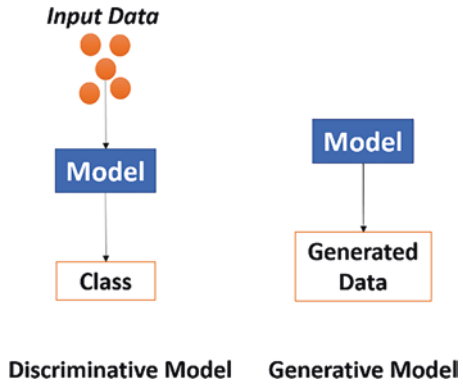

Fig. 1.4 Abstract definition of discriminative and generative models

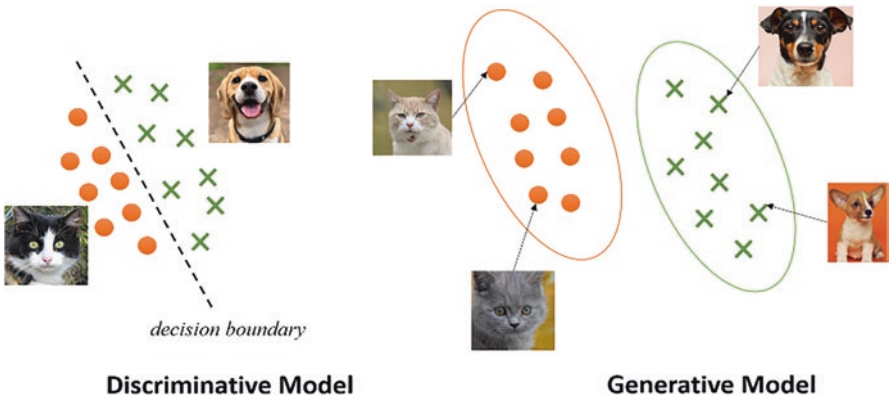

Fig. 1.5 Discriminative vs. generative model

(image, text, video, speech) from the acquired knowledge. As shown in Fig. 1.4, the discriminative model is a classifier, such as classifying an image as a dog or cat. However, generating an image of a dog or a cat is a generative problem.

The discriminative model calculates the conditional probability p(X|Y). The variable X is the set of features of the target-generated object, such as a big nose, long hair, whether it meows or not, or other feature sets. The variable Y says whether it is a cat or a dog. So, given a set of features X, determine the probability of class Y.

The generative model calculates the joint probability p(X, Y) or p(X) without labels. For example, it learns how to create realistic images of dogs or cats. The input of this model is a random number or set of numbers referred to as noise, such as [−3, 3.2, 5]. Also, sometimes it takes a class Y, such as dogs or cats. Given these inputs, the model will generate the features set X that will resemble Y. The noise is needed to generate different images every time.

As shown in Fig. 1.5, discriminative models draw a decision boundary between data. The decision boundary drawn by discriminative models separates different classes. However, the generative model tries to know how data is located in space.

By drawing the decision boundary, the discriminative model tries to tell the difference between a cat and a dog picture. If the boundary is drawn accurately, the model can recognize a cat photo from a dog photo without knowing the data samples' placement. On the other hand, a generative model tries to generate persuasive photos of cats and dogs that are close to the authentic images in the data space. Thus, the generative model needs to model the data all over the data space.

1.3 GANS

In this section, a full explanation of GANs will be displayed. The subsequent sections will delve deeper into GANs and discuss their architecture, the generator, the discriminator, the training stage, the loss function, and weaknesses.

Overview of GAN Structure

The terms generator/generative and discriminator/discriminative will be frequently used. They are opposite but will be used together since they are the two primary components of a GAN. The generator and discriminator are neural networks that aim to analyze the variations in the dataset. GAN treats the unsupervised generative task using the two neural networks as supervised. The generator is the model responsible for generating new reasonable examples from the problem domain. The discriminator is the model responsible for classifying examples as real belonging to the problem domain or fake generated by the generator. For example, if the problem is generating automobile images, the discriminator will tell whether the image is an automobile.

The generator learns to produce plausible data, while the discriminator learns to differentiate real from fake data. The discriminator is trained on actual samples to classify the input as real or fake. It uses the generated data from the generator as the negative training examples. The discriminator penalizes the generator every time it produces non-reasonable results by giving it a score. From the scores given by the discriminator, the generator will learn how to enhance the generated data.

Figure 1.6 shows that the generator is trying to generate fake automobile data images. It starts generating fake images when the training process starts. The discriminator quickly classifies them as fake. As the training process moves forward, the generator generates images closer to the real ones. However, the discriminator can still identify the images as fake, as shown in Fig. 1.7. When the training stage of the generator reaches an advanced level; the generator will generate images that look like real ones. Realistic-generated images will trick the discriminator that cannot distinguish fake from real images. Thus, it will classify fake images as real ones, as shown in Fig. 1.8. Now, the complete GAN structure can be formed.

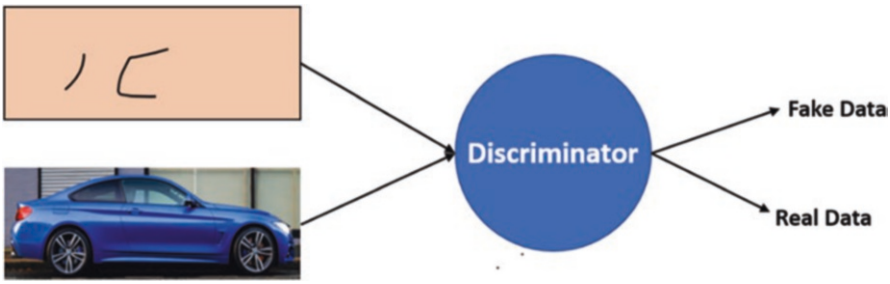

Fig. 1.6 The beginning of the training process

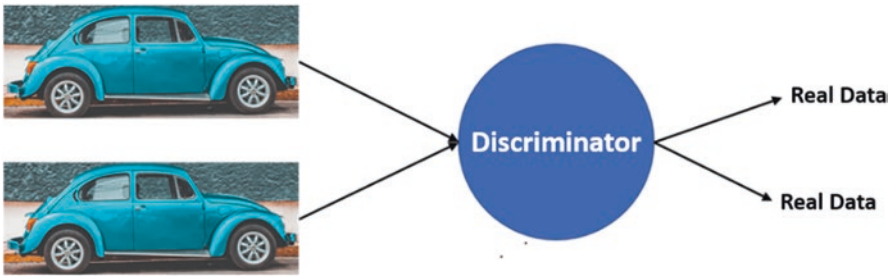

Fig. 1.8 Advanced generator training stage

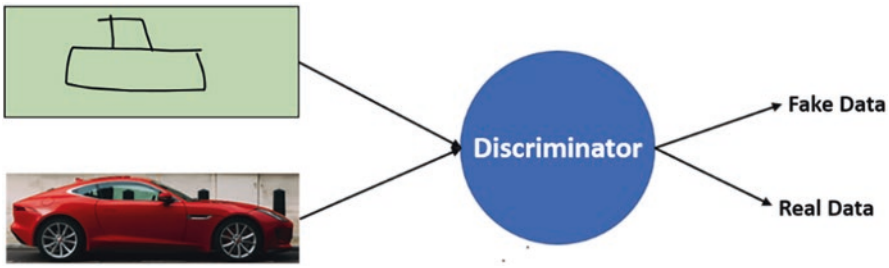

Fig. 1.7 Progress in training stage of generator

The Discriminator

The discriminator is a vital component of the GAN architecture, as it tries to classify whether the input data is real or fake. It can be any architecture, typically neural networks. Most discriminative models in GANs are made of a convolutional neural network (CNN). The discriminator model is trained on real-world data samples to learn specific patterns found in the real samples.

During training, the discriminator receives data that combines two sources. The first source is real data from the problem domain itself. Real data can be real-world images, text, or a combination of text and images. To illustrate, if the problem is

generating realistic photographs of human faces, the discriminator receives authentic images of human faces. However, if the problem is to generate images from text data, the discriminator will receive a combination of text with its corresponding image sample as input. This data source can be called the positive samples since they are real. The second data source is the samples coming out from the generator. These are fake data; hence they can be called negative samples.

Once the input data is fed to the discriminator, it will pass through the layers to produce a binary output. The binary output indicates whether the input is an actual or generated (fake) data sample. While the discriminator is in the training stage, the generator weights are frozen. At this stage, the generator feeds the discriminator with data to train on. The loss function used by the discriminator is *binary cross-entropy*, which measures the difference between the generator output and the input data sample's actual label.

The discriminator detection accuracy increases through backpropagation. Thus, it will better differentiate between real and fake samples. As the discriminator becomes more powerful than before, the generator improves its generated samples. This iterative process continues until the generator produces samples indistinguishable from real ones.

The discriminator architecture is presented in Fig. 1.9. The grey box shows that it is composed of several down-sampling layers. The down-sampling layers are represented by multiple rectangles placed beside each other with reducing size. The left side of the figure represents the real or fake input data to the discriminator. The right side of the figure represents the binary classes out from the discriminator, either fake or real.

The Generator

The generator is the main component in GANs, responsible for data generation. It can be any architecture, typically a neural network. Generally, GANs generator model is made of several up-sampling layers. The input data sample is passed

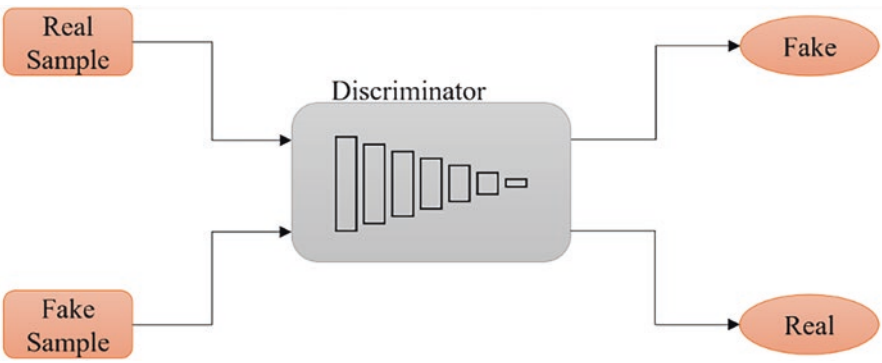

Fig. 1.9 Discriminator architecture

through these layers to reach the desired shape. For example, if the aim is to generate images of a human face with shape (224,224,3), the last layer in the generator will generate an image with this shape. Usually, the generator's input is based on the task required. To illustrate, in image generation, it will be a noise. In text-to-image translation, input will be a combination of text and noise. In the case of image-to-image translation, the input will be the source image.

The generator receives the input data during training and produces the desired output. It will start by producing random noise far from the actual data. It aims to fool the discriminator by evaluating the quality of its generated images using a specific loss function. The generator aims to reduce the loss function by updating weights and biases to make high-quality data.

The generator loss at the first stage of training is high. While training the generator, the discriminator weights are frozen. However, the generator updates its weights based on how much the discriminator model is accurate. Similarly, the generator is trained using backpropagation.

During the training, the generator keeps updating its weights based on the feedback given by the discriminator. At the end of the training process, the generator model can generate fake samples that look like real ones. The generator also aims not to overfit the training data.

Figure 1.10 displays the generator structure. The grey box demonstrates the several up-sampling layers of the generator model. The up-sampling layers are represented by multiple rectangles placed beside each other with increasing sizes. The input to the generator, usually a random noise, is shown on the left side of the figure. The generated fake samples are presented on the right side of the figure.

Training the GAN

Training the GAN means training its components, the generator, and the discriminator networks. This process involves alternating between training both networks in a way that leads to an equilibrium between them. The training process in GANs is responsible for updating the weights in the generator and the discriminator. It ends when both models reach convergence.

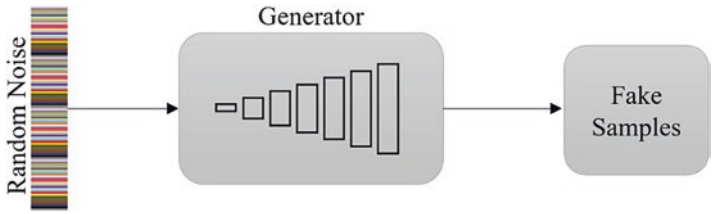

Fig. 1.10 Generator architecture

GAN training aims to reach a state where the discriminator cannot distinguish between generated and real data. In addition, the generator outputs high-quality data similar to the real one. This state is called Nash equilibrium, where neither network can improve its performance further.

Achieving good-quality data requires a robust neural network model. Thus, careful hyperparameter tuning, correct model architecture, and an appropriate number of layers are required. Building the generator model with optimized architecture does not need to be achieved through a single try. The training process is the longest part of GANs, where it is done over epochs and takes time to finish. The generated data quality increases as the number of epochs increases, but there might be a risk of overfitting. In addition, some GAN applications use progressive training, starting by producing new data with low resolution and then passing it to another GAN model to produce high-resolution data. It is usually applied to image generation so that the model learns the data features gradually and converges to a better solution. Figure 1.11 presents the GAN training architecture.

Loss Function

In general, the GAN loss function is known as *min-max* loss. It was described first in [1]. The generator aims to minimize the loss to produce high-quality data, while the discriminator aims to maximize it. GANs use two loss functions to train their components: generator loss and discriminator loss.

The generator loss function is used to evaluate the quality of generated data. In other words, it measures how well the generator generates fake images that resemble the real ones. The generator model aims to minimize this loss function and produce better images.

The discriminator loss function evaluates the model's ability to generate real data. Alternatively, it measures how well the discriminator can discriminate between real and fake data samples. The discriminator model aims to maximize this loss function to enhance its ability to distinguish between the two types of data samples.

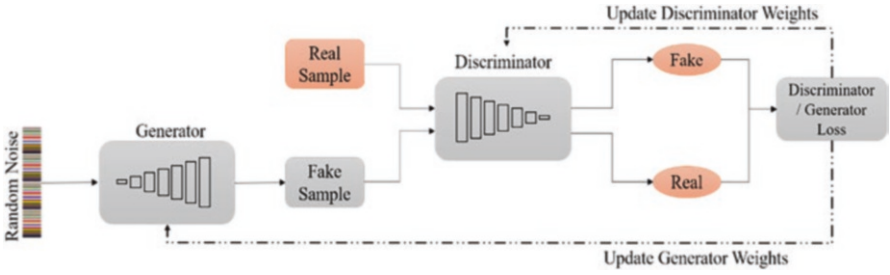

Fig. 1.11 GAN training

Various loss functions could serve the aim of GANs. The most used functions are *binary cross-entropy* loss, *mean squared error*, and the *Wasserstein loss*. The *binary cross-entropy* loss is a popular choice that is usually used to calculate the loss between the discriminator output and the real/fake label. The *binary cross-entropy* loss is measured using Eq. 1.1 [10]. The *mean squared error* is usually used to measure the difference between generated and real data samples. The equation of *mean squared error* is presented in Eq. 1.2 [11]. The *Wasserstein loss* is used in Wasserstein GANs (WGANs) to measure the distance between the generated and real data samples. The mathematical formulation for *Wasserstein loss* is shown in Eq. 1.3 [12].

$$\mathrm{BCE} = \frac{1}{n}\sum_{i=1}^{n} y_i \log\left(p\left(y_i\right)\right) + \left(1 - y_i\right)\log\left(1 - p\left(y_i\right)\right) \tag{1.1}$$

BCE is the *binary cross-entropy* loss, y is the label (either 1 or 0), p(y) is the predicted output, and n is the total number of samples.

$$\mathrm{MSE} = \frac{1}{\mathrm{n}}\sum\left(\mathrm{real} - \mathrm{fake}\right)^2 \tag{1.2}$$

MSE is the *mean squared error*, real is the real data sample values, fake is the generated data sample values, and n is the total number of samples.

$$Critic\_Loss = D\left(x\right) - D\left(G\left(z\right)\right) \tag{1.3}$$

D(x) is the average critic score on real images, D(G(z)) is the average critic score on generated images, and G(z) is the generator's output.

Reconstruction loss and *regularization* loss are other loss functions that can be used in GAN*s*. The first is utilized in variational autoencoder-GAN (VAE-GAN) [13] to measure the similarity between generated and real images. The latter prevents overfitting and improves the model's generalization by adding penalties to the network weights during training.

GANs Weaknesses

While GANs field is exciting, they experience some limitations and weaknesses. Some of the limitations are stability, diversity, and interpretability. These comprise difficulties in training, which can result in mode collapse or oscillation. In addition, GANs require a large amount of training data. Besides, bias can be introduced into the generated data due to the limitations and biases of the training data. Thus the real data representation will be unfair and misleading.

One of the challenges of evaluating GANs is the lack of concrete theoretical intrinsic evaluation metrics. The usual method of inspecting features across generated and real samples is an approximate and unreliable estimate. Furthermore, GAN

training can be unstable and time-consuming, requiring a degree of artistry. For example, mode collapse can result in the generator producing only a limited set of outputs. Lastly, GANs are not typically designed to be invertible, meaning it is impossible to determine the noise vector that maps onto a given image. However, new methods have emerged to address this issue. Some models do the opposite of the GAN or the implementation of cycleGANs that learn both directions simultaneously. Inversion can be helpful for image editing, allowing for applying controllable generation skills to any image's noise vector.

References

1. Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., Bengio, Y.: Generative adversarial networks. Commun. ACM. **63**, 139–144 (2020)
2. Karras, T., Laine, S., Aila, T.: A style-based generator architecture for generative Adversarial Networks. In: 2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), pp. 4401–4410 (2019)
3. Gulrajani, I., Ahmed, F., Arjovsky, M., Dumoulin, V., Courville, A.: Improved training of Wasserstein GANs. In: 31st International Conference of Neural Information Systems, pp. 5769–5779, USA (2017)
4. Radford, A., Metz, L., Chintala, S.: Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks. https://arxiv.org/abs/1511.06434 (2017)
5. Liu, M.-Y., Tuzel, O.: Coupled Generative Adversarial Networks. https://arxiv.org/abs/1606.07536 (2016)
6. Karras, T., Aila, T., Laine, S., Lehtinen, J.: Progressive Growing of Gans for Improved Quality, Stability, and Variation. https://arxiv.org/abs/1710.10196 (2017)
7. Brundage, M., Avin, S., Clark, J., Toner, H., Eckersley, P., Garfinkel, B., & Amodei, D.: The malicious use of artificial intelligence: Forecasting, prevention, and mitigation. arXiv preprint arXiv:1802.07228. https://arxiv.org/abs/1802.07228 (2018)
8. Hermosilla, G., Tapia, D.I.H., Allende-Cid, H., Castro, G.F., Vera, E.: Thermal face generation using StyleGAN. IEEE Access. **9**, 80511–80523 (2021)
9. Generative adversarial network: In *Wikipedia*. https://en.wikipedia.org/wiki/Generative_adversarial_network (2023)
10. Setia, M.: Binary Cross Entropy aka Log Loss-the cost function used in logistic regression, https://www.analyticsvidhya.com/blog/2020/11/binary-cross-entropy-aka-log-loss-the-cost-function-used-in-logistic-regression/
11. M, P.: End-to-end introduction to evaluating Regression Models, https://www.analyticsvidhya.com/blog/2021/10/evaluation-metric-for-regression-models/
12. Loss functions, https://developers.google.com/machine-learning/gan/loss
13. Larsen, A.B.L., Sønderby, S.K., Larochelle, H., Winther, Os.: Autoencoding beyond Pixels Using a Learned Similarity Metric. https://arxiv.org/abs/1512.09300 (2016)

Chapter 2
Your First GAN

Abstract This chapter will delve deeper into implementing generative adversarial networks (GANs). Chapter 1 presented a comprehensive overview of GANs, highlighting their versatility and promising results in various applications, including image, video, audio, and text generation. This chapter will focus on the most basic implementation of GANs, starting with the most straightforward code that generates a straight line "$y = x$". This approach will provide a fundamental understanding of the components and workings of GANs and how they can be utilized for 1D GAN. By the end of this chapter, the reader will better appreciate GANs and their capabilities.

2.1 Preparing the Environment

It is essential to prepare the environment properly before implementing a GAN. This process includes choosing the proper hardware and software requirements for implementing GANs. Hardware requirements are based on the size and complexity of the model, along with the size of the dataset. GANs typically require significant computational resources, especially for large-scale applications.

Hardware Requirements

The minimum hardware requirement for GAN implementation is a modern computer with a CPU with good speed processing and a dedicated GPU. These requirements provide enough computational power to train relatively small models on relatively small datasets. However, larger models or datasets may require a high-end computer or multi-GPU system. A higher-end GPU with more memory and a fast-processing speed will allow the GAN to process and train on larger datasets faster.

The amount of memory required is another crucial aspect to consider when training GANs. These models can consume significant memory resources, resulting in

S. Kaddoura, *A Primer on Generative Adversarial Networks*, SpringerBriefs in Computer Science, https://doi.org/10.1007/978-3-031-32661-5_2

slow training or memory-related problems. To avoid these issues, ensuring that the system to train GAN has adequate memory capacity is essential. Upgrading to a more powerful CPU and increasing the amount of RAM can significantly improve the training time. Although a minimum of 8GB RAM is recommended, it is always better to have more.

If the hardware requirements are unavailable, cloud-based platforms can be used. Amazon Web Services (AWS) [1], Google Cloud [2], and Microsoft Azure [3] provide access to GPUs and other resources required to train GANs. Additionally, there are online services such as FloydHub [4], Paperspace [5], and Google Colab [6] that offer free access to GPUs for a limited amount of time. These platforms are great for trying out GANs or running small projects.

Typically, training a GAN can take several hours to several days or even longer. The training duration depends on several factors, such as the size and complexity of the dataset, the architecture of GAN, the hardware used for training, the optimizer, and hyperparameter selection. These details will be explained later in this chapter.

Software Requirements

Once the hardware requirements for training a GAN have been satisfied, it is imperative to fulfill the software requirements. These requirements are divided into the programming language and the required libraries to implement the GAN model. It is vital to note that the choice of programming language and library will significantly impact the efficiency and ease of the implementation process. Popular programming languages for implementing GANs include Python, MATLAB, R, and C++. In addition to the programming language, it is vital to have a deep understanding of relevant deep-learning libraries such as PyTorch [7], Keras [8], MXNet [9], and TensorFlow [10]. The selection of these libraries should be based on the programming language used and the specific requirements of the GAN implementation, including the type of data being used and the desired output.

This book uses the Python programming language and the Keras library, which is integrated as part of the TensorFlow library. The implementation of a GAN model requires adhering to the following steps sequentially:

1. Install the required libraries, including TensorFlow and any other dependencies.
2. Import the necessary modules/packages and libraries into the programming environment.
3. Check if the imported libraries and modules are ready to be used.

Importing Required Modules and Libraries

One can use the package manager pip to install required libraries in Python. Figure 2.1 shows the syntax to install a library using the *pip* package manager.

```
pip install [library-name]
```

Fig. 2.1 Library installation syntax

```
1  # A module to transform data to NumPy arrays and create random noise
2  import numpy as np
3  from tensorflow.keras.models import Sequential
4  # Layers to be used in generator and discriminator
5  from tensorflow.keras.layers import Dense, LeakyReLU
6  # A module to plot generated data
7  import matplotlib.pyplot as plt
```

Fig. 2.2 Importing required libraries and modules

Implementing a 1D GAN necessitates the utilization of imperative libraries that are indispensable for numerical computing, data visualization, and data analysis and manipulation. By importing these libraries, the user can acquire the essential tools and functionalities required to implement the GAN model successfully. Figure 2.2 presents a detailed illustration of the procedures for importing these libraries into a Python environment.

Prepare and Preprocess the Dataset

Having a high-quality dataset is critical for the successful training of a GAN. A GAN model needs the information within the dataset to generate new data resembling real data. To create a straight line "$y = x$", one can generate two random variables with equal values evenly spaced over a specified interval. The same procedure can be followed for test data. Having good-quality data is a crucial step as it affects the quality of generated outputs and the stability of the training process.

In this chapter, the "$y = x$" dataset is prepared using NumPy linespace. The dataset comprises 1000 different values between [−1,1]. This data is formatted as a NumPy array. The data set is a single array comprising a stack of x and y sequences. The sequences of input arrays are placed horizontally (i.e., column-wise). Figure 2.3 illustrates a sample row of the dataset is shown.

This section will explore the process of preparing and preprocessing the "$y = x$" dataset for use in training the GAN model. Lines 2 and 4 in Fig. 2.4 present the first step in building a variable x with 1000 values between [−1, 1]. Then, creating a variable y equal to x is shown in line 6. A crucial aspect of this process is to combine both variables using horizontal stacking in a single array. These steps are demonstrated in lines 1 to 6 of the code presented in Fig. 2.4.

Fig. 2.3 Visualizing the first five rows in the dataset

X-Data	Y-Data
0.325325	0.325325
0.62963	0.62963
-0.173173	-0.173173

```
# Create variable X
X_train = np.linspace(-1, 1, 1000)
# Reshape X variable
X_train = X_train.reshape(-1, 1)
# Create variable Y such that y = x
y_train = X_train
```

Fig. 2.4 Create the dataset

2.2 Implementing the Generator

The generator produces two points, x and y, with equal values. It is implemented using a fully connected deep neural network. This network is trained to learn the underlying patterns in the training data in Fig. 2.4. After successful training, the generator can generate new examples that appear to belong to the training data. The implemented generator starts by passing the generated random noise vector through several layers to produce two output points.

The generator's architecture may vary depending on the specific problem being addressed. Generally, it comprises numerous up-sampling and fully connected layers to increase the output's resolution. In the 1D GAN model implemented, the generator is constructed using a sequential Keras model. It commences with an input layer that takes a random noise vector as input, with the noise dimension set to 100. While other values can be employed, a value of 100 is frequently used in GAN implementation literature. The input is then processed through a *Dense* layer with 16 units. A *LeakyReLU* activation function is then applied with an alpha value of 0.01 to handle negative input values.

In GANs, various activation functions, including *ReLU*, *Tanh*, *Sigmoid*, *Linear*, and *LeakyReLU*, can be employed. The activation function selection depends on the specific problem to be addressed. The final *Dense* layer has a *Linear* activation function and two output units. A *Linear* activation function is utilized in the output layer of the generator to generate continuous output values without any limiting range. In contrast to activation functions such as *Sigmoid* or *Tanh*, which restrict output values to a particular range, *Linear* activation allows the output values to take any real number value. Although the input data lies between -1 and 1, it can be altered to fit

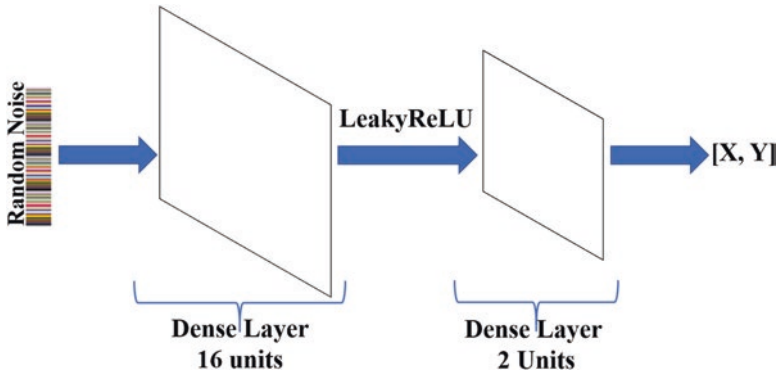

Fig. 2.5 GAN generator architecture

```
1   def build_generator(noise_dim):
2       # Add layers one after one in the sequence
3       model = Sequential()
4       # First fully connected layer
5       model.add(Dense(16, activation = 'relu'  kernel_initializer = 'he_uniform',
    input_dim = noise_dim))
6       # Add LeakyReLU activation function
7       model.add(LeakyReLU(alpha=0.01))
8       # Output layer with the shape of 2
9       model.add(Dense(2, activation = 'linear'))
10      return model
```

Fig. 2.6 Generator model

between any real value. Therefore, employing a *Linear* activation function in the output layer of the generator is deemed appropriate in this scenario. The complete implementation of the generator is shown in Fig. 2.5, demonstrating how the input noise is inputted and processed through the layers to generate two points where "$y = x$". The dense layer is represented with a 2D block because the dense layer outputs a 2D tensor where the first dimension represents the batch size and the second represents the output units. The dense layer will be represented with a 2D block for the rest of the chapters. The complete implementation of the code is presented in Fig. 2.6.

2.3 Implementing the Discriminator

The discriminator in GAN will evaluate the authenticity of the generated points and classify them as either real or fake. It is trained to predict the probability of whether given points are real (drawn from the actual dataset) or fake (generated by the generator). The discriminator is implemented using a fully connected deep neural

network with binary classification output. The input to the discriminator is trained on real data to maximize the probability of classifying it as real. The architecture of the discriminator is similar to that of the standard classifier, with multiple fully connected layers to extract features from input data points and to make the final classification. The activation function used in the discriminator can be *ReLU* or *LeakyReLU*, and the loss function is typically binary cross-entropy.

The construction of a GAN discriminator utilizes the sequential model of the *Keras* library. The process starts with adding a fully connected layer containing 32 units, followed by applying the *LeakyReLU* activation function with an alpha value of 0.01. The output tensor from the fully connected layer is then reduced by entering another fully connected layer with eight units. A LeakyReLU activation layer with an alpha value of 0.01 also follows this layer. Finally, a *Dense* layer with a *Sigmoid* activation function and a single unit is added to generate the final binary classification result. The architecture of the discriminator can be found in Fig. 2.6, while the complete implementation of the code is in Figs. 2.7 and 2.8.

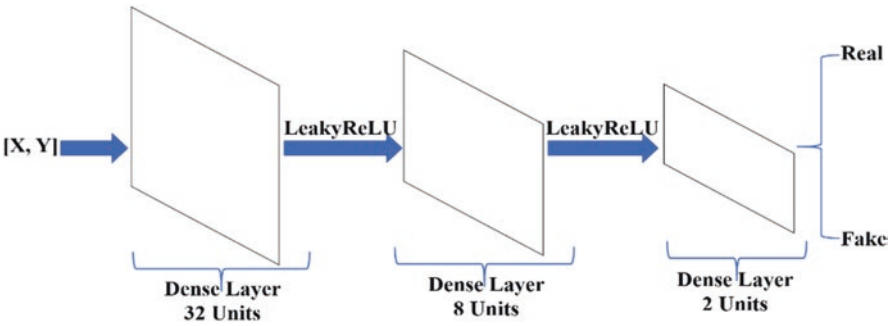

Fig. 2.7 GAN discriminator architecture

```
1   def build_discriminator():
2       # Add layers one after one in the sequence
3       model = Sequential()
4       # First fully connected layer
5       model.add(Dense(32, activation = 'relu', kernel_initializer = 'he_uniform', input_dim = 2))
6       # Add LeakyReLU activation function
7       model.add(LeakyReLU(alpha=0.01))
8       # Second fully connected layer
9       model.add(Dense(8, activation = 'relu', kernel_initializer = 'he_uniform'))
10      # Add LeakyReLU activation function
11      model.add(LeakyReLU(alpha=0.01))
12      # Output layer with the shape of 2
13      model.add(Dense(1, activation = 'sigmoid'))
14      return model
```

Fig. 2.8 Discriminator model

2.4 Training Stage

The GAN plays a game of probabilities between the generator and the discriminator. The generator tries to generate x and y that resembles real data, while the discriminator tries to distinguish real x and y points from the generated ones. The discriminator and the generator work in an adversarial manner, competing with each other in a two-player minimax game to improve their abilities over time. The generator improves its ability to generate points that can fool the discriminator. The discriminator improves its ability to distinguish it from real points generated to draw a straight line. The goal is to achieve a state where the generator produces random points capable of drawing a straight line.

Model Construction

Before training the GAN model and custom the training loop, it is necessary to construct the model and create loss functions. A GAN comprises two separate models: a generator and a discriminator, combined into a third model called the adversarial. The discriminator has been placed on top of the generator to custom the adversarial model. Optimization algorithms such as *Stochastic Gradient Descent (SGD)*, *Adam Optimizer*, *RMSprop*, and *Adagrad* can be used for GAN training. *Adam Optimizer* is commonly used, while *RMSprop* and *Adagrad* can be used if *SGD* struggles to converge. Two loss functions are used for GAN training, one for the generator model and another for the discriminator model. The generator tries to minimize the loss, while the discriminator tries to maximize it. The generator's loss function is typically *binary cross-entropy loss*, while *binary cross-entropy loss* or *hinge loss* can be used for the discriminator model. The discriminator and adversarial models are compiled with the *Adam optimizer* and *binary cross-entropy loss*. The code snippets for constructing the models are displayed in Fig. 2.9. To construct the adversarial model, called GAN in line 10; the discriminator is set as not trainable first, as shown in line 9. This means that only the generator will be updated during the adversarial model training.

Loss Function

The next step is to create the loss functions used during the training process to compute discriminator and generator losses. Figure 2.10 shows the complete implementation of loss functions. In this stage, the *train_on_batch* function, presented in lines 4 and 8, trains a model on a single batch of data with its corresponding labels. Then the model parameters are updated based on the loss calculated between the predicted and true labels. The discriminator loss function takes data X, a combination of real and fake data points, and their labels y as input. The generator loss takes the noise generated with real labels as input.

```
1  def construct_models(noise_dim):
2      # Build the discriminator model
3      discriminator = build_discriminator()
4      # Compile the discriminator
5      discriminator.compile(loss = 'binary_crossentropy' , optimizer = 'adam', metrics = ['accuracy'])
6      # Build the generator model
7      generator = build_generator(noise_dim)
8      # Freeze the discriminator during generator training
9      discriminator.trainable = False
10     gan = Sequential()
11     # Pit the generator and discriminator against each other
12     gan.add(generator)
13     gan.add(discriminator)
14     # Compile the GAN model
15     gan.compile(loss = 'binary_crossentropy', optimizer = 'adam', metrics = ['accuracy'])
16     return generator, discriminator, gan
```

Fig. 2.9 Model construction

```
1  # Define discriminator loss function
2  def disc_loss(model, X, y):
3      # Train the model on a batch of data and return the loss value
4      return model.train_on_batch(X,y)
5  # Define generator loss function
6  def gen_loss(model, X, y):
7      # Train the model on a batch of data and return loss value
8      return model.train_on_batch(X,y)
```

Fig. 2.10 Loss functions

Plot Generated Data Samples

Prior to initiating the training of the GAN model, it is necessary to establish a function that facilitates the visualization of the generated points. This function will be called during the training phase to evaluate the model's performance based on the quality of the generated points. Figure 2.11 displays the code for data visualization. Generated and actual points are plotted using a scatter plot, as shown in lines 9 and 10. It will plot all samples where real ones are represented in red and generated ones are represented in blue. Figures 2.16, 2.17, 2.18, and 2.19, displayed later in the chapter, are drawn using this function.

```
1  def print_generated_samples(batch_size, noise_dim):
2      # Create test data
3      X_test = np.random.uniform(low=-1, high =1, size=(batch_size//2, 1))
4      y_test = X_test
5      # Generate samples
6      noise = np.random.normal(0, 1, (batch_size//2, noise_dim))
7      gen_data = generator.predict(noise)
8      # Plot generated data
9      plt.scatter(X_test, y_test, color = 'red')
10     plt.scatter(gen_data[:,0], gen_data[:,1], color =  'blue')
11     plt.xlim(-1,1)
12     plt.ylim(-1,1)
13     plt.legend(["real data", "generated data"], loc = "lower right")
14     plt.show()
```

Fig. 2.11 Data visualization

Training GAN

In the training of GAN models, the utilization of epochs and batches is a widespread approach. The number of epochs determines the time the model is exposed to the training data, with a higher number leading to improved performance and a higher risk of overfitting. On the other hand, the batch size determines the portion of the training data used for updating the model's weights, affecting both the training speed and stability. A larger data size generally leads to better performance but requires more computing resources and processing time.

In each epoch, the generator and discriminator are trained alternately using a batch of the real and generated data points. This process is repeated for multiple epochs (chosen to be 600) to optimize the weights and achieve desired performance. The number of batches per epoch and the number of epochs is determined based on data size, model complexity, and the target performance. The complexity of the model, such as the number of layers and nodes, also affects its performance. A more complex model can achieve better results but may be more susceptible to overfitting.

The training stops when the generator produces y and x samples indistinguishable from real data points. In this case, the generator successfully deceives the discriminator with high accuracy. The model performance will be visualized by plotting the point in a 2D axis showing increasing epochs. Figures 2.12, 2.13, and 2.14 represent the discriminator's training stage, the training stage of the generator, and the training loop. The training process can be described as follows:

1. Generate a random normal noise as shown in line 5 in Fig. 2.14.

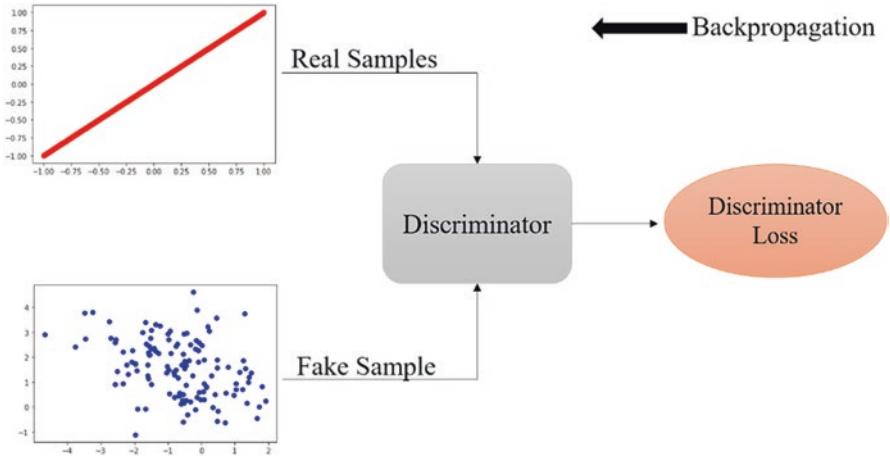

Fig. 2.12 Training stage in the discriminator

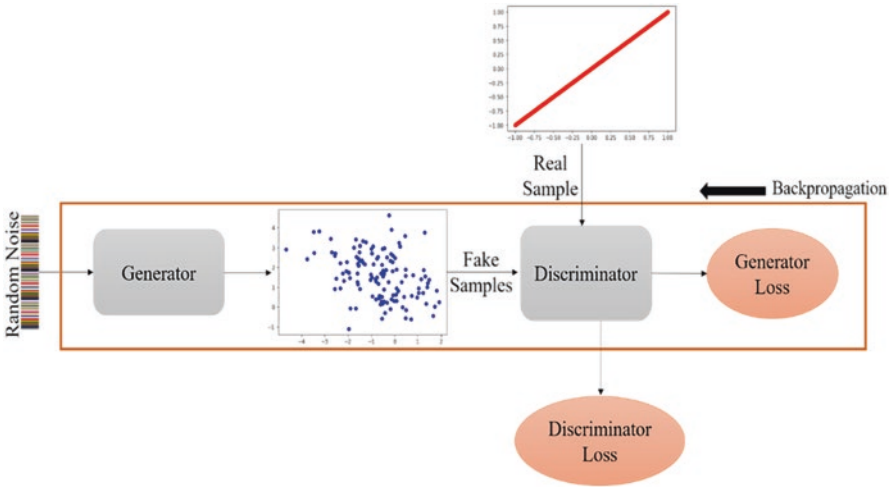

Fig. 2.13 Training stage in the generator

2. Generate fake data points from this noise through the generator as in line 7 in Fig. 2.14.
3. Sample a batch of the real data points from training data; refer to line 11 in Fig. 2.14.
4. Create labels for real data samples as 0.9 (line 13) and generated data samples as zeros as in line 15 in Fig. 2.14.
5. Train the discriminator and compute the loss in both generated and real data with their correct labels refer to lines 16 and 19 in Fig. 2.14. Figure 2.12 shows the discriminator training.
6. Generate another number of random data samples in line 23 in Fig. 2.14.

```
1  def training(generator, discriminator, gan, noise_dim, epochs, batch_size):
2      # Enumerate over epochs
3      for e in range(epochs):
4      # Random normal array for generator input
5      noise = np.random.normal(0, 1, (batch_size, noise_dim))
6      # Create fake data points by the generator
7      fake_samples = generator.predict(noise)
8      # Stack X and Y variable horizontally to build the dataset
9      real_data = np.hstack((X_train, y_train))
10     # Get random real data points from the training data
11     real_data = real_data[np.random.randint(0, real_data.shape[0], size = batch_size)]
12     # Create real labels
13     real_labels = np.ones((batch_size, 1))*0.9
14     # Generate fake labels generated data points as zeros
15     fake_labels = np.zeros((batch_size, 1))
16     # Calculate the loss of real data points
17     discriminator_loss_real = disc_loss(discriminator, real_data, real_labels)
18     # Calculate the loss of generated data points
19     discriminator_loss_fake = disc_loss(discriminator, fake_samples, fake_labels)
20     # Compute total discriminator loss
21     discriminator_loss = 0.5 * np.add(discriminator_loss_real, discriminator_loss_fake)
22     # Generate random points as input for the generator
23     x_gan = np.random.normal(0, 1, (batch_size, noise_dim))
24     # Generate real labels for GAN
25     y_gan = np.ones((batch_size, 1))
26     #calculate the generator loss
27     gan_loss = gen_loss(gan, x_gan, y_gan)
28     # Print the progress
29     if e % 10 == 0 or e == epochs-1:
30        print( 'Epoch: ', e, ' Generator Loss: ', gan_loss , ' Discriminator Loss: ', discriminator_loss)
31        print_generated_samples(batch_size, noise_dim)
```

Fig. 2.14 GAN training process

7. Train the GAN model on the newly generated data samples with labels equal to one to trick the discriminator, as represented in line 27 in Fig. 2.14. The generator training architecture is presented in Fig. 2.13.
8. Repeat steps 1 to 7 for several epochs to improve the generator and discriminator further.

In order to launch the training process, it is necessary first to call the training function and construct the models. Figure 2.15 outlines the code written to start training. Within this figure, it is also essential to set the batch size and the number of epochs, as they are crucial parameters in the training process.

As previously mentioned, the performance of the generative model during the training process is closely related to the number of training iterations, commonly referred to as epochs. With each increasing epoch, the generated data points become more realistic and resemble the training data. This is due to the improvement of the generator's ability to fool the discriminator. Multiple examples are presented to

```
# Construct the models
generator, discriminator, gan = construct_models(100)
# Train GAN
training(generator, discriminator, gan, 100, 600, 128)
```

Fig. 2.15 Calling the training function

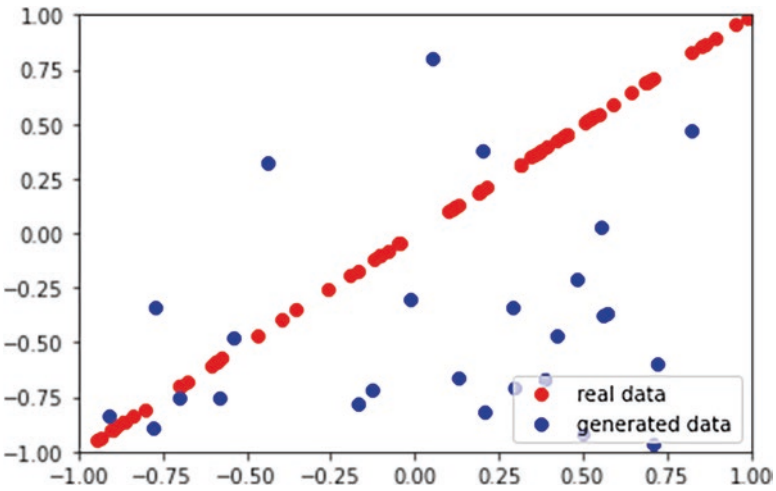

Fig. 2.16 Generated points at epoch 0

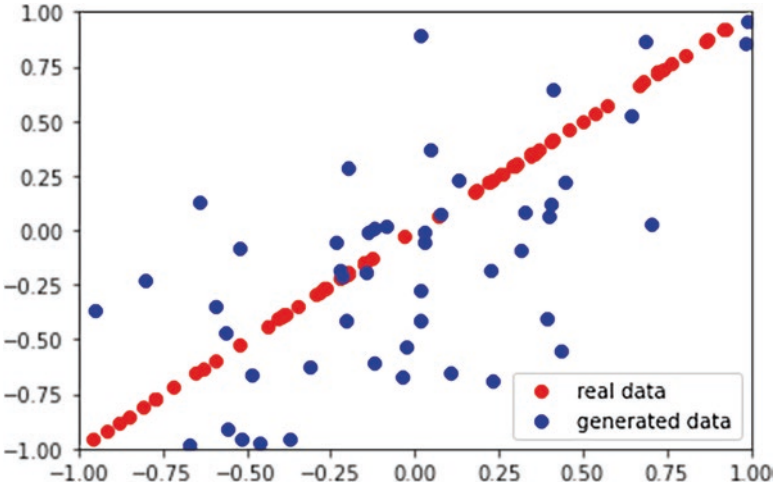

Fig. 2.17 Generated points at epoch 200

highlight the relationship between the increasing epochs and the generated data points. Figures 2.16, 2.17, 2.18, and 2.19 demonstrate the progress of the generated data points as the resolution improves. One can observe the improvement in the

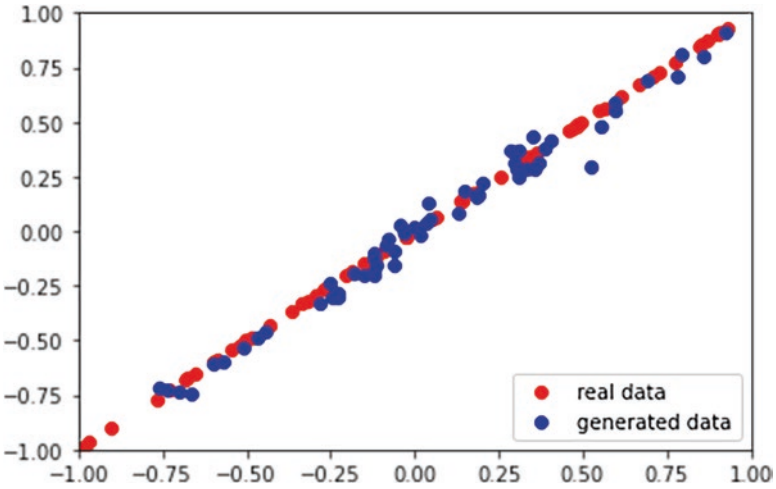

Fig. 2.18 Generated points at epoch 400

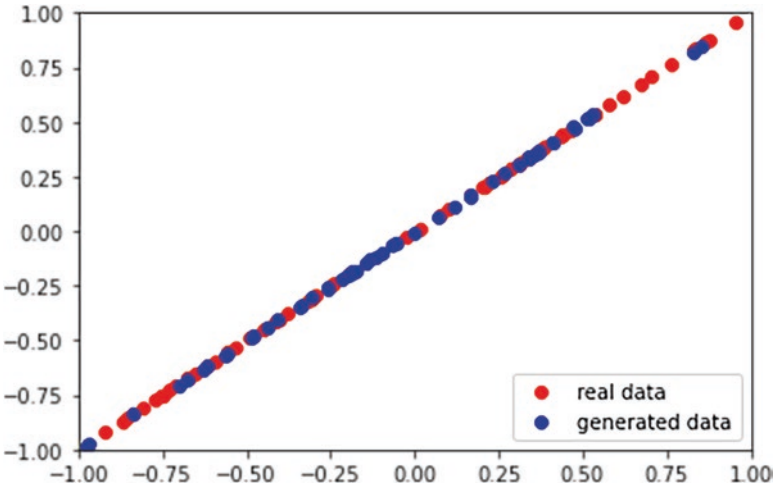

Fig. 2.19 Generated points at epoch 600

generator's ability to deceive the discriminator by comparing Figs. 2.16 and 2.19, for instance, and comparing the blue line (generated data points) with the red one (real data points).

Common Challenges While Implementing GANs

Several obstacles and errors can arise while building and training a GAN model. These include model collapse, where the generator produces limited output and fails to capture the diversity of the data. The instability of the model training, where the

generator and discriminator oscillate instead of converging to a stable solution, is another common error. Additionally, the vanishing gradient problem, where the gradients in the training process become too small, can also hinder the model's training. A common obstacle is overfitting, where the model is too complex and memorizes the training data. Selecting an appropriate model architecture, hyperparameter tuning, and regularization techniques can help resolve these issues.

References

1. Cloud computing services - amazon web services (AWS), https://signin.aws.amazon.com/
2. Google Cloud - Google, https://cloud.google.com/
3. Cloud computing services: Microsoft Azure, https://azure.microsoft.com/en-us
4. FloydHub blog, https://blog.floydhub.com/
5. Cloud computing, evolved, https://www.paperspace.com/
6. Google Colab, https://colab.research.google.com/
7. Pytorch, https://pytorch.org/
8. Team, K.: Simple. flexible. powerful, https://keras.io/
9. Apache MXNet, https://mxnet.apache.org/
10. Tensorflow, https://www.tensorflow.org/

Chapter 3
Real-World Applications

Abstract This chapter explores the practical applications of GANs in the real world. As previously discussed, GANs are deep neural networks capable of generating new data resembling a target distribution, such as images and audio signals. GANs have garnered much attention in machine learning in recent years due to their versatile usage across various industries.

This chapter will delve into the various ways in which GANs have been employed in real-world applications. They include but are not limited to the generation of synthetic data for training machine learning models, the creation of photorealistic images and videos, the generation of human faces, the production of fake videos, image-to-image translation, text-to-image translation, CycleGAN, enhancement of image resolution, semantic image inpainting, and text-to-speech.

A comprehensive overview of the exciting possibilities of GANs and their impact on various industries will be provided. By the end of the chapter, the reader will clearly understand the practical applications of GANs and their potential to shape the future.

3.1 Human Faces Generation

GANs have emerged as a popular method for image generation, using two deep neural networks: a generator and a discriminator. The generator creates realistic images that can fool the discriminator. The discriminator tries to classify images accurately as real or fake. This competition leads to generating highly realistic images, including natural landscapes, animals, abstract art, and synthetic data for training machine learning models in computer vision.

However, one of the most challenging tasks in image generation using GANs is the creation of realistic human faces. Human faces have a high degree of complexity, including subtle variations in color, texture, and shape. This task requires a complex generator model to ensure the production of high-resolution face images.

The practical applications of generating realistic human faces include virtual characters in the gaming and entertainment industry, biometric authentication

S. Kaddoura, *A Primer on Generative Adversarial Networks*, SpringerBriefs in Computer Science, https://doi.org/10.1007/978-3-031-32661-5_3

systems, and data augmentation to increase the size of training datasets, improving facial recognition systems' performance. GAN-generated human faces can preserve privacy and anonymity in applications where real human faces may not be suitable. It also leads to new forms of self-expression and can be used in scientific studies and experiments where large datasets of real faces may not be feasible.

Generating realistic and high-quality human faces using GANs requires significant data and computational resources. The minimum requirements for training a GAN model for human face generation include a large dataset of high-resolution human face images, sufficient computational resources, a well-designed GAN architecture optimized for generating high-quality human faces, proper tuning of hyperparameters, and a generator model with enough capacity to learn the complex patterns and variations in human faces.

Implementing GANs for generating human faces requires a meticulous approach, as the architecture of GANs is complex and demands a large volume of high-quality training data. The typical process of generating human faces using GANs entails the following steps: data collection and preparation, model design, training, evaluation and refinement, and deployment. The capability of GANs to generate realistic human faces makes it a fascinating and promising development area.

Any GAN-based image generation model can be used to generate human faces. However, the quality of the generated faces may not be as good as those produced by a GAN model designed explicitly for human face generation, as human faces have unique complex features. General GAN models can achieve promising results in generating human faces.

In order to generate human faces, one approach is to use any image generation model with a human faces dataset. There are several publicly available datasets, including "Generate Realistic Human Face using GAN" [1] and "Generating Fake Faces using GANs" [2], that can be used for human face generation. This section will present a general simple DCGAN model that generates handwritten images of the letter A. This model can be considered a base GAN model for human face generation. It can be improved to generate high-resolution human face images because human faces possess unique and complex features.

The first step in implementing any machine learning model is to import the required modules. Figure 3.1 includes all the required modules for implementing image generation GAN.

Data Collection and Preparation

The first step in generating handwritten images of the letter "A" GANs entails the collection of a substantial dataset of real handwritten letters. This dataset will serve as the basis for training the GAN. The "A-Z Handwritten Alphabets" dataset, which is readily accessible to the public [3], is used. The dataset comprises all the alphabets stored in CSV files containing each image's pixel values. The images in the dataset are in grayscale, with a single channel and a size of 28x28. The pixel values

```
1  from tensorflow.keras.models import Model, Sequential
2  # Layers to be used in the generator and discriminator
3  from tensorflow.keras.layers import Input, Dense, Conv2D, Flatten, Dropout, Conv2DTrans-
   pose, Reshape, MaxPooling2D, BatchNormalization, LeakyReLU
4  # For an optimal set of weights and biases that minimize the loss function
5  from tensorflow.keras.optimizers import Adam
6  # Transform data to NumPy arrays and create random noise
7  import numpy as np
8  # To load data
9  import pandas as pd
10 # To plot generated images
11 import matplotlib.pyplot as plt
12 # To map numeric values to strings
13 import string
```

Fig. 3.1 Importing required libraries and modules

	0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	...	0.639	0.640	0.641	0.642	0.643	0.644	0.645	0.646	0.647	0.648
0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
2	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
3	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
4	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

5 rows × 785 columns

Fig. 3.2 Visualizing the first five rows in the dataset

of the images in the dataset represent the intensity of each pixel, with values ranging from 0 (black) to 255 (white). There are 28 x 28 = 784 pixels in a 28 x 28 grayscale image. By utilizing this dataset, the implementation process is simplified. Figure 3.2 illustrates sample rows of the dataset, with the first column representing the image number and the remaining columns representing the pixels. Utilizing this publicly available dataset is an alternative to collecting a large dataset from scratch.

This section will explore the process of preparing and preprocessing the "A-Z Handwritten Alphabets" dataset to be used in the training of the GAN model. Figure 3.3 presents the first step in this process: loading the dataset using the Pandas module. Loading data is followed by a sequence of preprocessing procedures designed to format and normalize the images to align with the model's requirements. A crucial aspect of this process is data visualization, which encompasses plotting the data in various forms to comprehend the distribution of pixel values and detect any anomalies that may hinder the model's performance. A visual representation for printing some rows in the training dataset is noted in Line 6 of the code of Fig. 3.3.

In the pursuit of implementing a simple GAN model capable of generating the alphabet "A," it is necessary to preprocess the data in a way that removes all other alphabets and retains only this letter. This step is demonstrated in lines 7 to 14 of the code presented in Fig. 3.3.

```
# Import the data "A_Z Handwritten Data.csv" from csv
dataset = pd.read_csv('A_Z Handwritten Data. csv').astype('float32')
# Define image dimensions
width, height, channel = 28, 28, 1
# Visualize the first 5 rows
print(dataset.head())
# Create a dictionary to map numbers to the alphabets
letters = dict(zip(range(0, 26), string.ascii_uppercase))
# Rename first column as letters by index
dataset.rename(columns={dataset.columns[0]: 'letters'},inplace=True)
# Map numerical values in letters column to alphabets using letters dict.
dataset['letters'] = dataset['letters'].map(letters)
# Create a new data frame that contains images of the letter A only
letter_A_images = dataset[dataset['letters'] == 'A']
# Remove 1st column in the data since it is the image labels
data = letter_A_images.iloc[:,1:].to_numpy()
# Reshape the data as the shape of the image
data = data.reshape((len(data), width, height))
# Normalizing the data between [0,1] by dividing each pixel by the total number of pixels
data = data/255
# Shuffle the data to randomize order
np.random.shuffle(data)
```

Fig. 3.3 Load and pre-process the dataset

The same data preprocessing steps will be followed to generate a GAN model capable of producing all human faces. If the data is given as an image instead of numerical pixels in Fig. 3.2, it must be transformed into a floating-point tensor.

Model Design

The next step is to design the GAN model. GANs consist of two main components: a generator and a discriminator. The generator generates handwritten images of the letter A that resemble the real data in [3]. It is implemented using a deep neural network. This network is trained to learn the underlying patterns in the training data [3]. The discriminator in GAN will evaluate the authenticity of the generated handwritten images of letter A and classify it as either real or fake. It is trained to predict the probability of whether a given image is real (drawn from the actual dataset [3]) or fake (generated by the generator).

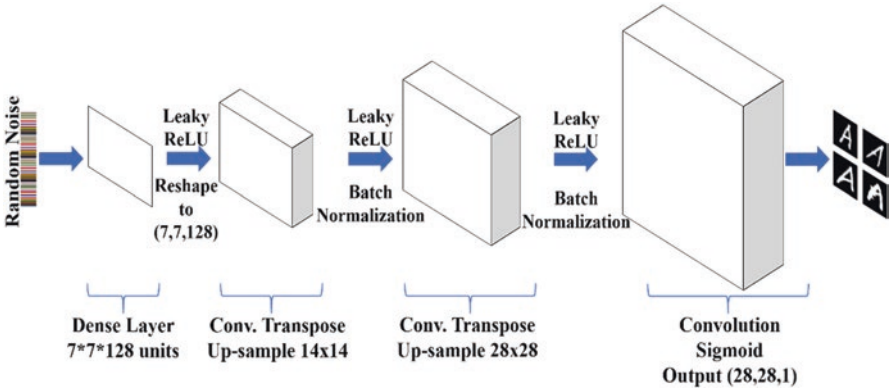

Fig. 3.4 GAN generator architecture

The Generator Model

A combination of fully connected and Convolutional Transpose layers was used in the case of the "A" alphabet GAN generator to enable the generator to generate high-resolution images. The Convolutional Transpose is also known as Deconvolution, a prevalent method for up-sampling low-resolution representations. During training, the transpose layer acquires weights that enable the generator to produce letter A images similar to those in the training dataset.

The complete implementation of the generator is shown in Fig. 3.4. It demonstrates how it starts by passing the generated random noise vector through several layers to produce an output image of the letter A. Unlike dense layer representation, the convolution transpose layer is represented by a 3D block. This representation is chosen because the transposed convolutional layer outputs a 3D tensor in the case of image generation, similarly for the convolutional layer. This representation will be used in the following chapters.

The complete implementation of the code is presented in Fig. 3.5. The generator in the GAN model is implemented using a sequential model in TensorFlow. It begins with a *Dense* input layer (Line 5), which inputs a random noise vector. The input is then processed through a *LeakyReLU* activation function (Line 7) with an alpha value of 0.01 to handle negative input values. The shape of the input is reshaped to 7x7x128 (Line 9), and padding is applied to the transpose convolutional layers.

The model includes two *Conv2DTranspose* layers for up-sampling the input to 28x28, with batch normalization and *LeakyReLU* activation functions in between (lines 11 to 13). Different activation functions, such as *ReLU*, *Tanh*, *Sigmoid*, and *LeakyReLU*, can be used in GANs, and the choice depends on the problem being solved. In this case, the *LeakyReLU* activation function is used in Convolution Transpose layers to avoid the "dead neurons" problem. In traditional *ReLU* activation, if the input is negative, the output will be 0, resulting in the corresponding

```
1  def build_generator():
2      # Add layers one after one in the sequence
3      model = Sequential()
4      # First fully connected layer
5      model.add(Dense(7*7*128, use_bias=False, input_shape=(100,)))
6      # Add LeakyReLU activation function
7      model.add(LeakyReLU(alpha=0.01))
8      # Reshape the output to be suitable for the Conv2DTranspose layer
9      model.add(Reshape((7, 7, 128)))
10     # Up-sample to 14x14
11     model.add(Conv2DTranspose(128, (1, 1), strides=(2, 2), padding='same'))
12     model.add(BatchNormalization())
13     model.add(LeakyReLU(alpha=0.01))
14     # Up-sample to 28*28
15     model.add(Conv2DTranspose(128, (2, 2), strides = (2, 2), padding = 'same', activation = 'sigmoid'))
16     model.add(BatchNormalization())
17     model.add(LeakyReLU(alpha=0.01))
18     # Output layer with the shape of image w*h*c = 28*28*1
19     model.add(Conv2D(1, (5,5), activation = 'sigmoid', padding = 'same'))
20     return model
```

Fig. 3.5 Generator model

neuron not contributing to the learning process during training. However, as the data is normalized between 0 and 1, *LeakyReLU* and *ReLU* can be used.

The model ends with a *Conv2D* layer (Line 19) as the output layer, with a 5x5 kernel size and *Sigmoid* activation function. The output layer has a shape of 28x28x1, representing an image's height, width, and depth.

The Discriminator Model

The discriminator is implemented using convolutional neural network (CNN) with binary classification output. The input to the discriminator is trained on real data to maximize the probability of classifying it as real. The architecture of the discriminator is similar to that of the standard image classifier. A typical image classifier consists of convolutional and pooling layers to extract features from input images and fully connected layers to make the final classification. The activation function used in the discriminator can be *ReLU* or *LeakyReLU*, and the loss function is typically binary cross-entropy.

Figure 3.6 displays the discriminator's architecture. The figure shows the input images and output labels to the discriminator. The complete implementation of the code is presented in Fig. 3.7. The TensorFlow library's sequential model is used to construct a GAN discriminator. The process begins by adding a convolutional layer with 128 filters (Line 5) and applying the *LeakyReLU* activation function with an alpha value of 0.02 (Line 6). The output tensor from the convolutional layer is then

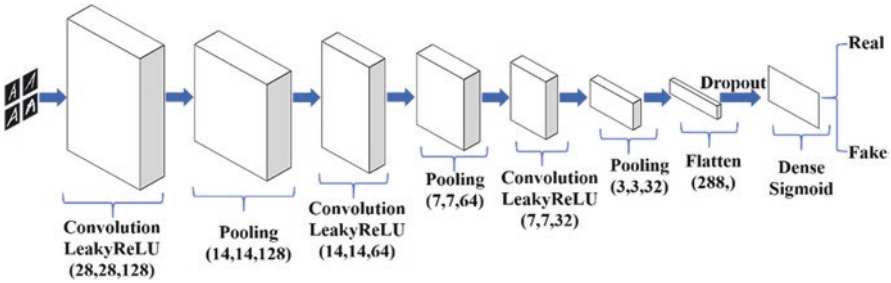

Fig. 3.6 GAN discriminator architecture

```
1   def build_discriminator():
2       # Add layers one after one in the sequence
3       model = Sequential() #to add layers one after one in the sequence
4       # Conv2D layer with 128 filters and output (28, 28, 128)
5       model.add(Conv2D(128, (2, 2), strides=(1, 1), padding = 'same', input_shape = [28, 28, 1]))
6       model.add(LeakyReLU(alpha=0.02))
7       # Decrease the height and the width of the output tensor by downsampling to 14*14
8       model.add(MaxPooling2D(2,2))
9       # Conv2D layer with 64 filters and output (14, 14, 64)
10      model.add (Conv2D(64, (2, 2), strides=(1, 1), padding='same'))
11      model.add(LeakyReLU(alpha=0.02))
12      # Down sample to 7*7
13      model.add(MaxPooling2D(2,2))
14      # Conv2D layer with 32 filters with output (7, 7, 32)
15      model.add(Conv2D(32, (2, 2), strides=(2, 2), padding='same'))
16      model.add(LeakyReLU(alpha=0.02))
17      # Down sample to 3*3
18      model.add(MaxPooling2D(2,2))
19      # Convert the tensor into a 1D vector with output (288,)
20      model.add(Flatten())
21      # Prevent overfitting
22      model.add(Dropout(0.5))
23      # Produce binary classifier
24      model.add(Dense(1, activation='sigmoid'))
25      return model
```

Fig. 3.7 Discriminator model

reduced using MaxPooling2D with a stride of 2 (Line 8). This sequence is repeated twice with filters for convolutional layers 64 and 32. Following the final MaxPooling2D layer, the output tensor is flattened and transformed into a 1D vector through the *Flatten* layer (line 20). Then, a *Dropout* layer is used with a rate of 0.5 to mitigate overfitting. Finally, a *Dense* layer (Line 24) with a single unit and a *Sigmoid* activation function is added to generate the final binary classification result.

Training

Once the model has been designed, it must be trained on the handwritten images of the letter "A" dataset. During training, the generator produces letter A images, and the discriminator evaluates each one, providing feedback to the generator on how to improve. This back-and-forth process continues until the generator produces faces indistinguishable from real human faces. The human handwritten letter A generation training process using GANs follows the same procedure described in Fig. 2.14 of Chap. 2. Start by constructing the models, creating loss functions, plot functions, and custom the training loop.

Fig. 3.8 presents a comprehensive framework for implementing GANs for letter A generation. It provides a clear illustration of the training process of the model. The generator and the discriminator work together in a continuous loop. The generator produces fake images, and the discriminator determines the authenticity of these images. As the model trains, the generator and the discriminator improve their performance. The generator becomes better at producing images that fool the discriminator. The training process is demonstrated through backpropagation. In this way, the model continually improves its performance and becomes better at producing high-quality human-like faces.

During the training process, a batch size of 128 was used, and the training was repeated for 200 epochs. To assess the performance of the GAN model in generating the letter A, 100 generated samples were produced every ten epochs. Figs. 3.9, 3.10, 3.11, and 3.12 depict the model's progress in generating the letter A as the number of epochs increases. The results show that the model can generate clear images of the letter A in the later epochs.

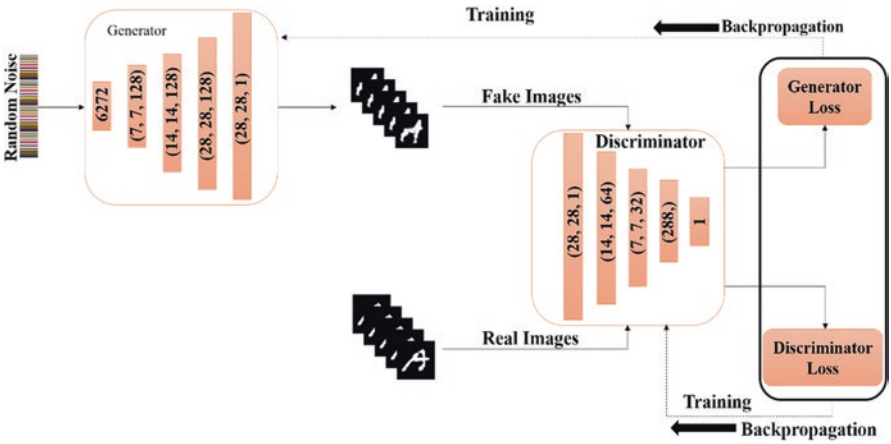

Fig. 3.8 Implemented framework of face GAN

Fig. 3.9 Generated images at epoch 60

Fig. 3.10 Generated images at epoch 10

Evaluation and Refinement

The generator and discriminator, in Figs. 3.5 and 3.7, respectively, are implemented for generating the handwritten letter "A," demonstrating their capability in image generation tasks. However, the complexity of the model architecture may vary

Fig. 3.11 Generated images at epoch 130

Fig. 3.12 Generated images at epoch 200

based on the difficulty of the task at hand. When generating human faces, it may be necessary to fine-tune the model by modifying parameters and adding additional layers.

After training, the GAN must be evaluated to determine its performance. The first step is to train the letter A generation GAN model on a human face dataset. When training is completed, the performance will be evaluated. Model evaluation

Fig. 3.13 Random baby face generated using GANs

typically involves the comparison of generated faces to real human faces and the evaluation of metrics such as visual quality, diversity, and consistency. The implementation of the model may not be optimal from the initial attempt. Thus it may be necessary to refine the GAN by making changes to its architecture or retraining it on a larger dataset.

The evaluation and refinement of the model result in images that cannot be differentiated from real human faces. Figure 3.13 presents fake images generated by the GAN, which cannot be distinguished from real human images. These images are produced by the AI face generator powered by StyleGAN, a neural network developed by Nvidia in 2018 [4].

The discriminative model illustrated in Fig. 3.6 also applies to any binary-output image generation GAN model. Thus, it can be utilized for generating faces. However, due to the difference in size and structure of the data, the architecture might need to be modified, such as adjusting the hyperparameters or increasing the number of layers. The generative model also is capable of generating images from noise.

The output size of the generator needs to be adjusted to match the size of the training data images. As shown in Fig. 3.5, the generator generated images of size 28x28x1. However, human faces are typically represented as RGB images, so the generator's output size must be adjusted to accommodate the width, height, and three-color channels.

Deployment

The final step is to deploy the trained GAN in a practical application. Practical deployment of human face generation using GANs involves several steps, including training a GAN model, deploying the trained model to a production environment, and integrating it into an application. Once the model has been trained, it can be deployed to a production environment such as a cloud platform like AWS or Google Cloud or run on a local server.

Once the model has been deployed, it can be integrated into an application through APIs or embedding it. For example, the generated human faces could be used in a virtual try-on application for cosmetics, or they could be used to generate characters in a video game. The exact integration method will depend on the specific use case and the application's requirements.

It is important to note that the steps for generating human faces using GANs have been outlined in this section. These steps include the acquisition of appropriate datasets and the design of the model architecture. Numerous online resources provide code for human face generation using libraries such as PyTorch, TensorFlow, and Keras. Additionally, the model presented for image generation can be fine-tuned to suit individual needs and requirements.

3.2 Deep Fake

Deep fakes refer to media files that have been manipulated and can be used to spread false information or create fake news. Despite their potential positive applications in many fields, such as entertainment and virtual assistants, the negative implications of deep fakes far outweigh the positive ones. Deepfakes can be utilized to manipulate public opinion, spread misinformation during elections, create fake news stories, and even blackmail individuals. These actions can have significant and detrimental consequences for society, including eroding trust in information sources and disrupting public discourse.

Several measures need to be taken to prevent the misuse of deep fakes. Detection techniques must be developed for authenticating media content. Media literacy and critical thinking skills must be promoted among the public. Additionally, laws and regulations should be implemented to deter the creation and distribution of deep fakes for malicious purposes. Addressing the potential harms of deep fakes is crucial in maintaining the integrity of information and ensuring a healthy democracy.

Deep fakes generation using GANs is a common practice. GANs can create realistic images and videos of individuals who do not exist or manipulate existing media files to create deep fakes. In a GAN, the generator network creates images or videos, while the discriminator network distinguishes between real and fake media files.

Data Collection and Preparation

Various GAN models can be used in deep fake video generation, such as speech-to-video or video-to-video GANs. Speech-to-video deep fake GANs mean generating videos of talking faces based on audio files and images of the target. On the other hand, video-to-video GANs involve generating counterfeit videos for a target individual with source and target person as a requirement. Thus, GAN will swap faces and voices in a video. In this section, video-to-video GANs will be explained.

To create a deep fake video, a dataset of videos of a real person is fed into the GANs. One way is to collect a large and diverse dataset of real videos, which can be the basis for generating fake videos. The dataset should have sufficient variability regarding different viewpoints, lighting conditions, backgrounds, and other relevant factors. Additionally, the dataset should be annotated to facilitate training and evaluation of the model. The dataset must contain different videos of the source and target speaker. The vocals and image of target speaker B should replace the vocals and image of source speaker A. Another way is to use public datasets. Various video datasets are available for this purpose, such as the FaceForensics++ dataset [5], which contains multiple individuals' real and deep fake videos. The VoxCeleb dataset [6] is another viable option. It includes over 1,000 hours of audio and video recordings of individuals, making it suitable for training deep fakes that involve audio and video. There are various datasets online; however, VoxCeleb serves the problem. It consists of short clips of human speech extracted from interview videos uploaded to YouTube.

Once the dataset is collected, it must be preprocessed and formatted for training the GAN model. This includes resizing the videos to a consistent resolution, normalizing the pixel values, cropping the videos to remove unwanted parts of the frame, such as black borders, and splitting the videos into individual frames. Splitting the videos into individual frames is essential since working with entire videos can be computationally intensive and time-consuming. The frames can be further augmented to increase the variability in the training data by applying random rotations, zooms, and flips.

Model Design

Compared to image GANs, video GANs require different treatments due to the complexity of video data, which includes multiple images and the additional time dimension. There are four strategies commonly used by video GAN generators: RNN architecture combined with 2D convolutional networks [7], 3D convolutional networks [8], coarse-to-fine strategy [9], and two-stream architecture [8, 10].

An encoder-decoder pair is required to create a deep fake video generator. Deep learning models consist of layers, each representing a mathematical abstraction of the previous one, resulting in a latent representation. The encoder extracts latent features from the source data (source videos), while the decoder uses this information to reconstruct new data [11].

The discriminator in video GANs may use the same strategy as the generator. It also might adopt other strategies, such as two-stream or a 3D convolutional network [8, 10]. In deep fakes, the goal is to generate a talking person with a different face than the input face while maintaining the motion from the input video. This has been proposed in previous research, where the model disentangles the video representations of the source and target videos into distinct parameters that can be combined to produce a retargeted video [12].

Training

Generative models like GANs have shown promising results in various domains, including images, videos, audio, and texts. However, the current state of video GANs is not as advanced as other domains due to the high computational power required to handle high-dimensional video data. Video GANs require a complex architecture that considers spatial and temporal data and involves collecting domain-specific videos that are time-consuming and expensive. As an illustration, DVD-GAN employs a single TPU, while TGANv2 utilizes 8 GPUs [12]. Nevertheless, video GANs have many applications, including speech animation, prediction, and retargeting. The paper aims to review and categorize different video GAN models based on their applications and highlight their differences.

3.3 Image-to-Image Translation

The image-to-image translation is a type of computer vision task that involves translating an input image into an output image that has some desired properties or characteristics. This can include converting a black-and-white image into a colored image or generating a realistic image from a low-resolution input.

Image-to-image translation using GANs has a broad range of applications across different fields. In medical imaging, GANs can generate synthetic images to enhance medical images and train machine-learning models for image segmentation, detection, and classification. GANs can design new fashion products, generate product images, and create virtual try-on experiences in fashion. Gaming can benefit from GANs by generating realistic images of game characters and environments. Architecture can use GANs to visualize 3D models of buildings and cities, while autonomous vehicles can be trained with GAN-generated realistic road scenes and objects. Additionally, GANs can create new art forms, such as realistic portraits and abstract images. Overall, GAN-based image-to-image translation has the potential to revolutionize many industries by providing new tools and capabilities for creating, designing, and analyzing images.

The process of training a GAN for image-to-image translation involves several steps. The first step is to collect and preprocess a large dataset of paired input-output images. Next, the generator and discriminator architectures are defined as convolutional neural networks with skip connections. The generator is trained to minimize the difference between the generated and real output images. In contrast, the discriminator is trained to distinguish between real and fake output images. Once the GAN is trained, it can be evaluated and used for image-to-image translation on new input images. The generated output images can be used for various applications such as image enhancement, colorization, or style transfer.

Data Collection and Preparation

Depending on the specific task and application, several datasets are available for image-to-image GAN training. Cityscapes is a popular dataset for urban street scenes [13], CelebA for celebrity faces [1], Maps for Google Maps satellite images [14], Horse2zebra for horse and zebra images [15], and Edges2shoes for shoes images and their corresponding edge maps [16]. These datasets are publicly available for research and development purposes, but ensuring compliance with applicable data licensing agreements and ethical considerations is essential.

Preparing data for image-to-image translation using GANs is similar to other image-generation tasks using GANs. Initially, the image data is transformed into floating pixels and then normalized to a specific range, either [0, 1] or [-1, 1], based on the task requirements. The Cityscapes dataset is employed in this case. It consists of source and target images merged with a size of 256x512, as depicted in Fig. 3.14. Thus, the primary step is to isolate the source from the target images or, in other words, the original photograph on the left half of the image, along with the semantic segmentation output on the right half.

Figure 3.15 displays the code snippets written to load the train image data, normalize the pixels and split the images. Two lines of code are written to split images, as presented in lines 14 and 15. Figure 3.15 displays the images after the splitting process. The exact process is applied to the test data.

Figure 3.16 presents the source and target image after preprocessing. Now the single image presented in Fig. 3.14 is represented by two: the ground truth image (photograph) to the left and the representation to the right.

Fig. 3.14 Sample image from Cityscapes dataset

```
# Define a list to store source images for training
image_data_source = []
# Define a list to target source images for training
image_data_target = []
# Enumerate over all image files in the dataset
for image_file in glob('/cityscapes_data/cityscapes_data/train/*.jpg'):
    # Open image file
    single_image=Image.open(image_file)
    # Transform image data into an array
    img = img_to_array(single_image)
    # Normalize pixel values to the range [0, 1]
    img = img / 255.0
    # Split the image into two tensors
    input_image = img[:, :256, :]
    target_image = img[:, 256:, :]
    # Save train images in lists
    image_data_target.append(target_image)
    image_data_source.append(input_image)
```

Fig. 3.15 Load image from Cityscapes dataset

Fig. 3.16 Image files after splitting

Model Design

Like any GAN model, the image-to-image translation model comprises a generator and a discriminator. The generator is responsible for generating the target images. In this case, the real photograph taken in the city street from the source semantic

segmentation image is input. The discriminator is responsible for differentiating between the original images in the Cityscapes dataset from the generated images.

The Generator Model

In the case of Cityscapes image-to-image translation, the generator takes the semantic segmentation image as an input to generate the original photograph of the streets. The generator follows an encoder-decoder architecture with skip connections. An encoder-decoder generator is a specific type of generator used in GANs that consists of two parts: an encoder and a decoder. The encoder takes the semantic segmentation image as an input and encodes it into a lower-dimensional latent space. On the other hand, the decoder takes the encoded image and decodes it into a high-dimensional image that closely resembles the original street image.

Figure 3.17 presents the generator architecture. It shows how the source input image is passed through different deep neural network layers to extract features and generate the target image. A 3D block presents the concatenate function because it takes two 3D tensors as input to produce a single 3D numerical tensor.

The implementation of the generator model is presented in Python as shown in Fig. 3.18. The encoder model uses a for loop from line 2 to line 23, where the input is passed through several convolutional layers with progressively increasing channels and down-sampling. Each convolutional layer is followed by batch normalization and a *LeakyReLU* activation function. The output of each convolutional layer is stored in a list called *encoder_tensor*. After the final convolutional layer, the output is passed through another layer with 512 channels.

The decoder model is defined using a for loop from lines 24 to 39 in Fig. 3.18. The input is passed through several transposed convolutional layers with

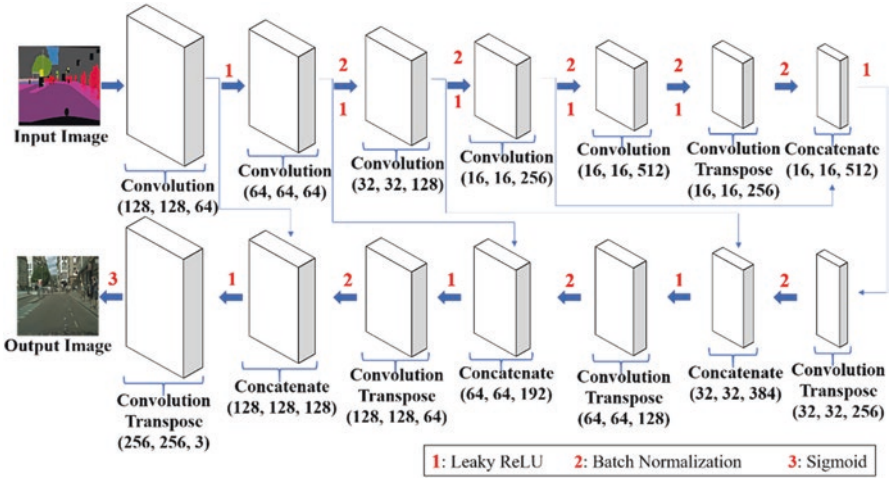

Fig. 3.17 Generator model architecture

```
def build_generator():
    # Define a list to store encoder output features
    encoder_tensor = []
    # Define the input shape
    inputs = Input(shape=(width, height, 3))
    # Define the first layer in encoder model
    x = Conv2D(64, (3, 3), strides=(2, 2), padding='same')(inputs)
    x = LeakyReLU(alpha=0.2)(x)
    # Store the features out from convolution layer
    encoder_tensor.append(x)
    # Define the rest of the layers in encoder model using for loop
    for i in range(3):
        # Define the number of channels for the current layer
        channels = 2 ** (i + 6)
        x = Conv2D(channels, (3, 3), strides=(2, 2), padding='same')(x)
        x = BatchNormalization()(x)
        x = LeakyReLU(alpha=0.2)(x)
        # Append the output tensor to the encoder tensor
        encoder_tensor.append(x)
    # Define the last layer in encoder model
    x = Conv2D(512, (3, 3), strides=(1, 1), padding='same')(x)
    x = BatchNormalization()(x)
    x = LeakyReLU(alpha=0.2)(x)
    # Define the first layer in decoder model
    x = Conv2DTranspose(256, (3, 3), strides=(1, 1), padding='same')(x)
    x = BatchNormalization()(x)
    # Concatenate the output of the current decoder layer with the output of the last encoder layer
    x = Concatenate()([x, encoder_tensor[len(encoder_tensor)-1]])
    x = LeakyReLU(alpha=0.2)(x)
    # Define other layers in decoder model using for loop
    for i in range(3):
        # Define the number of channels for each layer
        channels = 512 // 2 ** (i + 1)
        x = Conv2DTranspose(channels, (3, 3), strides=(2, 2), padding='same')(x)
        x = BatchNormalization()(x)
        x = Concatenate()([x, encoder_tensor[len(encoder_tensor)-2-i]])
        x = LeakyReLU(alpha=0.2)(x)
    # Define the last layer in decoder model
    x = Conv2DTranspose(3, (3, 3), strides=(2, 2), padding='same')(x)
    # Add activation layer
    outputs = Activation('sigmoid')(x)
    # Define the full encoder-decoder model
    encoder_decoder = Model(inputs, outputs)
    return encoder_decoder
```

Fig. 3.18 Generator Model

progressively decreasing channels and up-sampling. Batch normalization and a *LeakyReLU* activation function follow each transposed convolutional layer. Then the output is concatenated with the corresponding layer from the encoder. The output from the decoder model's final convolutional layer is passed through a transposed convolutional layer with three channels. A *Sigmoid* activation function is applied to generate the final output image. The full encoder-decoder model is defined in line 28, and it takes the input image and generates the output image with the exact dimensions.

The Discriminator Model

The discriminator in image-to-image translation aims to differentiate between real and fake generated images from the generator. Figure 3.19 illustrates the discriminator model architecture, followed by the code snippets in Fig. 3.20. It takes two images as input, a source image and a target image, and predicts whether the image is real or fake. The model consists of several layers. First, a concatenate function merges source and target images into a single tensor (Line 6). Then this tensor is down-sampled using multiple convolutional layers. This is done to extract features from the image that can be used to classify whether it is real or fake. Specifically, the discriminator applies four convolutional layers with increasing filters. Each layer has a kernel size of 4 and a stride of 2 to reduce the spatial dimensions of the tensor while increasing the number of filters to capture more complex features. This is shown in lines 1 to 3 in Fig. 3.20. Then the output from the last convolutional layer is flattened and passed through a fully connected layer (*Dense*) with a single output neuron and a *Sigmoid* activation function. This final layer outputs a probability score between 0 and 1, representing the probability that the target image is real. Finally, the discriminator model is compiled using the *binary cross-entropy* loss function and the Adam optimizer with a learning rate of 0.0002 and a beta_1 of 0.5. The discriminator loss is weighted equally with the generator loss in the combined model by setting the *loss_weights* to [0.5].

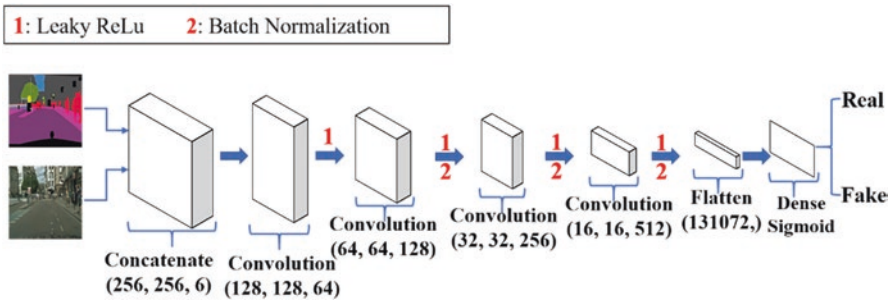

Fig. 3.19 Discriminator architecture

```
def build_discriminator():
    # Define the inputs to the discriminator model
    input_source_image = Input(shape=(width, height, 3))
    input_target_image = Input(shape=(width, height, 3))
    # Concatenate the source and target images
    merged = Concatenate()([input_source_image, input_target_image])
    # Downsample the concatenated images
    x = Conv2D(64, kernel_size=4, strides=2, padding='same')(merged)
    x = LeakyReLU(alpha=0.2)(x)
    x = Conv2D(128, kernel_size=4, strides=2, padding='same')(x)
    x = BatchNormalization()(x)
    x = LeakyReLU(alpha=0.2)(x)
    x = Conv2D(256, kernel_size=4, strides=2, padding='same')(x)
    x = BatchNormalization()(x)
    x = LeakyReLU(alpha=0.2)(x)
    x = Conv2D(512, kernel_size=4, strides=2, padding='same')(x)
    x = BatchNormalization()(x)
    x = LeakyReLU(alpha=0.2)(x)
    # Flatten and add a dense layer for classification
    x = Flatten()(x)
    output = Dense(1, activation='sigmoid')(x)
    # Create the model with the inputs and output layers
    disc_model = Model([input_source_image, input_target_image], outputs= output)
    # Compile the model with binary cross-entropy loss and Adam optimizer
    discriminator_optimizer = Adam(lr=0.0002, beta_1=0.5)
    disc_model.compile(loss='binary_crossentropy', optimizer=discriminator_optimizer, loss_weights=[0.5])
    # Return the discriminator model
    return disc_model
```

Fig. 3.20 Discriminator model

The Adversarial Network

The generator and discriminator models must be combined to train the generative adversarial network (GAN). The discriminator model is set to be non-trainable, as it is used solely to evaluate the generated images produced by the generator model. The generator loss is then computed based on the discriminator's evaluation, and its weights are updated through backpropagation. Similarly, as in the discriminator, the adversarial network is compiled. However, this network is compiled using two types of losses "*binary cross-entropy*" and "*mean square error.*" Adam optimizer with a learning rate of 0.0002 and a beta_1 of 0.5 is used. Figure 3.21 shows the adversarial network.

Training

The training process in image-to-image translation closely resembles that of traditional GAN models. During training, the generator and discriminator weights are updated through backpropagation. When the generator updates its weights, the discriminator is set to non-trainable.

```
def build_gan(generator, discriminator):
    discriminator.trainable = False
    input_source = Input(shape=(width, height, 3))
    target_image = generator(input_source)
    validity =  discriminator([input_source, target_image])
    gan = Model(inputs=input_source, outputs=[validity, target_image])
    gan_optimizer = Adam(learning_rate=0.0002, beta_1=0.5)
    gan.compile(loss = ["binary_crossentropy", "mae"], optimizer = gan_optimizer, loss_weights = [1,100])
    return gan
```

Fig. 3.21 Adversarial network

Fig. 3.22 Image generated by the GAN model at epoch 0

Fig. 3.23 Image generated by the GAN model at epoch 70

The training process was carried out over 10,000 epochs, with a batch size of 128. The steps per epoch are equal to the length of the data divided by the batch size. The generator and discriminator losses were computed during each step, and their weights were updated. To monitor the progress of the generator output, a plot that shows its output every ten epochs is generated. The generated images were visually inspected to evaluate the model's progress. The images generated by the model show an increase in resolution with each epoch, as shown in Figs. 3.22, 3.23, 3.24, 3.25, and 3.26, indicating that the generator is learning to produce more accurate and realistic images.

Fig. 3.24 Image generated by the GAN model at epoch 170

Fig. 3.25 Image generated by the GAN model at epoch 5000

Fig. 3.26 Image generated by the GAN model at epoch 10000

3.4 Text to Image

Text-to-image translation using GANs has gained much attention in recent years. This technology is used to generate images based on textual descriptions. It has numerous applications in various domains, such as computer graphics, gaming, and advertising. Text-to-image translation has been used in several real-world

applications. One example is generating product images for e-commerce websites, where the textual description of products can be transformed into a visual representation of the product. Another use case is in generating graphical illustrations for educational or technical documents. It can also be used to generate synthetic data for training and improving the accuracy of computer vision algorithms.

The importance of text-to-image translation lies in its ability to convert textual descriptions into visually appealing images. This helps to overcome the limitations of traditional text-based representation and allows for a more immersive and interactive experience.

Text-to-image translation using GANs requires computational resources and complex models to perform well. Generating high-quality images from textual description requires the model to understand the relationships between the textual description and the corresponding visual representation. The complexity of the model increases with the amount of detail and diversity in the images being generated. Additionally, the large amount of data needed for training, along with the computationally intensive nature of GANs, requires significant computational resources and can be quite demanding regarding hardware requirements. It is often necessary to use powerful GPUs and distribute the computation across multiple machines to achieve good results.

This section provides a complete elucidation of the procedure to implement text-to-image GAN. The process starts by importing required modules till reaching deployment in addition to a basic code illustration. The code implementation requires minimal computational resources and is considered the most straightforward approach to initiate text-to-image GAN.

Module Requirements

The process of text-to-image translation begins with importing the necessary modules. This application of GAN has two different types of data: text and images. Thus, it is necessary to import multiple modules, and each is used for a specific purpose. To illustrate, implementing GAN deep learning models requires deep learning frameworks such as TensorFlow, PyTorch, or Keras. In addition, deep learning models understand input data in a specific numerical format.

For this reason, additional modules that can preprocess image and text data are required. Figure 3.27 shows a sample code that includes importing the preprocessing libraries that can be used in text-to-image GANs. This code imports various required libraries for text preprocessing using the Keras API of the Tensorflow Library. The code also imports libraries such as Numpy, Pandas, and Matplotlib and various packages from the Natural Language Toolkit (*nltk*) for cleaning text data. The code also downloads the required packages for text processing, such as the Punkt tokenizer, WordNet Lemmatizer, and the averaged perceptron tagger.

```
# Import Tokenizer module to transform text data to numerical format
from tensorflow.keras.preprocessing.text import Tokenizer
# Import pad_sequences to make all numerical input to same size
from tensorflow.keras.preprocessing.sequence import pad_sequences
# Import the Natural Language Toolkit (nltk) for text processing
import nltk
# Download the required datasets from nltk
nltk.download('wordnet')
nltk.download('stopwords')
# Import stopwords from nltk
from nltk.corpus import stopwords
# Import lemmatizer from nltk
from nltk.stem.wordnet import WordNetLemmatizer
# Import string module for processing text
import string
# Import image preprocessing functions from Keras
from tensorflow.keras.preprocessing.image import img_to_array, load_img
# Import Image from PIL for image processing
from PIL import Image
# Import matplotlib for plotting
import matplotlib.pyplot as plt
# Import glob for matching file patterns
from glob import glob
# Import Path for file system operations
from pathlib import Path
```

Fig. 3.27 Importing required modules for data preparation

Dataset

Once the modules are imported, the next step is typically to load the dataset that will be used for training and evaluating the model. This dataset can be created from an extensive collection of images paired with textual descriptions. It will train the generative model to generate new images from textual descriptions. Several datasets are commonly used for text-to-image translation using GANs. These datasets can also be used for implementing an image captioning deep learning model. COCO Captions is an example of these datasets, which includes the Common Objects in Context (COCO) dataset and their corresponding captions [17]. Flickr30k is another dataset which includes images and their corresponding images and captions written by human annotators [18]. Flickr8k is similar to Flikr30k but includes 8000 images instead of 30000 [19]. Pascal Sentences dataset consists of images from the Pascal Visual Object Classes Challenge and captions describing the objects within the

images [20]. Oxford-102 Flowers contains images of 102 different flower species and captions describing the flowers' species and attributes [21].

The datasets mentioned above require high computational resources. However, the aim is to understand the implementation of text-to-image GAN. Other datasets comprising images with textual descriptions, such as MNIST Fashion and CIFAR-10, can be used to implement a simpler model. These datasets will simplify the implementation so that the model input text will be a single word, such as "coat" in MNIST Fashion and "airplane" in CIFR-10. The model will generate an image that depicts the word. The following sections will present text-to-image implementation using the MNIST Fashion dataset.

It is crucial to preprocess the data and to prepare it for use in the model, as this will significantly impact the performance and accuracy of the model. Next, an explanation of preprocessing and preparing the text data to form the GAN model will be presented. This procedure can be applied to any textual data preceding text implementation to image GAN. Additionally, loading and preprocessing image files will be presented.

Data Preprocessing

Preparing text and image data is the initial step in implementing any deep learning model, including text-to-image GANs. This process involves preprocessing the text data and transforming it into numerical representations. In this regard, a comprehensive description of preparing the data for text-to-image GANs, explicitly focusing on MNIST Fashion data, will be presented in this section.

To begin with, the text data found in datasets comprises a combination of different words that describe the images, which are primarily written by humans. In English, a sentence typically starts with a capital letter and ends with a period, while it may also include other punctuation, such as commas. Additionally, English sentences may have multiple words with similar meanings but different types, necessitating lemmatization. Lemmatization reduces words to their base or dictionary form to simplify the analysis of text data. For example, the words "coming," "came" and "comes" would be reduced to the lemma "come."

Once the text data has been cleaned, the next step is to transform it into numerical representations. Tokenization is applied to break down the text into individual tokens or words. It allows text to be presented numerically and processed by machine learning models. However, representing words as numbers and sentences as a sequence of numbers is not enough. In text-to-image GANs, each image is represented by a specific sentence, but the representations are of different lengths. Padding is used to ensure that all input sequences have the same length, as most machine-learning models require fixed-size inputs. Padding with zeros is typically used and can be placed at the beginning or end of the sequence.

Figure 3.28 demonstrates text preprocessing code to prepare text data for text-to-image GAN. The first function, which starts at line 2, is used for text cleaning where the punctuations are removed in line 4, all words are transformed to lower letters in

```
1  stop_words = set(stopwords.words("english"))
2  def preprocess_text(text):
3      # Remove punctuation from the text
4      text = text.translate(str.maketrans("", "", string.punctuation))
5      # Convert to lowercase
6      text = text.lower()
7      # Tokenize the text
8      tokens = word_tokenize(text)
9      # Remove stop words
10     tokens = [token for token in tokens if token not in stop_words]
11     # Initialize the lemmatizer
12     lemmatizer = WordNetLemmatizer()
13     # Lemmatize and stem each token
14     lemmatized_tokens = [lemmatizer.lemmatize(token) for token in tokens]
15     return 'startseq ' + " ".join(lemmatized_tokens)+ ' endseq'
16 def text_preprocessing(data):
17     # Count the number of unique words
18     tokenizer = Tokenizer(num_words=None, lower=True, split=' ')
19     tokenizer.fit_on_texts(data)
20     num_words = len(tokenizer.word_index) + 1
21     # Find the length of the longest sentence
22     max_length = max([len(s.split()) for s in data])
23     # Tokenize the sentences and pad the sequences
24     tokenized_data = tokenizer.texts_to_sequences(data)
25     padded_data = pad_sequences(tokenized_data, maxlen=max_length, padding='post')
26     return padded_data, num_words, max_length, tokenizer
```

Fig. 3.28 Data cleaning and preprocessing

line 6, and the lemmatization is done between lines 12 and 14. The second function starts at line 16 for tokenization and padding. Tokenization is represented between lines 18 and 19 and padding between lines 24 and 25.

The second step is to load and preprocess the image data. Image data must be converted into a numerical array, including the image's pixel values. The image array values must be normalized between [0, 1] or [-1,1]. Figure 3.15, in the previous section, discussed how to load and process image files. The same procedure will be followed, and all target images must be saved in a list.

Figure 3.29 illustrates loading text images and mapping them to their corresponding text description. The images are loaded and normalized between lines 6 and 18. Then each image is mapped to its corresponding id using a dictionary, as illustrated in line 20. In addition to loading image data, the corresponding text labels must be mapped to each image. This mapping is done by loading the labels and

```
# Load images
all_image_data = []
image_id_all = []
width, height = 224, 224
# Enumerate over all image files
for image_file in glob('PATH/*.jpg'):
    # Save all image file names in a list named "image id"
    image_id_all.append(Path(image_file).name.split(".")[0])
    # Open image file and save it in a variable
    single_image = Image.open(image_file)
    # Reshape image file to 224*224
    single_image = single_image.resize((width, height))
    # Transform image to a numerical array of pixels
    img = img_to_array(single_image)
    # Normalize images between 0 and 1
    img = img / 255
    # Save all image features in a list
    all_image_data.append(img)
# Save image features and their corresponding names (id's) in a dictionary
mapped = dict(zip(image_id_all, all_image_data))
# Load corresponding text labels file
text = pd.read_csv("captions.txt", sep=",")
# Split the image column to two columns to have the image name in a column as id
text[['image_id', 'extension']] = text['image'].apply(lambda x: pd.Series(str(x).split(".")))
text = text.drop(['image', 'extension'], axis = 1)
# Map each label with its corresponding image
real_images = []
for id in text['image_id']:
    real_images.append(mapped[id])
```

Fig. 3.29 Image loading and mapping to labels

arranging the image list in the same order as the image labels. Loading images and mapping them to descriptions is shown between lines 22 and 29.

As stated before, to start with a simple text-to-image GAN model, the MNIST Fashion dataset is used. MNIST Fashion dataset can be loaded with Keras. Figure 3.30 presents a code fragment on loading and preprocessing the dataset. It is divided into train and test images, each with its label. Each train and test image is represented by an array of pixel values between 0 and 255. This image data must be normalized between [0,1], as shown in lines 7 and 8. The label in the dataset belongs to 10 classes a T-shirt/top, trouser/pant, pullover shirt, dress, coat, sandal, shirt, sneaker, bag, and ankle boot, as shown in line 10. Each one of the labels is

```
1  # Load the dataset
2  (X_train, y_train), (X_test, y_test) = fashion_mnist.load_data()
3  # Reshape image
4  X_train = X_train.reshape(X_train.shape[0], 28, 28, 1).astype('float32')
5  X_test = X_test.reshape(X_test.shape[0], 28, 28, 1).astype('float32')
6  # Normalize image data between [0,1]
7  X_train = X_train / 255.0
8  X_test = X_test / 255.0
9  # Define class names
10 class_names = ['T-shirt/top', 'trouser', 'pullover', 'dress', 'coat',
                 'sandal', 'shirt', 'sneaker', 'bag', 'Ankle boot']
11 # Change the labels into a single column and multiple rows format
12 y_train1 = y_train.reshape(-1, 1)
13 y_test1 = y_test.reshape(-1, 1)
```

Fig. 3.30 Loading and preprocessing MNIST Fashion dataset

represented by a numerical value between [0,9] respectively. To map images with corresponding labels, "y_train" and "y_test" are transformed into a single-column list with multiple rows, as shown in lines 12 and 13. This dataset will simplify the loading and preprocessing of text and image data.

Model Design

The text-to-image GAN is similar to any GAN model composed of a generator and discriminator. The generator is composed of multiple layers. It is responsible for generating images that describe the labels. The discriminator is also made of multiple layers with binary output. It aims to differentiate between generated images and real samples from MNIST Fashion.

Generator Model

In text-to-image GANs, the generator model will take two inputs: noise and label. Before combining the two inputs, the label data represented numerically must pass through an *Embedding* layer. The generator architecture is presented in Fig. 3.31. A 2D block represents the embedding and LSTM layers in the architecture. This representation is chosen based on the output tensor from each layer. The code implementation for the generator model is presented in Fig. 3.32. This *Embedding* layer will map the input values into a high-dimensional fixed-length space vector. Each dimension in the dense vector represents a feature or attribute of the input. An

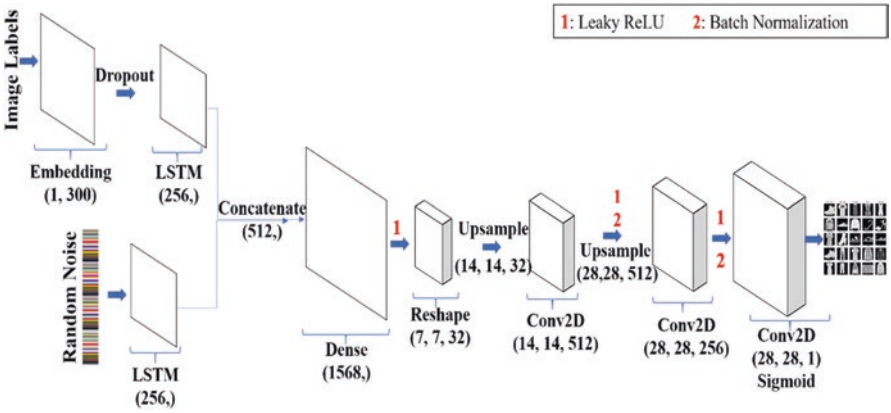

Fig. 3.31 Test-to-image generator architecture

Embedding layer is often used to capture the relationships between input data, the labels in this case, in a way that allows the model to learn the underlying patterns and structures.

A *Dropout* layer in line 8 follows the *Embedding* layer that is in line 6. This layer is used as a regularization technique to prevent overfitting. It randomly drops out or turns off a fraction of neurons from the *Embedding* layer. The *Dropout* layer prevents the model from relying too heavily on specific *Embedding* dimensions. The *Dropout* layer is followed by a long short-term memory (*LSTM*) layer (refer to line 10). This *LSTM* layer is added to capture the temporal structure of the data out from the *Embedding* layer. *LSTM* is usually used because in text-to-image GAN the input labels might be a sequence of words or tokens that describe the image. So, the sequence might be quite complex, and it is difficult for the model to capture the relevant information from the text using only a superficial *Embedding* layer. In conclusion, an *Embedding* layer followed by an *LSTM* layer is a common technique for preprocessing text data in deep learning models.

The noise input is chosen to have a size of 100, as in previous applications. However, the two inputs must have the exact dimensions to concatenate them. For this reason, the input noise is fed to a *Dense* layer (Line 14) with a number of neurons equal to that of the *Embedding* layer. Then the data is passed through multiple up-sampling layers to reach the desired output image size (28,28,1). The up-sampling layers can be built using convolutional transpose or the *Upsampling2D* in Keras followed by a convolutional layer with a stride of 1. To show how these two up-sampling methods work, text-to-image GANs to build the generator upsampling2D, followed by a convolutional layer, are utilized. The size of two in the up-sampling layer means double the previous layer's output. The activation function used is *Sigmoid* to output data between [0,1] as the normalized images. If the images are normalized between [-1, 1], *Tanh* activation function will be used.

```
# Define the generator model
def build_generator():
    # Define the shape if input 1 (label)
    input_embed = Input(shape=(1,))
    # Map the input data into high dimensional space vector
    se1 = Embedding(num_classes, 300, mask_zero=True)(input_embed)
    # Prevent overfitting
    se2 = Dropout(0.5)(se1)
    # Capture long-term dependencies and maintain memory of embedding layer output
    se3 = LSTM(256)(se2)
    # Define the shape of input 2 (noise)
    input_noise = Input(shape=(100,))
    # Pass the noise through a dense layer to make two inputs with same dimensions
    fe2 = Dense(256, activation='relu')(input_noise)
    # Concatenate both inputs
    gen_input = Concatenate(axis=1)([fe2, se3])
    # Feed the gen_input to a dense layer
    x = Dense(32 * 7 * 7, use_bias=False)(gen_input)
    x = LeakyReLU()(x)
    # Reshape output of dense layer to (7, 7, 32)
    x = Reshape((7, 7, 32), input_shape=(32 * 7 * 7,))(x)
    # Upsample to 14x14
    x = UpSampling2D(size=(2, 2))(x)
    x = Conv2D(512, kernel_size=3, padding="same", strides=1, use_bias=False)(x)
    x = BatchNormalization()(x)
    x = LeakyReLU()(x)
    # Upsample to 28x28
    x = UpSampling2D(size=(2, 2))(x)
    x = Conv2D(256, kernel_size=3, padding="same", strides=1, use_bias=False)(x)
    x = BatchNormalization()(x)
    x = LeakyReLU()(x)
    # Output layer with shape (28, 28, 1)
    x = Conv2D(1, kernel_size=3, padding="same", strides=1, use_bias=False)(x)
    x = Activation(activation='sigmoid')(x)
    # Define the model inputs and outputs
    model = Model(inputs=[input_embed, input_noise], outputs=x)
    return model
```

Fig. 3.32 Test-to-image generator code implementation

Discriminator Model

The discriminator model is composed of several down-sampling layers. In the context of text-to-image GANs, the input to the discriminator will be the image output from the generator with the corresponding label. Thus, the model will be fed by the same label fed to the generator to generate an image x with the image x. The complete architecture and implementation of the discriminator are presented in Figs. 3.33 and 3.34, respectively. Similarly, as in the generator, the input label is fed to an *Embedding* layer (Line 7) followed by a *Dropout* layer (Line 9) and then *LSTM* (Line 10). However, the two inputs have different dimensions. The image has three dimensions, whereas the labels have only one.

For this reason, the output from the *LSTM* layer is reshaped to have three dimensions. This reshaped layer is followed by two up-sampling layers (lines 12 to 18) using convolution transpose to reach the image size (28, 28, 1). Now both inputs, the image and label, have the exact dimensions. So, they are concatenated (Line 20).

After adding the two inputs, several down-sampling layers using convolution with a stride of 2 are used (lines 22 to 35). To reach a binary output *Flatten* layer is used to flatten the output from the convolutional layer to have only one dimension. The output from *Flatten* layer is passed through a final dense layer with a single output and *Sigmoid* activation function. The discriminator model is then compiled using *binary cross-entropy* loss and Adam optimizer with a learning rate of 0.002.

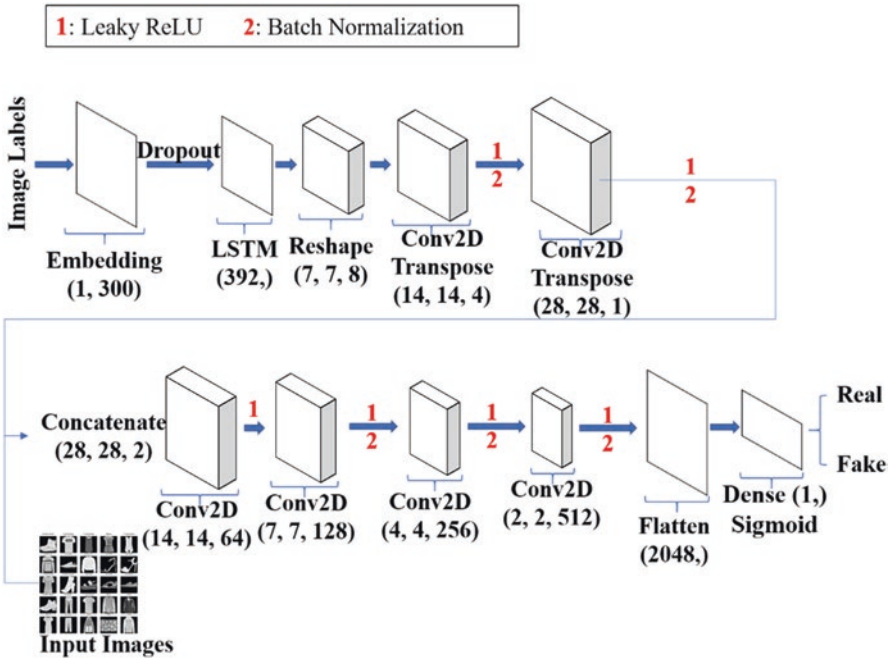

Fig. 3.33 Test-to-image discriminator architecture

```
def build_discriminator():
    # Define the shape of input 1 (image)
    input_target_image = Input(shape=(28, 28, 1))
    # Define the shape of input 2 (label)
    input_label = input_embed = Input(shape=(1,))
    # Map the input data into high dimensional space vector
    se1 = Embedding(num_classes, 300, mask_zero=True)(input_embed)
    # Prevent overfitting
    se2 = Dropout(0.5)(se1)
    # Capture long-term dependencies and maintain memory of embedding layer output
    se3 = LSTM(7*7*8)(se2)
    # Reshape the output of LSTM layer to (7,7,8)
    se4 = Reshape((7, 7, 8))(se3)
    # Upsample to 14x14
    se5 = Conv2DTranspose(4, (1, 1), strides=(2, 2), padding='same')(se4)
    se6 = BatchNormalization()(se5)
    se7 = LeakyReLU(alpha=0.01)(se6)
    # Upsample to 28*28
    se8 = Conv2DTranspose(1, (2, 2), strides = (2, 2), padding = 'same', activation = "sigmoid")(se7)
    se9 = BatchNormalization()(se8)
    se10 = LeakyReLU(alpha=0.01)(se9)
    # Concatenate both inputs
    merged = Concatenate()([input_target_image, se10])
    # Downsample to 14*14
    x = Conv2D(64, kernel_size=4, strides=2, padding='same')(merged)
    x = LeakyReLU(alpha=0.2)(x)
    # Downsample to 7*7
    x = Conv2D(128, kernel_size=4, strides=2, padding='same')(x)
    x = BatchNormalization()(x)
    x = LeakyReLU(alpha=0.2)(x)
    # Downsample to 4*4
    x = Conv2D(256, kernel_size=4, strides=2, padding='same')(x)
    x = BatchNormalization()(x)
    x = LeakyReLU(alpha=0.2)(x)
    # Downsample to 2*2
    x = Conv2D(512, kernel_size=4, strides=2, padding='same')(x)
    x = BatchNormalization()(x)
    x = LeakyReLU(alpha=0.2)(x)
    # Change Data to 1D
    x = Flatten()(x)
    # Output layer
    output = Dense(1, activation='sigmoid')(x)
    # Create the model
    disc_model = Model([input_target_image, input_label], outputs=output)
    # Compile model
    discriminator_optimizer = Adam(lr=0.0002, beta_1=0.5)
    disc_model.compile(loss='binary_crossentropy', optimizer=discriminator_optimizer)
    return disc_model
```

Fig. 3.34 Test-to-image discriminator code implementation

Adversarial Model

The adversarial model, GAN, is built to assess the generator and update its weight. Figure 3.35 illustrates how the GAN model is built. It shows that the discriminator and the generator are placed on top of each other. The discriminator's weights are frozen in the GAN model, as presented in Line 8. In this case, the discriminator is non-trainable and will act by only giving feedback to the generator to update its weights. The GAN model takes two inputs, image label and noise, and outputs the generated image and predicted labels from the discriminator.

Training Stage

The training stage in text-to-image GAN is similar to any training stage in the previous GAN models. The weights of the generator and discriminator are updated using backpropagation. It is illustrated in the previous chapter, Figure 2.14, how to custom the training loop. The training loop is repeated for n epochs, where $n = 100$.

At each epoch, a random batch of size = 128 from training data is chosen to train the model. Each time the discriminator and generator are trained, their losses are computed using the *train_on_batch* method. In text-to-image GAN, the discriminator loss comprises real, fake, and wrong loss. The real loss is the discriminator loss

```
1   def build_adversarial(generator_model, discriminator_model):
2       # Define the shape of input 1 (label)
3       input_layer1 = Input(shape=(1,))
4       # Define the shape of input 2 (noise)
5       input_layer2 = Input(shape=(100,))
6       # Define generator output
7       x = generator_model([input_layer1, input_layer2])
8       discriminator_model.trainable = False
9       # Define discriminator output
10      probabilities = discriminator_model([x, input_layer1])
11      # Build GAN model
12      adversarial_model = Model(inputs = [input_layer1, input_layer2], outputs =
    [probabilities, x])
13      # Compile GAN model
14      gan_optimizer = Adam(learning_rate=0.0002, beta_1=0.5)
15      adversarial_model.compile(loss = ["binary_crossentropy", "mae"], optimizer =
    gan_optimizer, loss_weights=[1, 100])
16      return adversarial_model
```

Fig. 3.35 Test-to-image GAN model code implementation

on actual image data from the training dataset. The fake loss is the discriminator loss on generated images. The wrong loss is an additional loss computed to guarantee that the output image from the generator describes the input label. The wrong loss refers to the state when the discriminator model is tested using the generated images from the generator with wrong random labels. Figure 3.36 illustrates the discriminator losses computed during the training process.

However, the generator loss is computed by comparing the real and the generated image. Figure 3.37 illustrates the generator loss computed during the training process. The generator loss is computed using the GAN model. It takes the image labels and noise as input to generate images and then computes the loss between real and generated images.

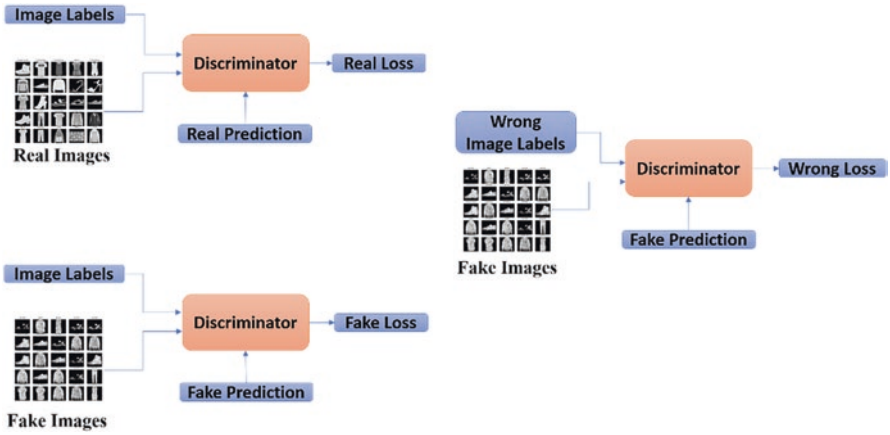

Fig. 3.36 Discriminator loss

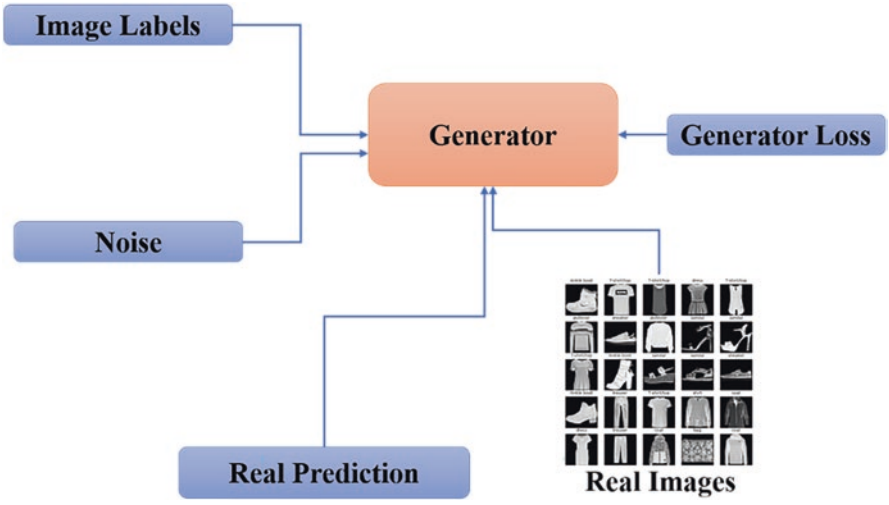

Fig. 3.37 Generator loss

The model effectiveness is tested by feeding the generator with a random sample from test data labels. The output images are plotted to assess the model performance visually instead of only depending on loss values. Figures. 3.38, 3.39, 3.40, and 3.41 show how the model performance is enhanced with increasing epochs.

Evaluation and Refinement

Model evaluation and refinement is the last step in any GAN model. Evaluation means assessing the model's ability to generate realistic images that match the labels. Refinement means performing hyperparameter tuning and optimization if needed. If the model can generate realistic data that cannot be distinguished from real data, then it does not need optimization. The model efficiency and robustness

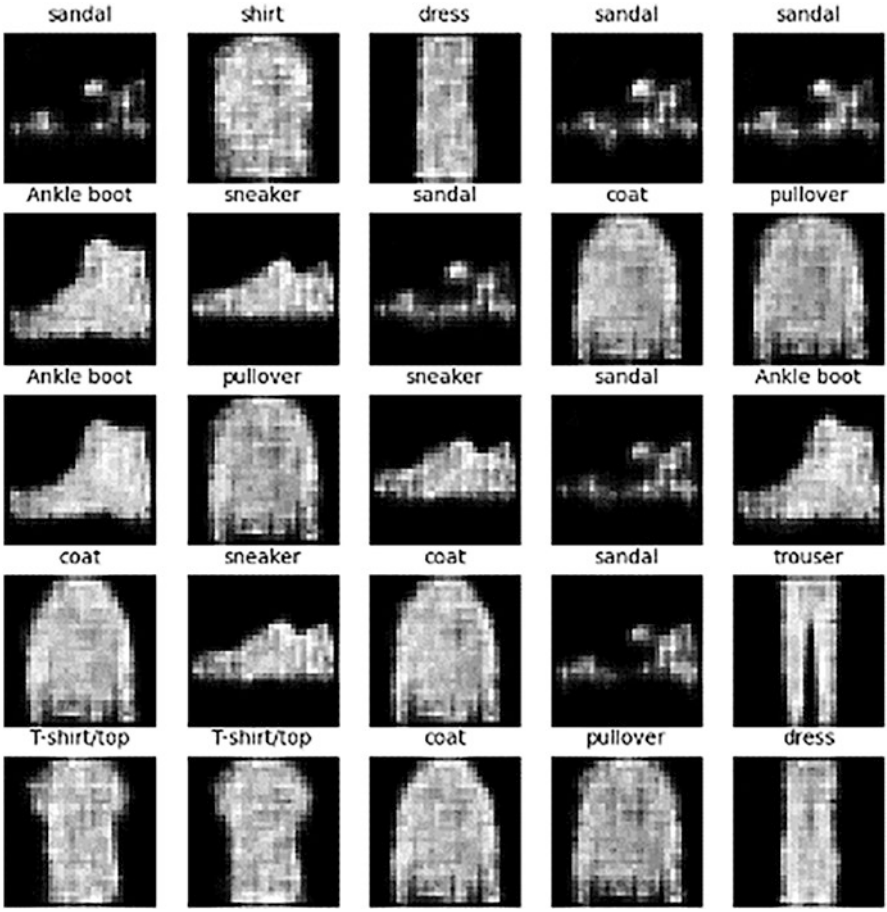

Fig. 3.38 Generated images at epoch 0

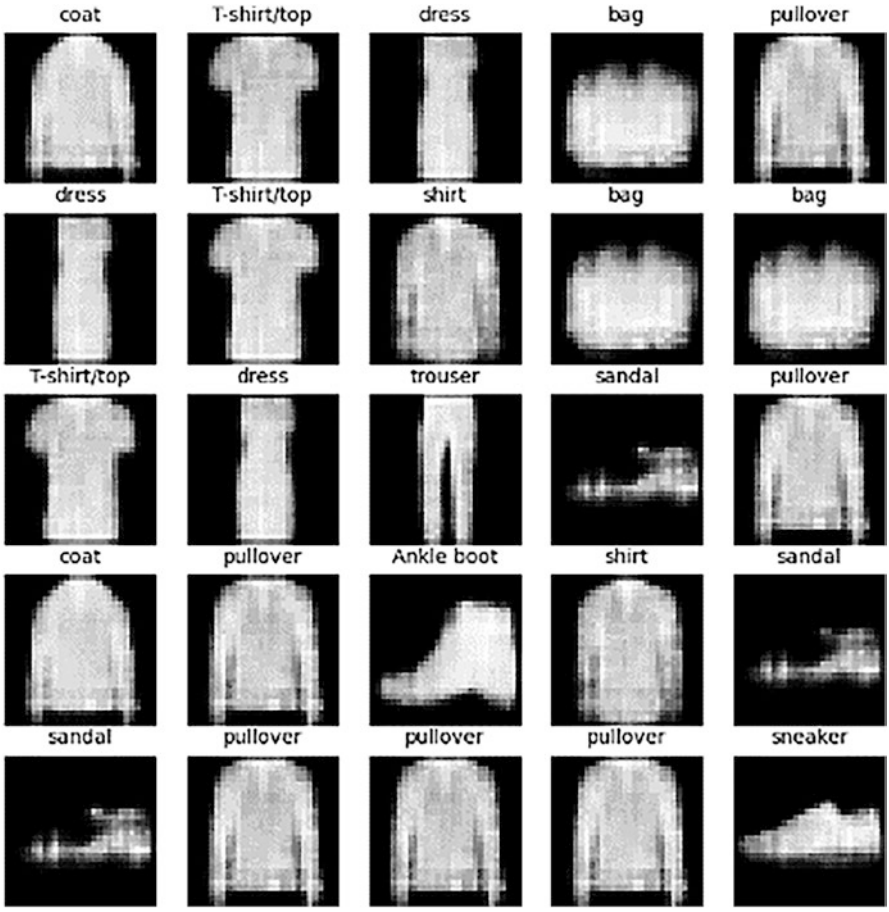

Fig. 3.39 Generated images at epoch 10

can be assessed from the values of losses in addition to the printed images. A more complex model is needed when the model cannot generate realistic images. Model tuning can be done by adding layers to the models, changing the hyperparameter values of the models, changing the optimizer type, or changing the learning rate.

3.5 CycleGAN

CycleGAN, also known as Cycle-Consistent GAN, is a type of generative adversarial network (GAN) introduced in 2017 by Zhu et al. [22]. It was designed to overcome the limitations of traditional image-to-image translation by achieving success in this task without the need for paired training data. Like other GAN models, CycleGAN consists of a generator and discriminator.

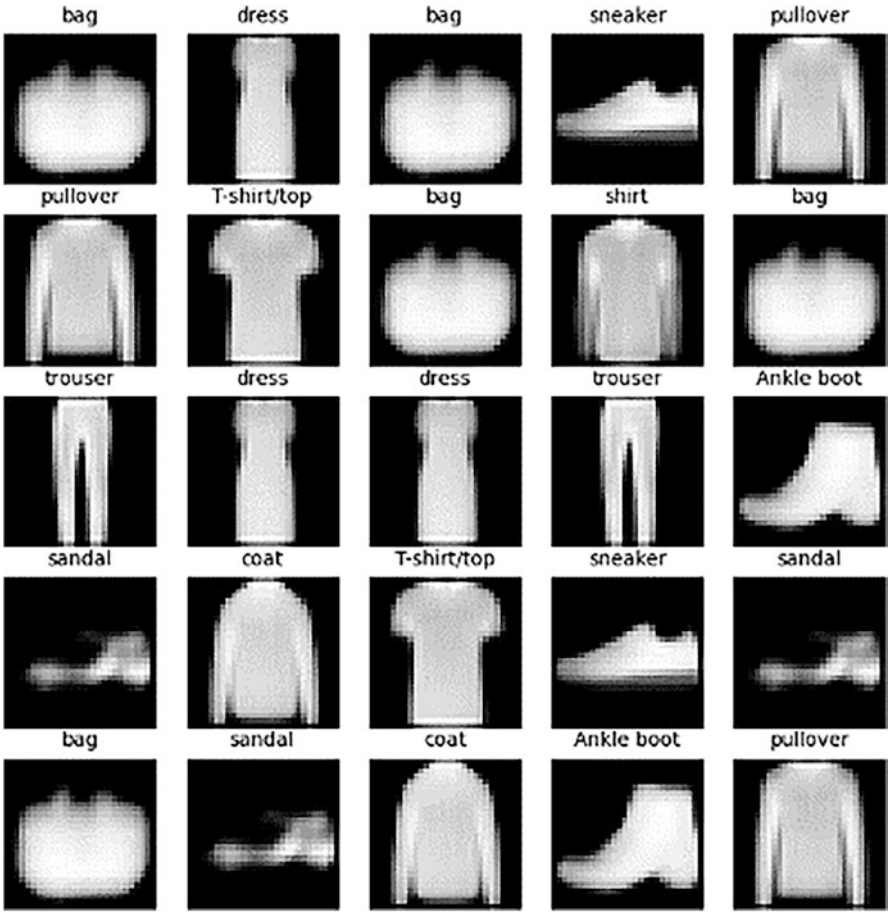

Fig. 3.40 Generated images at epoch 50

One of the main applications of CycleGAN is style transfer, which allows for transforming images into different artistic styles. It can also be used for domain adaptation, where the source image is translated into another domain. For example, it can translate a dog image into a cat or a spring image into autumn. CycleGAN has also demonstrated success in voice conversion, as shown by Kaneko et al. [23], and can be used for enhancing image resolution, removing noise from images, and image segmentation.

CycleGAN has several advantages over traditional GANs, including creating high-quality images with minimal training data and handling complex image translations without manual intervention. However, its unsupervised nature can lead to unpredictable results and is limited to specific tasks such as image-to-image translation and voice conversion. Moreover, it can only perform one-to-one translation and cannot generate multiple images from a single input. Recently GANs have been used for medical purposes, such as COVID-19 detection by Bargshady et al. [24].

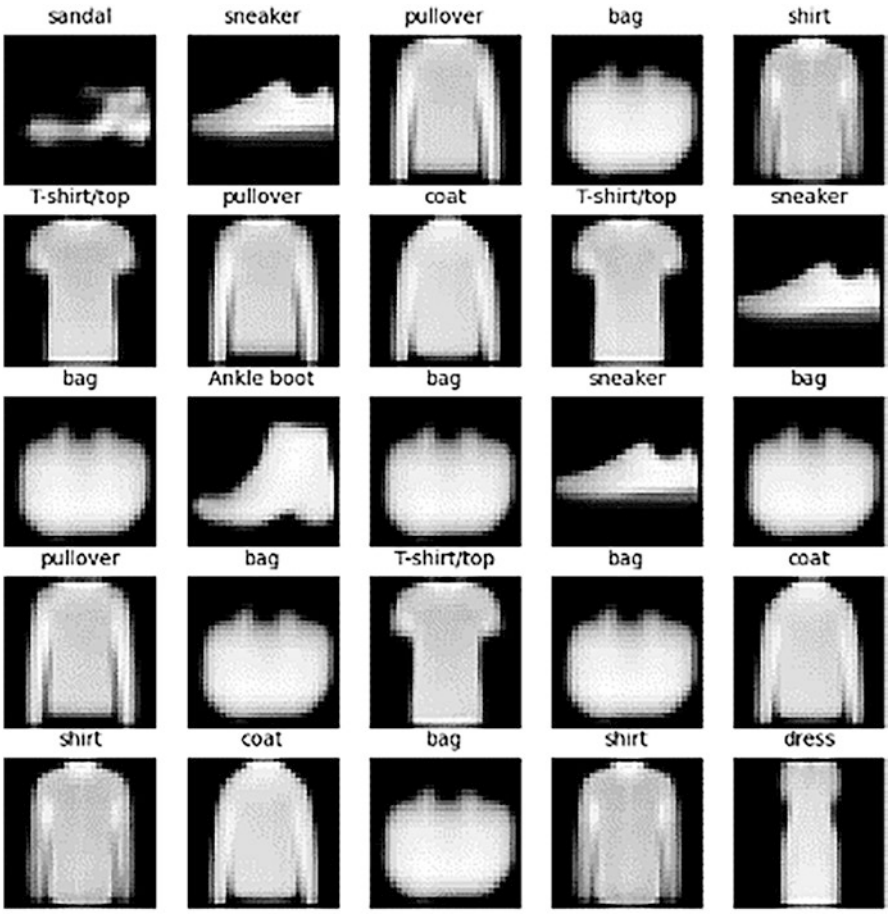

Fig. 3.41 Generated images at epoch 100

CycleGAN translates images in two domains, X and Y, and learns mapping in both directions, i.e., G: X to Y and F: Y to X [22]. It ensures that these mappings are reverses of each other and that both mappings are bijections by using cycle consistency loss, which encourages $F(G(x)) \approx x$ and $G(F(y)) \approx y$ [22]. In other words, CycleGAN consists of two GANs, with two generators and two discriminators.

Finally, the most straightforward code for image-to-image translation using GANs will be illustrated in this section. It will illustrate how cycleGAN can extract image features from a specific domain and translate them into another domain. CycleGAN can do this even when paired training examples are unavailable.

Dataset

CycleGAN can translate images without relying on paired training examples. However, datasets containing paired source and target images can be used. The reason for using paired images is because cycleGANs translate source to target images and target to source using two GANs. Thus, it consists of two generators and two discriminators. The first generator translates the source image into the target image, while the second generator performs the inverse translation. The first discriminator distinguishes between the real and generated target images from generator 1. However, the second discriminator aims to differentiate between the real source image and the image generated from generator 2. Therefore, paired images are essential in this context. The dataset loading and preprocessing process is discussed in Figs. 3.15 and 3.29. This involves loading all images from a specific path, transforming them into numerical arrays, and normalizing them. It is worth noting that the loading and preprocessing of images in cycleGAN follow the same procedure as in any GAN model.

Model Design

Assuming the image-to-image translation task involves transforming an elephant image into a rhinoceros image, the first generator of the cycleGAN is responsible for generating rhinoceros images from elephant images. Conversely, the second generator generates elephant images from rhinoceros images. The first discriminator distinguished between real rhinoceros images and generated rhinoceros images. Similarly, the second discriminator differentiates between real and generated elephant images.

Generator Model

The generator model used in cycleGAN for image-to-image translation follows the same approach as other GAN models used for image translation. The generator models comprise an encoder-decoder architecture. In this context, generators 1 and 2 are referred to as F and H, respectively. Both generators have the same number of layers, the same model architecture, and the same hyperparameters. They follow the generator model discussed in Fig. 3.18. However, unlike batch normalization used in lines 24, 30, 34, and 43, cycleGAN uses instance normalization. Instance normalization is preferred because it normalizes each sample independently, unlike batch normalization, which normalizes samples in a batch. By normalizing the activations of each feature map across spatial dimensions, instance normalization stabilizes and speeds up the training process of CycleGAN.

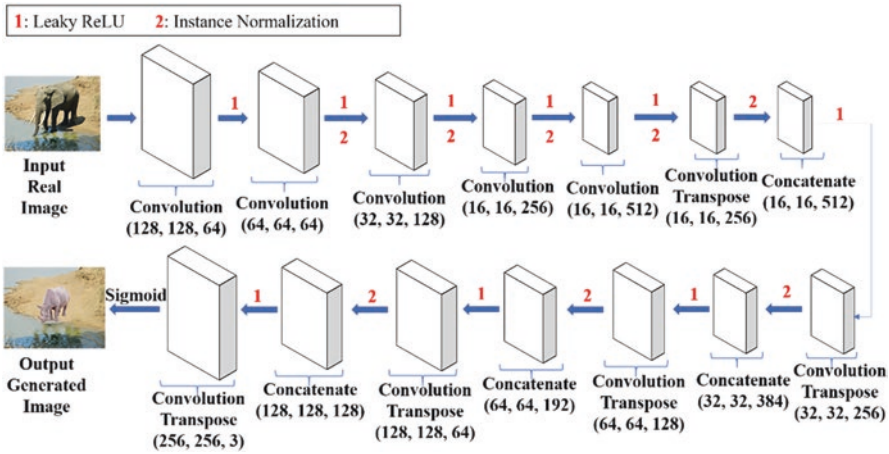

Fig. 3.42 Generator model architecture

The image-to-image translation process in CycleGAN involves an elephant image (E) and a rhinoceros image (R). Generator F extracts feature from image E and use them to generate an image R. In contrast, generator H extracts features from image R and uses them to generate image E. The generator process, illustrated in Fig. 3.42, shows the architecture of the generator model labeled as (F). Generator (H) has the same architecture but with inversed input and output.

Discriminator Model

In CycleGAN, the discriminator is crucial in distinguishing real from fake images. As with any other GAN discriminator, it is responsible for assessing the authenticity of the images presented to it. In particular, in CycleGAN, two discriminators are used, Discriminator 1 and Discriminator 2.

Discriminator 1, labeled DR, differentiates between real images (R) and the generated F(E). F(E) means rhinoceros-generated images by applying generator F on elephant images. On the other hand, Discriminator 2, labeled as D_E, is responsible for distinguishing between real images (E) and the generated images H(R). H(R) means elephant-generated images by applying generator H on rhinoceros images. This setup lets the model learn the mappings between two domains and achieve cycle consistency.

In terms of implementation, the same discriminator model used in traditional GAN image-to-image translation can be used for CycleGAN. This was discussed in detail in Sect. 3.3. The model should have the same number of convolutional layers, hyperparameters, and input and output specifications. The code can be implemented similarly to that presented in Fig. 3.20. However, in contrast to the batch normalization used in the generator, instance normalization will be utilized in lines 11, 14, and 17. Figure 3.43 presents the D_E architecture similar to that illustrated in Fig. 3.19.

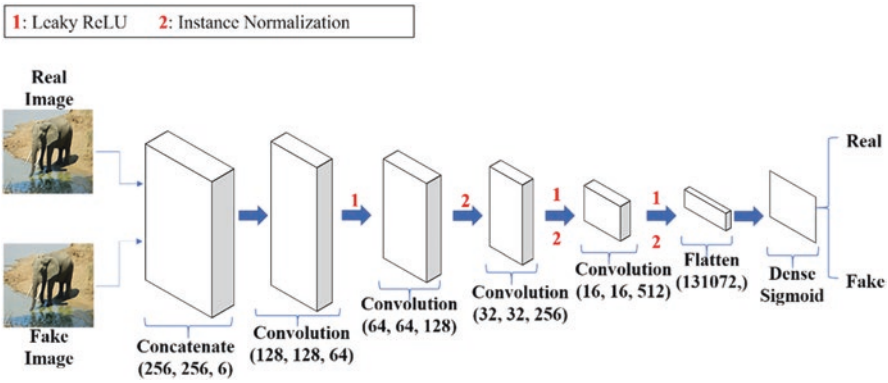

Fig. 3.43 Discriminator model architecture

The architecture of D_R is similar to $D_{E,}$ whereas the input will be real and fake images of the rhinoceros.

Training Stage

During the training stage of cycleGAN, the generator is fed with data to produce fake images. The training data is divided into random batches. At each epoch, data of batch size is inputted into the generator. The generator loss and discriminator loss are computed in the training loop. The discriminator loss is computed *using binary cross-entropy*, a common loss function for GAN models. CycleGAN differs in the generator loss, where a *cycle-consistency* loss is introduced in addition to the *binary cross-entropy* loss. Previously, the image-to-image translation generator model discussed in Sect. 3.3 was evaluated based on two losses: *binary cross-entropy* and *mean squared error*. However, in CycleGAN, the *cycle-consistency* loss is used instead of the mean squared error. The goal of cycleGAN is to ensure that the generated images are consistent in both directions, which means that $H(F(E)) \approx E$ and $F(H(R)) \approx R$, where E and R are real images of elephants and rhinoceros, respectively. The cycle-consistency loss computes the difference between the generated and the corresponding input images. For example, the real elephant image has to be almost similar to the image generated by generator H while passing the fake rhinoceros image generated by generator F from the real elephant image. Therefore, the cycle-consistency loss is calculated at each epoch. Figure 3.44 illustrates the cycle-consistency loss.

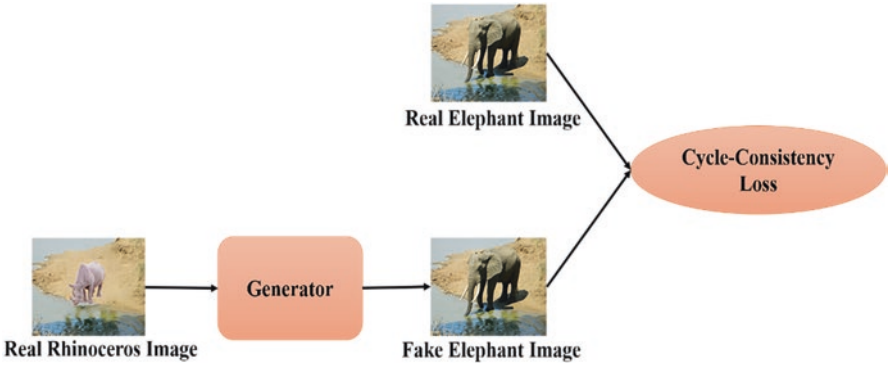

Fig. 3.44 Cycle-consistency loss

3.6 Enhancing Image Resolution

Enhancing image resolution means increasing the size of an image while maintaining high quality. This is also known as recovering high-resolution images while maintaining details from low-resolution ones. Improving image quality is not limited to photographs, medical images, paintings, or compressed images. Many traditional methods have been used, such as reducing the noise, adjusting the color, interpolating images, and up-scaling.

Deep neural networks have succeeded in many fields, especially in image processing. This success has enabled GANs to increase image resolution through training on a pair of images. The image pairs include low and their corresponding high-resolution label. After successful training, the GAN model will produce a high-resolution image without its corresponding high-resolution label.

Improving image quality through GANs is used to solve the limitations of traditional methods. Although traditional algorithms are easier to implement, they might generate distorted or blurry images instead of improving the resolution. The importance of improving image translation lies in the model's ability to learn features from low-resolution images and generate high-resolution ones. This section will discuss implementing a GAN model to improve high-resolution image resolution.

Dataset

To enhance image resolution using the GAN model, a dataset of paired low-resolution images and their corresponding high-resolution images is needed. Many image datasets can be used as DIV2K [25, 26] and super image resolution [27]. These datasets are publicly available for research and development purposes.

The process of preparing data for improving the resolution of images using GANs is similar to that of other image generation tasks using GANs. Initially, the

image data is transformed into floating tensor pixels and then normalized to a specific range, either [0, 1] or [-1, 1], based on the task requirements. Loading and preprocessing image data is similar to that presented in previous sections, as discussed in Fig. 3.15, lines 6 to 12 and Fig. 3.29, lines 6 to 18.

Model Design

The GAN model is composed of a generator and a discriminator. The generator model is responsible for producing high-resolution images. It takes low-resolution images as input and produces high-resolution images as output. On the other hand, the discriminator's objective is to differentiate real high-resolution images from generated ones.

Generator Model

The low-resolution images are passed as input to the generator. The generator model is composed of several residual blocks. These blocks are composed of convolutional layers, batch normalization, and parametrized *ReLU* (PReLU). Using parametric *ReLU* has an advantage over *LeakyReLU* in enhancing image resolution GAN generator. In parametric *ReLU*, the slope for negative inputs is a learnable parameter, which the neural network figures out itself. It will speed up the network training and solve the "dying *ReLU*" problem. These residual blocks are skipped connection blocks that learn from residual functions concerning the layer inputs instead of learning unreferenced functions. They are introduced as a part of the ResNet architecture. Several up-sampling layers follow the residual blocks to reach the desired image dimensions. The up-sampling layers can be built using the *Upsampling2D* function followed by a convolutional layer with a stride of 1 or built using convolution transpose. The batch normalization layer and *LeakyReLU* activation function follow the up-sampling layers. Finally, as any other generator model, it will end with a *Sigmoid* activation function if images are normalized between [0,1] or *Tanh* if the input images are normalized between [-1,1]. The general architecture of the generator is provided in Fig. 3.45. The code implementation will follow the same implementation procedure of generator models discussed before while following the architecture in the figure.

Discriminator Model

The discriminator model is similar to any discriminator GAN model. The binary classification deep convolutional layer model states whether the high-resolution images are real or generated. The discriminator model takes two input images, the fake high-resolution image, with the real one. Thus, the enhancing image resolution

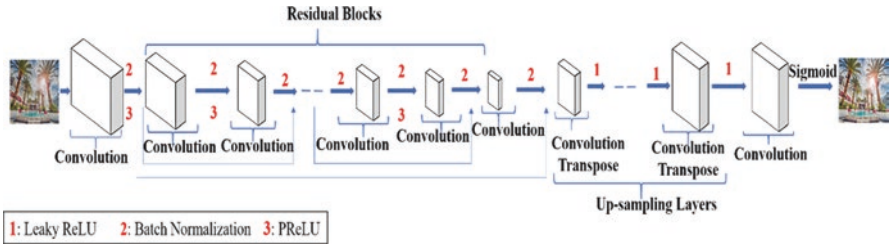

Fig. 3.45 Generator model architecture

discriminator model is similar to that implemented for image-to-image translation. The model architecture and implementation code are presented in Figs. 3.19 and 3.20, respectively. First, the two input images are merged into a single tensor using the concatenate function. To concatenate inputs, they must have the same size. In enhancing image resolution, the generator outputs high-resolution images with sizes equal to actual high-resolution images, so no previous steps are required. Then the output image tensor is passed through multiple down-sampling layers, i.e., convolutional layers with increasing filters followed by batch normalization and *LeakyReLU*. The down-sampling layers will allow for extracting features from the image. These features can be used to classify whether the generated image is real or fake. Precisely, to measure how the generated images are similar to actual ones.

Training Stage

Training the enhancing image resolution is similar to any image-to-image translation GAN model. The data is divided into batches trained over several epochs. In each epoch, the generator model will generate high-resolution images from the random low-resolution images chosen from the training dataset. The discriminator assesses the generator's ability to produce realistic images. It is trained on both generated and real images and computes the discriminator loss, a *binary cross-entropy* loss function. Then the generator loss will be computed to assess how much the generator can fool the discriminator. In any image, to image translation, two GAN losses are computed the *binary cross-entropy*, which sees how much the generator is fooling the discriminator and the *mean squared error*, which measures how much the generated image resembles the realistic ones. The training stage ends when the GAN model reaches convergence. Figure 3.46 shows an example of a high-resolution generated image from a low-resolution image. GAN succeeded in enhancing the resolution of the image.

Fig. 3.46 Enhancing image resolution GAN model output

3.7 Semantic Image Inpainting

Ancient images are distorted in multiple ways and are highly damaged. Sometimes images will be torn or missing a specific region, thereby requiring correction by completing the missing region based on the image data. Traditionally, image distortion was fixed with the help of an artist. However, this technique requires much effort and takes much time to complete. Repairing damaged images was tiring, with much attention to image details and imagination to complete the missing parts.

With the progress of time, multiple tools that can help repair images have been suggested. These tools can help restore images, fix anomalies, remove watermarks, and remove unwanted objects. For example, tools like Adobe Photoshop and GNU Image Manipulation Program (GIMP) can help in photo editing by removing or adding objects. However, these tools might fail in case of significant damage to the image.

GANs have shown their success in many image applications as they can produce high-quality images. They can generate images from text, enhance the resolution of images, translate images, or even produce new unseen images. These GAN capabilities have enabled it to repair images, such as filling a specific missing image region. This process is known as semantic image inpainting.

Dataset

A dataset containing distorted images with their corresponding corrected labels must be used to build a GAN model. Additionally, image-to-image datasets can also be used for semantic image inpainting. These datasets must include input images with missing objects or parts. Moreover, a dataset of single complete images can be

used. Nevertheless, this dataset requires preprocessing to create training and test data image pairs. Figure 3.47 illustrates a Python code for creating an image data pair for semantic image inpainting GAN. The code starts by defining the directory, which includes the original images and the other directory in which the distorted images will be saved (lines 2 and 3). The original image is distorted by zeros filling the pixels in specific regions. Choosing the region and filling zero pixels can be found in lines 23 to 36. The code output for a single image is presented in Fig. 3.48. The image used to test the code was taken from the Adobe website[1]. This code can build training and test data for a semantic image in painting the GAN model from a dataset of single images. The original image presented on the left side of Fig. 3.48 is the target image needed to be produced by the generator. The distorted image presented to the right of Fig. 3.48 is the input image to the generator, which needs to be enhanced.

Model Design

The GAN model for enhancing image resolution is composed of a generator and discriminator. The generator model is responsible for correcting distorted images by getting distorted images as input. At the same time, the discriminator model aims to differentiate between actual original images and corrected generated images.

Generator Model

The semantic image inpainting GAN generator model is similar to the model implemented before for image-to-image translation. It is made up of an encoder-decoder model. The encoder model is composed of many down-sampling. In contrast, the decoder model is composed of multiple up-sampling layers. The model architecture and code are in Figs. 3.17 and 3.18, respectively. The encoder comprises several convolutional layers with batch normalization and *ReLU* activation functions. The decoder model comprises multiple up-sampling layers to reach the target image size. The output from the generator is a restored image from the distorted input image.

Discriminator Model

The image inpainting discriminator model has similarities with any other GAN discriminator model. It combines actual original and generated restored images as input. It consists of several down-sampling layers, including convolutional layers,

[1] "Convert a Color Image into Black and White", https://helpx.adobe.com/photoshop/using/convert-color-image-black-white.html

```
# define original and distorted images directories
original_directory = str("/content/original/")
distorted_directory = str("/content/distorted/")
# Define the percentage of image to be removed
region_scale = 0.15
# Save existing files in the target directory in a list
all_files = os.listdir(distorted_directory)
# Enumerate over all files in source directory
for filename in os.listdir(original_directory):
    # Take only image files
    if filename.endswith('.png'):
        # Open image files
        img = Image.open(original_directory + filename)
        # Get image dimensions
        width, height = img.size
        # Scale the box size that based on image dimensions
        box = round(min(width,height) * region_scale)
        # Name the distorted image file
        distorted_filename = filename.replace('.png','') + '_distorted' + '.png'
        # Process only unprocessed files
        if (distorted_filename) not in all_files:
            # Set row number to place the box
            row = 2
            # Set column number to place the box
            column = 4
            # Define box dimension
            box_size = box
            # Define box starting coordinates
            x_side = round(box*0.8) + (box * column)
            y_side = round(box*0.8) + (box * row)
            # Define box coordinates
            shape = [(x_side, y_side), (x_side + box_size, y_side + box_size)]
            # Draw the source image
            img1 = ImageDraw.Draw(img)
            # Remove a the desired region from the source image
            img1.rectangle(shape, fill = 0)
            # Save the distorted image
            img.save(distorted_directory + distorted_filename)
```

Fig. 3.47 Removing specific region from image

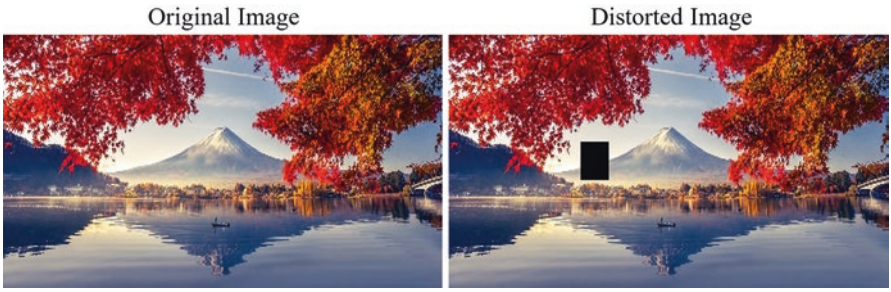

Fig. 3.48 Original and distorted images

batch normalization, and *LeakyReLU*. It will proceed in the same way as the design and discriminator code in Figs. 3.19 and 3.20. The deep convolutional layer model uses binary classification to determine whether the corrected images are fake or genuine.

Training

Training an image inpainting translation model is comparable to training any other image-to-image translation model. The data is split up into training batches that span several epochs. The generator model will create a batch every epoch to replace any damaged or missing portions of the input pictures picked from the training dataset. The discriminator, trained on both generated and real images, will evaluate the ability of the generator to create realistic images. It also calculates the discriminator loss, which is a *binary cross-entropy.* The generator loss will then be calculated to determine how well the generator can trick the discriminator. The binary cross-entropy, which determines how much the generator is deceiving the discriminator, is calculated as one of two GAN losses in any image-to-image translation. The training process stops when both the generator and discriminator models converge. To illustrate, when the generator can restore distorted images in a way that the discriminator cannot differentiate between generated and original images.

3.8 Text to Speech

Text-to-speech means transforming input text data into audio signals. This transformation type is a sequence-to-sequence model, where discrete data (text) is transformed into continuous data (audio signals). A male or a female voice can speak this audio. The voice used must be natural and represent an authentic human voice.

Many deep learning techniques have been used in literature to transform text data into speech signals. Some of the proposed models are WaveNet [16], SampleRNN

[17], DeepVoice 1 [18], DeepVoice 2 [19], DeepVoice 3 [20], and WaveRNN [21]. The WaveNet model uses CNN to convert text to voice but is very slow. For this reason, RNN models such as SampleRNN and DeepVoice have been introduced. To improve text-to-speech synthesis, DeepVoice 1 has been updated to DeepVoice 2 and 3 to demonstrate significant audio quality improvement and allow multi-speaker text-to-speech synthesis. However, traditional deep learning models have limitations as they might produce a nonrealistic voice. Hence, GAN was introduced.

Using text-to-speech GAN, a model can generate high-quality audio signals from a given text sequence. The text-to-speech GAN model is like any other one, consisting of a generator and a discriminator. The model needs a training dataset to learn and generate speech data from any text model.

Dataset

Any dataset consisting of audio clips and their corresponding text is considered beneficial for the text-to-speech GAN model. Some dataset suggestions for implementing the model are:

1. The LJ Speech Dataset comprises 13,100 short audio recordings for an individual speaker. The dataset comprises the audio recordings and the transcription [28].
2. The Common Voice Dataset consists of audio recordings and corresponding text labels. Unlike the LJ speech dataset, the audio snippets ate taken from 60,000 contributors instead of a single speaker [29].
3. LibriSpeech Dataset contains audio segments for 2,484 speakers. It is available for non-commercial use [30].

Multiple publicly available datasets can be used for transforming text to speech. However, textual representations and audio must be preprocessed before training the text-to-speech synthesis model.

Data Preprocessing

Data preprocessing in the text-to-speech model is divided into two components: preprocessing text and preprocessing audio. Processing the text labels is similar to preprocessing the text data in the text-to-image GAN model. It includes lowercase/uppercase all input words, removes punctuations as commas, ends each utterance by a period, for example, and conveys a particular emotional tone to words. Adding emotion to words involves giving scores to words where happy words have high scores and sad words have low scores. The second step in text preprocessing is phonology. This step is where phonetic transcriptions are assigned to processed words. The transformation of preprocessed text data into phonetics requires using a specific module. The publicly available module "*nltk*" can be used. This module is

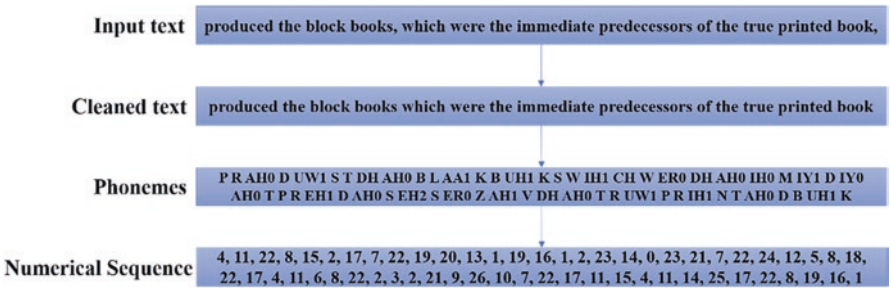

Fig. 3.49 Text preprocessing sample output

implemented using Python. It is able to transform words into their corresponding written pronunciation based on CMU pronouncing dictionary. Figure 3.49 visually illustrates the steps to reach a preprocessed text sample from the LJ speech dataset. Figure 3.50 presents a code snippet that shows how to use the module to transform the text into phonetics. It starts by loading the text data into a data frame (line 2), preprocessing the text (lines 4 to 13), transforming it to phonetics (lines 15 to 27), and then to numerical sequences (lines 29 to 38). The text phonemes are transformed into a sequence of numbers because deep learning models are able to process numerical data.

The audio data have to be also preprocessed. It must be preprocessed to ensure it is in the correct format for text-to-speech translation. The sound signals can be represented in the time or frequency domains. The use of the frequency domain is more popular than the time domain. Sound preprocessing involves resampling, which means converting the audio into a standard sample rate like 16kHz or 24kHz. Then, the features are extracted from audio data by transforming the audio waves to a sequence using short-term Fourier transform (STFT) or Mel-frequency cepstral coefficients (MFCCs). Thus, the audio waves will transform into feature vectors. Fig. 3.51 shows sound preprocessing by transforming the sound waves to a mel-spectogram. For example, the mel-spectogram is normalized between [-1, 1].

Model Design

Like any other GAN model, the text-to-speech synthesis GAN comprises a generator and discriminator. The generator aims to produce audio snippets for corresponding text labels. The discriminator aims to differentiate between real audio segments and generated ones.

```
# Load text data and save it in pandas data frame
df = pd.read_csv('metadata.csv')
# Define the function to preprocess text
def preprocess_text(text):
    # Remove punctuation from the text
    text = text.translate(str.maketrans("", "", string.punctuation))
    # Convert to lowercase
    text = text.lower()
    # Tokenize the text
    tokens = word_tokenize(text)
    return " ".join(tokens) +'.'
# Apply the processing function to the sentences column in the DataFrame
df["preprocessed_sentences"] = df["sentence"].apply(preprocess_text)
# Download the CMU Pronouncing Dictionary from nltk library
nltk.download('cmudict')
# Load the CMU Pronouncing Dictionary
phonetics = cmudict.dict()
# Define a function to convert text to phonemes
def text_to_phonemes(text):
    # Define an empty list to store word pronunciation
    phonemes = []
    # Enumerate over each word in the sentence
    for word in text:
        if word in phonetics:
            # Convert each word to its list of phonemes
            phonemes.extend(phonetics[word][0])
    return phonemes
# Define a function to convert phonemes to a numerical sequence
def phonemes_to_sequence(phonemes):
    # Define a mapping from phonemes to unique integers
    all_phonems = []
    # Keep only unique phonemes
    phoneme_set = set(all_pronounciations)
    # Map each phoneme to a specific number
    phoneme_to_int = {phoneme: i for i, phoneme in enumerate(phoneme_set)}
    # Convert each phoneme to its corresponding integer
    sequence = [phoneme_to_int[phoneme] for phoneme in phonemes]
    return sequence
# Transform text data to phonemes
word_pronounciation = df["preprocessed_sentences"].apply(text_to_phonemes)
# Transform the n-D word pronounciation list to 1D
```

Fig. 3.50 Text preprocessing

```
# Define Sampling Rate
sr = 22050
Define number of FFT points
n_fft = 2048
# Define number of samples between frames
hop_length = 512
# Define number of Mel frequency bins
n_mels = 128
# Load audio files and transform into mel-spectogram
mel_spec_norm_all = []
all_files = os.istdir('/content/wavs')
for i in all_files:
    audio, sr = librosa.load('/content/wavs'+i, sr=sr)
    # Extract mel- spectogram
    mel_spec = librosa.feature.melspectrogram(y=audio, sr=sr, S=None, n_fft=n_fft,,
hop_length=hop_length, n_mels=n_mels,  fmax=sr/2)
    # Convert the mel-spectogram to decibels
    mel_spec_db = librosa.power_to_db(mel_spec, ref=np.max)
    # Normalize between o and 1
    mel_spec_norm = (mel_spec_db - mel_spec_db.min()) / (mel_spec_db.max() -
mel_spec_db.min())
```

Fig. 3.51 Audio preprocessing

Generator Model

The generative text-to-speech model is a sequence-to-sequence model. It maps discrete data into a continuous time/frequency domain. The input to the generator model is the sequence of numerical data representing the phonemes. The generator model is composed of multiple deep neural network layers. The numerical sequence passes through an encoder model in order to extract embeddings from phonemes. Then the text embeddings will pass through multiple up-sampling convolutional layers followed by the *ReLU* activation function and batch normalization. The output from the up-sampling layers will be a sequence representing audio samples. Figure 3.52 shows the sample architecture for the text-to-speech generator. All arrows labeled "1" indicates Batch Normalization with ReLU activation function.

Discriminator Model

The discriminator model is a binary classification model with two real or fake outputs. The discriminator model will take the mel-spectograms as input. It also can be implemented by taking two inputs: the numerical text representation and the audio

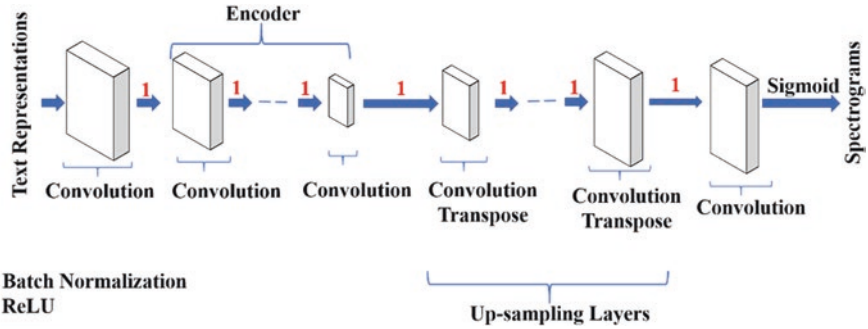

Fig. 3.52 Generator architecture

spectrograms. Using two inputs in a text-to-speech discriminator will ensure that the produced audio signals describe the written text. It consists of multiple down-sampling layers that consist of convolutional layers followed by *ReLU* activation functions. The output from the discriminator will state whether the visual representation of audio, mel-spectograms are real or fake. The discriminator architecture is similar to that presented in Fig. 3.6 however, instead of the letter A images, it accepts spectrograms as inputs. However, if the generator is designed to be fed with two inputs, namely text representations and spectrograms, the process would be similar to text-to-image translation presented in Fig. 3.33, with the only difference being that the second input being imaged is spectrograms.

Training

The training process is similar to any GAN model. During training, two types of losses have computed the generator and discriminator loss. The generator loss will measure the difference between generated and real audio segments. While the discriminator will measure how much the generated audio segment is realistic and similar to real ones. Throughout the training stage, the generator updates its weights based on the feedback taken from the discriminator. The generator will continue updating its weights and improve its output quality while the discriminator is able to distinguish between real and generated speech signals. When the model reaches convergence, the spectrograms produced by the generator can be transformed back into audio signals. The mean opinion score will be utilized to better assess the text-to-speech model. It is a numerical measure between 1 and 5 that assesses the quality of generated audio waves. The higher the mean opinion score, the better the audio signal quality is.

References

1. Liu, Z., Luo, P., Wang, X., Tang, X.: Deep learning face attributes in the wild. In: 2015 IEEE International Conference on Computer Vision (ICCV), pp. 2380–7504 (2015)
2. Kottarathil, P.: Face Mask Lite Dataset, Kaggle, Kaggle Data. https://www.kaggle.com/datasets/prasoonkottarathil/face-mask-lite-dataset (2020)
3. Patel, S.: A-Z Handwritten Alphabets in. CSV Format, Kaggle, Kaggle Data. https://www.kaggle.com/datasets/sachinpatel21/az-handwritten-alphabets-in-csv-format(2018)
4. Sashaborm: Thispersondoesnotexist - Random AI Generated Photos of Fake Persons, This Person Does Not Exist - Random Face Generator, Google. https://this-person-does-not-exist.com/en (2021)
5. Lytic: FaceForensics++, Kaggle, Kaggle Data. https://www.kaggle.com/datasets/sorokin/faceforensics (2020)
6. VoxCeleb: A Large Scale Audio-Visual Dataset of Human Speech. https://www.robots.ox.ac.uk/~vgg/data/voxceleb/ (2018)
7. Tulyakov, S., Liu, M.-Y., Yang, X., Kautz, J.: Mocogan: decomposing motion and content for video generation. In: 2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition (2018)
8. Vondrick, C., Pirsiavash, H., Torralba, A.: Generating Videos with Scene Dynamics. https://arxiv.org/abs/1609.02612 (2016)
9. Karras, T., Aila, T., Laine, S., Lehtinen, J.: Progressive Growing of Gans for improved Quality, Stability, and Variation. https://arxiv.org/abs/1710.10196 (2018)
10. Sun, X., Xu, H., Saenko, K.: Twostreamvan: Improving motion modeling in video generation. In: 2020 IEEE Winter Conference on Applications of Computer Vision (WACV) (2020)
11. Finger, L.: Overview of How To Create Deepfakes – It's Scarily Simple, Forbes. Forbes Magazine. https://www.forbes.com/sites/lutzfinger/2022/09/08/overview-of-how-to-create-deepfakesits-scarily-simple/?sh=73b154b12bf1 (2022)
12. Aldausari, N., Sowmya, A., Marcus, N., Mohammadi, G.: Video generative adversarial networks: a review. ACM Comput. Surv. **55**, 1–25 (2022)
13. The cityscapes dataset, Cityscapes dataset, https://www.cityscapes-dataset.com/
14. Saha, A.: Satellite-Googlemaps-Masks, Kaggle, Kaggle Data. https://www.kaggle.com/datasets/arka47/satellitegooglemapsmasks (2021)
15. Zhu, J.-Y., Park, T., Isola, P., Efros, A.A.: Unpaired image-to-image translation using cycle-consistent adversarial networks. In: 2017 IEEE International Conference on Computer Vision (ICCV) (2017)
16. Isola, P., Zhu, J.-Y., Zhou, T., Efros, A.A.: Image-to-image translation with conditional adversarial networks. In: IEEE conference on computer vision and pattern recognition, pp. 1125–1134. IEEE (2017)
17. Lin, T.-Y., Maire, M., Belongie, S., Hays, J., Perona, P., Ramanan, D., Dollár, P., Zitnick, C.L.: Microsoft Coco: Common Objects in Context. Computer Vision – ECCV. **2014**, 740–755 (2014)
18. Hsankesara: Flickr Image Dataset, Kaggle. Kaggle Data. https://www.kaggle.com/datasets/hsankesara/flickr-image-dataset (2018)
19. Adityajn105: Flickr 8K Dataset, Kaggle. Kaggle Data. https://www.kaggle.com/datasets/adityajn105/flickr8k (2020)
20. Rashtchian, C., Young, P., Hodosh, M., Hockenmaier, J.: Collecting Image Annotations Using Amazon's Mechanical Turk. In: Proceedings of the NAACL HLT 2010 Workshop on Creating Speech and Language Data with Amazon's Mechanical Turk, vol. 2010,
21. 102 category Flower Dataset, Visual Geometry Group - University of Oxford, https://www.robots.ox.ac.uk/~vgg/data/flowers/102/
22. Zhu, J.-Y., Park, T., Isola, P., Efros, A.A.: Unpaired image-to-image translation using cycle-consistent adversarial networks. In: 2017 IEEE International Conference on Computer Vision (ICCV)., pp. 2223–2232 (2017)

23. Kaneko, T., Kameoka, H.: Parallel-data-free voice conversion using cycle-consistent adversarial networks. https://arxiv.org/abs/1711.11293
24. Bargshady, G., Zhou, X., Barua, P.D., Gururajan, R., Li, Y., Acharya, U.R.: Application of cyclegan and transfer learning techniques for automated detection of COVID-19 using X-ray images. Pattern Recognit. Lett. U.S. National Library Med. **153**, 67–74 (2022)
25. Agustsson, E., Timofte, R.: NTIRE 2017 Challenge on Single Image Super-resolution: Dataset and study. In: 2017 IEEE Conference on Computer Vision and Pattern Recognition Workshops (CVPRW) (2017)
26. Timofte, R., Agustsson, E., Van Gool, L., Yang, M.-H., Zhang, L., Lim, B.: NTIRE 2017 challenge on single image super-resolution: methods and results. In: 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR) Workshops (2017)
27. Kapse, A.D.: Super Image Resolution, Kaggle, Kaggle Data. https://www.kaggle.com/datasets/akhileshdkapse/super-image-resolution (2020)
28. The LJ speech dataset, Keith Ito. https://keithito.com/LJ-Speech-Dataset/
29. Mozilla Common Voice, Common Voice. https://commonvoice.mozilla.org/en/datasets
30. Panayotov, V., Chen, G., Povey, D., Khudanpur, S.: Librispeech: an ASR corpus based on public domain audio books. In: 2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP) (2015)

Chapter 4
Conclusion

GANs are a type of deep learning model subset of machine learning that has been a hot area of research in recent years. Its architecture comprises two networks: the generator and the discriminator are mainly made up of a deep neural network. The generator model aims to generate new data that looks like the real one. Typically, the generator model is composed of multiple up-sampling layers. The discriminator aimed to differentiate between actual data and generated one. The discriminative model is made of multiple down-sapling layers. The discriminator is a typical binary classifier deep neural network. The discriminator gives the generator feedback to update its weights and yield more realistic results.

In the previous chapters, multiple real-world applications have been discussed. Chapter two starts by introducing how to build and train GAN models. The most straightforward GAN application has been chosen to enhance the reader's understanding of GANs. GAN can be applied to generate 1D output as a single point, text, images, and speech. It has shown success in many applications. The input to the generator model could be either noise or specific input.

This book comprehensively explains some popular applications and code snippets to help improve understanding. Although GAN has succeeded in many real-world applications, it has some weaknesses. To produce high-resolution data similar to the actual one requires high computational resources, a large dataset, and a long time to converge.

Researchers and developers encounter several challenges when implementing GANs and conducting the study. One of the main issues is the high computational cost of training GANs, which can make them hard to scale and lead to long training periods. The size and complexity of the training dataset, the model's architecture, and the training algorithm all affect how much computing power is needed and how long it takes to train a model. GANs usually require a high-end computer or a cluster of computers with powerful CPUs and GPUs. Additionally, the GAN model is trained over several epochs, this might increase the training time and make them prone to overfitting, which can limit their ability to generalize to new data.

S. Kaddoura, *A Primer on Generative Adversarial Networks*, SpringerBriefs in Computer Science, https://doi.org/10.1007/978-3-031-32661-5_4

Another issue with GANs is that meticulous hyperparameter tuning, which can be time-consuming and labor-intensive. Nevertheless, it is highly needed for the generator and discriminator models in order to produce realistic results. Moreover, GANs frequently need a lot of training data to produce high-quality results, which can be a constraint in some uses. Some GAN applications lack the availability of publicly available datasets, which can result in the need to implement a large dataset. This process may take days, especially if the dataset is collected from scratch.

Despite these limitations, GANs have demonstrated excellent potential in producing lifelike images, videos, and other kinds of data. Ongoing GAN research aims to overcome these challenges and raise GAN models' general efficiency and scalability.