BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

SYNTHETIC FINGERPRINT GENERATION USING GAN

GENEROVÁNÍ SYNTETICKÉHO OTISKU PRSTU POMOCÍ GAN

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR **JIŘÍ DVOŘÁK**
AUTOR PRÁCE

SUPERVISOR **Ing. ONDŘEJ KANICH, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2020

Brno University of Technology
Faculty of Information Technology

Department of Intelligent Systems (DITS) Academic year 2019/2020

Bachelor's Thesis Specification

Student: **Dvořák Jiří**
Programme: Information Technology
Title: **Synthetic Fingerprint Generation Using GAN**
Category: Security

Assignment:

1. Study the literature on fingerprint biometric recognition and the generation of synthetic fingerprints. Learn the concepts of GAN (Generative Adversarial Network).
2. Design an algorithm using GAN to generate synthetic fingerprints.
3. Implement the proposed algorithm from the previous step.
4. Analyse the algorithm results from the previous step by generating a synthetic fingerprint database.
5. Summarise and discuss the results. Suggest possible extensions of your solution.

Recommended literature:

- Minaee, S., Abdolrashidi, A.: *Finger-GAN: Generating Realistic Fingerprint Images Using Connectivity Imposed GAN*, Preprint, 2018.
- Maltoni, D., Maio, D., Jain, A.K. and Prabhakar, S.: *Handbook of Fingerprint Recognition*. Springer, 2009, pages 512. ISBN 978-1-8488-2254-2.
- Kanich, O.: *Fingerprint Damage Simulation*, LAP LAMBERT Academic Publishing GmbH & Co. KG, 2014, p. 57. ISBN 978-3-659-63942-5.

Requirements for the first semester:

- Parts 1 and 2.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor: **Kanich Ondřej, Ing., Ph.D.**
Head of Department: Hanáček Petr, doc. Dr. Ing.
Beginning of work: November 1, 2019
Submission deadline: July 31, 2020
Approval date: January 10, 2020

Abstract

This thesis is focused on the generation of synthetic fingerprints using a model based on the principle of generative adversarial networks. The work summarizes the basic theoretical information about biometrics with emphasis on fingerprints. It also describes the principle of one of the popular synthetic fingerprint generators called SFinGe. The model based on a deep convolutional generative adversarial network is discussed together with several methods that improved its performance. The results were evaluated by computing the Fréchet Inception Distance between the generated and real fingerprints. The generated dataset of 100 samples was also evaluated by NFIQ 2.0 which proved that the proposed model is able to generate fingerprints with almost the same quality of the training samples.

Abstrakt

Tato bakalářská práce se zabývá generováním syntetických otisků prstů za pomoci modelu založeném na principu generativních soupeřících sítí. Práce shrnuje základní teoretické informace z biometrie se zaměřením na otisky prstů. Zaobírá se také principem jednoho z populárních generátorů syntetických otisků prstů – nástrojem SFinGe. Práce představuje model postavený na hluboké konvoluční generativní soupeřící síti a představuje několik metod, které vedly ke zlepšení jeho výkonu. Vyhodnocení výsledků bylo provedeno výpočtem „Fréchet Inception Distance“ mezi vygenerovanými a existujími otisky. Dále byl vygenerován dataset obsahující 100 snímků. Ten byl vyhodnocen nástrojem NFIQ 2.0, který ukázal, že model je schopný generovat otisky prstů kvality srovnatelné s reálnými trénovacími daty.

Keywords

fingerprints, synthetic fingerprints, fingerprint generation, deep neural networks, GAN, Fréchet Inception Distance, NFIQ 2.0

Klíčová slova

otisky prstů, syntetické otisky prstů, generování otisků prstů, hluboké neuronové sítě, GAN, Fréchet Inception Distance, NFIQ 2.0

Reference

DVOŘÁK, Jiří. *Synthetic Fingerprint Generation Using GAN*. Brno, 2020. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Ondřej Kanich, Ph.D.

Rozšířený abstrakt

V dnešní době jsou již technologie založené na snímání otisků prstů součástí běžného života. Umožnují rychlé odemknutí mobilního telefonu či dokonce vstupních dveří pouhým dotykem prstu. Vývoj těchto technologií ovšem vyžaduje důkladné testování, k čemuž je zapotřebí dostatečně velká databáze otisků prstů. Vytvoření vlastní databáze je však časově a finančně náročný proces a sdílení již existující databáze je mnohdy složité z důvodu ochrany osobních údajů. Proto je vhodné využít generátorů syntetických otisků prstů, které tyto problémy eliminují.

Otisky prstů mají velmi komplexní strukturu, kterou je mnohdy složité napodobit. Jedním z populárních generátorů otisků prstů je nástroj SFinGe, který při generování otisku nejprve vytvoří dokonalý otisk, který je následnými transformacemi upraven tak, aby působil realisticky. Tato práce se zabývá vytvořením generátoru syntetických otisků prstů založeného na modelu hlubokého učení. Konkrétně se jedná o generativní soupeřící síť, která se stala velmi populární v oblasti generování dat. Cílem je naučit tento model generovat syntetické otisky prstů, které budou nerozeznatelné od těch reálných.

Teoretická část práce nejprve shrnuje základní informace o biometrii se zaměřením na otisky prstů. Z těchto informací je čtenář schopný zhodnotit vizuální kvalitu výsledků této práce. Také je blíže popsán princip již zmíněného nástroje SFinGe. V další části jsou představeny umělé neuronové sítě a zejména jejich speciální typy, jako jsou konvoluční a generativní soupeřící sítě. Ty tvoří základní stavební prvky navrženého modelu. V praktické části práce jsou poskytnuty detailní informace o navrženém modelu, včetně popisu datasetu, který byl využit při jeho trénování. Pro zajištění stability trénovacího procesu a zvýšení kvality výsledných otisků prstů bylo implementováno několik metod, které jsou také popsány v této části. V poslední kapitole se práce věnuje implementačním detailům navrženého modelu a vyhodnocení jeho výsledků.

Navržený model je založený na principu generativní soupeřící sítě, která obsahuje dvě dílčí neuronové sítě. První z nich je generátor, jehož úkolem je vygenerovat ze vstupního vektoru hodnot syntetický otisk prstu v rozlišení 96×96 pixelů. Naproti tomu druhá síť, nazývaná diskriminátor, má na vstupu dva snímky otisků prstů – jeden z reálného datasetu a druhý vyprodukovaný generátorem. Na základě těchto snímků provádí klasifikaci, zda se jedná o reálné či syntetické snímky. Cílem generátoru je, aby diskriminátor nebyl schopný rozlišit reálný snímek od syntetického. Cílem diskriminátoru přitom je, aby dosáhl co nejvyšší rozlišovací schopnosti. Jelikož jsou tyto dvě neuronové sítě trénovány současně a soupeří navzájem mezi sebou, je stěžejní udržet proces trénování stabilní. Tedy zajistit, aby jedna ze sítí znatelně nepřekonávala druhou. Toho bylo dosaženo po implementaci metody spektrální normalizace a modifikaci označení třídy dat. Některé reálné snímky byly během trénování označeny jako syntetické a naopak. Zároveň reálné otisky prstů nebyly označeny explicitní hodnotou 1, ale náhodnou hodnotou v rozmezí 0.9 až 1. Tyto uvedené metody vedly k výrazné stabilizaci trénování modelu. Přetrvával však problém s nízkou diverzitou generovaných snímků. Tento problém byl redukován implementací metody klasifikace napříč dávkou dat, která dává diskriminátoru možnost ohodnotit snímek s ohledem na ostatní smínky ve stejné dávce. To mu umožňuje rozpoznat nízkou diverzitu mezi snímky a tím také snadněji určit, že se jedná o výstup generátoru. Tím je generátor nucen k tomu, aby generoval data s vyšší diverzitou. Dále pomohla také metoda opakování zkušeností diskriminátoru, při které jsou diskriminátoru po určitém počtu kroků znovu ukázány některé ze starších vygenerovaných snímků. To nutí generátor, aby nezůstal u shodných snímku a více je modifikoval.

Model, po implementaci metod pro zlepšení výkonu, prokázal, že je schopný generovat syntetické otisky prstů, které jsou téměř nerozpoznatelné od snímků z reálného datasetu. Výsledky byly vyhodnocovány během trénování podle „Fréchet Inception Distance“ (FID), která je populární metrikou při vyhodnocování generativních soupeřících sítí jejichž cílem je generování obrazových dat. Metoda FID do výsledného skóre promítá jak kvalitu vygenerovaných snímků, tak jejich diverzitu. Následně byl vygenerován dataset obsahující 100 snímků, které byly vyhodnoceny nástrojem NFIQ 2.0, který se stal referenční implementací standardu ISO/IEC 29794-4. Nejlepších výsledků bylo dosaženo s modelem implementujícím veškeré popsané rozšiřující metody po 60 epochách. V té fázi model dosáhl FID 36,8 a průměrného skóre kvality vygenerovaných otisků prstů 18,21 podle nástroje NFIQ 2.0.

Synthetic Fingerprint Generation Using GAN

Declaration

I hereby declare that this Bachelor's thesis was prepared as an original work by the author under the supervision of Mr. Ing. Ondřej Kanich, Ph.D. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

.....
Jiří Dvořák
July 31, 2020

Acknowledgements

I would like to thank my supervisor Ing. Ondřej Kanich, Ph.D., for his professional support during both good and hard times. I would also like to extend my gratitude to my friends, who helped me with proofreading English. Computational resources were supplied by the project "e-Infrastruktura CZ" (e-INFRA LM2018140) provided within the program Projects of Large Research, Development and Innovations Infrastructures.

Contents

Chapter 1

Introduction

Since their formal acceptance in the nineteenth century for identifying an individual's identity, fingerprints became the most popular biometric characteristics [16]. They proved to be a very reliable method of authentication, and technologies based on their recognition are cheap and easy to use. These advantages are the main reasons why this technology carried over to our everyday life. However, as the popularity of fingerprints is rising, so does the need for more sophisticated recognition algorithms. These algorithms need large datasets of fingerprints to be tested appropriately; however, acquiring such a database is a very time and money consuming process. Once the database is collected, it is also complicated to share it because of privacy issues. These problems lead us to synthetic fingerprint generators. There are many existing approaches to generating synthetic fingerprints, but many fail to generate realistic-looking fingerprints since they are not able to express their complicated structure.

Less than a decade ago, an up-and-coming model for realistic image generation was proposed in paper [12] by Goodfellow et al. This model, called generative adversarial network, became a powerful tool for many computer vision and pattern recognition tasks. It is successfully used in areas like image-to-image translation, text-to-image synthesis, resolution enhancement, text/image/video content generation, and many more. Its usage in deep fakes, which became famous and frightening at the same time, has shown that they can capture detailed biometric characteristics.

In this thesis, a synthetic fingerprint generator based on the generative adversarial network is proposed. Currently only one publicly accessible project has implemented this approach and reached exciting results [27]. The goal is to generate highly realistic fingerprints that are indistinguishable from the real fingerprints.

Chapter 2 provides information about biometrics with emphasis on fingerprints, so the reader gets insight into the intent of this work. In Chapter 3, artificial neural networks are described, together with more advanced models such as convolutional neural networks, and generative models based on adversarial training that are important for this work. Chapter 4 describes the proposed solution, including the used dataset, the model architecture, and methods implemented to improve the performance of the proposed model. Chapter 5 provides implementation details of the proposed solution and evaluation of the results based on the *Fréchet Inception Distance* [14] as well as one of the conventional methods for the fingerprints quality evaluation. The last Chapter 6 provides a summary of this work and suggests possibilities for future work.

Chapter 2

Fingerprints and introduction to biometrics

People are in contact with fingerprint-sensing technologies so commonly that many of them accepted this technology as part of their everyday lives. People can easily log into their phones or notebooks, and they can even open a locked door with only the touch of a finger. In this section, the general information needed to understand this work is described. Since this thesis is focused on the generation of synthetic fingerprints, this chapter will cover a description of the term biometrics with the emphasis on fingerprints. Finally, the principle of conventional synthetic fingerprint generators is described.

2.1 Biometrics

In today's world, when technology is on the rise, there is also a risk of stolen identity growing. Given this, the reliable method of authentication of an individual's identity is more and more crucial.

When talking about how to prove our digital identity, there are three basic approaches:

- Showing the knowledge of a secret (e.g., password, PIN).
- Demonstrating the possession of something unique enough (e.g., ID card, security token).
- Satisfying physiological or behavioral requirements called biometric characteristics, or simply biometrics [15] (e.g., fingerprint, face, iris, signature). [17, 25]

One of the main reasons why biometrics is so widely used nowadays is that these characteristics cannot be easily lost, shared, guessed, copied, or eventually forged. Based on these aspects, biometrics is considered to be significantly more difficult to abuse than traditional authentication methods (knowledge, possession) [16]. However, once the biometric of a person is revealed, there is no simple way to change it [17].

Although fingerprints are probably the most widely known biometrics, any physiological or behavioral characteristic can be used as a biometric characteristic. According to [25], the basic properties, when comparing different biometrics, are:

- Universality – each person should have this trait.

- Distinctiveness – two persons should not have the same trait.
- Permanence – the trait should not vary over time.
- Collectability – the trait should be easy to acquire.
- Performance – the trait should not change or age.
- Acceptability – the public should allow the trait to be recorded and used.
- Circumvention – how difficult it is to forge this trait.

One of the reasons that make fingerprints one of the most widely used biometrics is their uniqueness. Sir Francis Galton's calculations stated that the likelihood of two fingerprints being the same is 1 to 64 billion [25]. Other primary advantages of this biometric are permanence, performance, circumvention, and the low price of deploying a biometric system [17].

2.2 Physiological information about fingerprints

Fingerprints are created by papillary lines, protrusions on the inner side of hands (and feet). Papillary lines are fully formed at about seven months of fetal development, and the process of their formation is also a reason why there is such a small chance of two individuals having the same fingerprint [25].

The general form of the fingerprint emerges as the skin on the fingertip begins to differentiate. This process is caused by the basal layer of the epidermis, which grows faster than dermis (inner layer) and epidermis (outer layer) layers around. That causes compressive stress in the basal layer, as shown in Figure 2.1. If the stress is large enough, a buckling causes the formation of the papillary line [22]. Additionally, the environment around the fetus continuously affects the process, including the position and movement of the fetus in the womb and the density of amniotic fluid [25].

Papillary lines can be seen and captured as a fingerprint; however, they are just a projection from the dermis layer to the epidermis layer. That means that it is not possible to change or delete the fingerprint by superficial injury because it will regenerate as the skin grows back [17, 25].

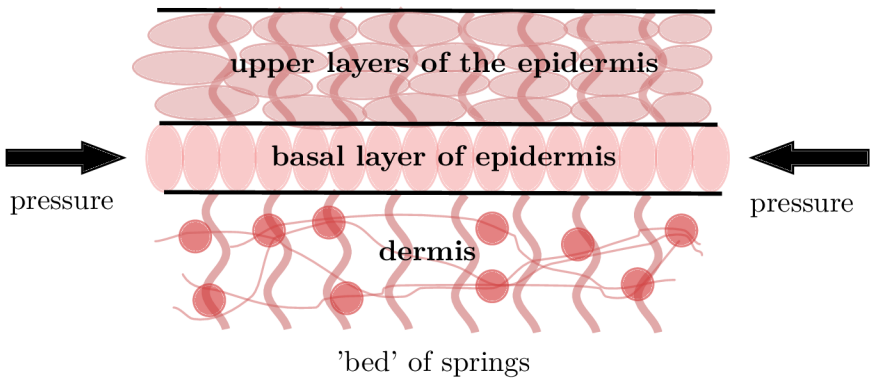


Figure 2.1: Basal layer of epidermis trapped between dermis and intermediate epidermis layers (inspired by [22]).

2.3 Classification of fingerprints

The FBI's IAFIS (Federal Bureau of Investigation's Integrated Automatic Fingerprint Identification System) processes tens of thousands of requests daily with hundreds of millions of records in a database [33]. With so many incoming requests, it would be challenging and inefficient to simply compare any two fingerprints against each other. Therefore, fingerprints are divided into classes to make this process more efficient. This allows us to reject immediately those from another class and focus just on those that belong to the same class [17].

IAFIS system uses Henry's classification system, which contains three basic classes – arch, loop, and whorl. Examples of these fingerprint classes can be seen in Figure 2.2. From these basic classes, other more specific ones are derived. The main characteristics to distinguish fingerprint classes are called delta and core. Delta is a place where papillary lines run to three different directions. Core is located in the innermost loop of a fingerprint [17].

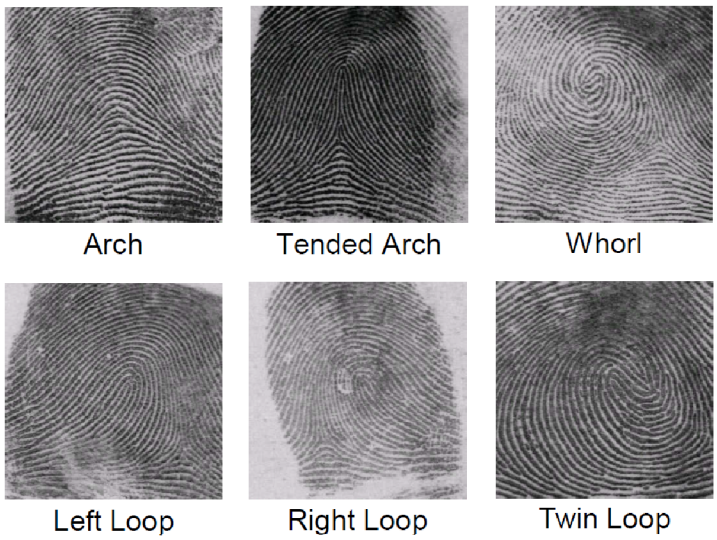


Figure 2.2: Different classes of fingerprints according to the Henry's classification (taken from [17]).

2.4 Fingerprint minutiae

The previous section defined how fingerprints can be separated into different classes, which helps to reduce the number of fingerprints that need to be checked. However, this still does not solve an issue with unambiguously identifying a person's identity.

To be able to distinguish every finger in the world, analyzing fingerprint minutiae is needed. Minutia is a specific formation created by papillary lines. Many of them are distinguished, each of them with a different likelihood of appearance [17]. Since it would be too demanding to work with all these complicated patterns, just two basic minutiae types for automatic recognition are being recognized in practice – ridge ending and bifurcation. Examples of those minutiae types can be seen in Figure 2.3. Nevertheless, these provide enough information to identify an individual's identity successfully [25].

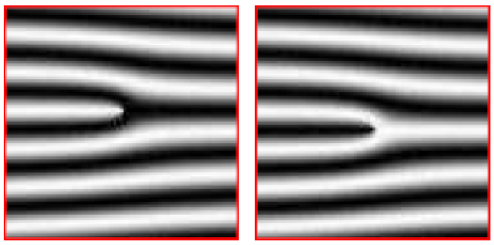


Figure 2.3: Fingerprint minutiae – bifurcation and ridge ending (taken from [17] and modified).

2.5 Fingerprint acquirement

For the automatic fingerprint recognition, getting a fingerprint into a digital form is the start of the process. Even though there are different methods for achieving that, the most convenient way is getting the fingerprint into the computer directly using a fingerprint sensor [17].

In this work, different fingerprint datasets for training a synthetic fingerprint generator were used to see how the best results can be achieved. These datasets were acquired using different sensor technologies. This section will present the standard sensor technologies used nowadays to get a better understanding of how fingerprints were acquired and what the limitations of these methods are.

2.5.1 Optical technology

One of the oldest fingerprint sensor technologies is based on a relatively simple optical principle indicated in Figure 2.4. A finger is placed onto a protective glass, so ridges (papillary lines) touch the glass, and valleys are in the distance. From a light source (LED), ray falls on the finger surface and is reflected by ridges and absorbed at valleys. CCD/CMOS camera then captures reflected rays through optics, which creates the final image of the fingerprint. The finger does not have to be necessarily placed on a surface – also, contactless sensors based on this technology exist [9]. The main advantages of this technology are the resistance to temperature fluctuations and possible operation in 3D. The disadvantage is a high sensitivity to dirty fingers. Except for the contactless technology, there is also a problem with latent fingerprints [17].

2.5.2 Thermal technology

This technology is based on thermal radiation. Ridges have higher thermal radiation than valleys. When a person sweeps a finger over a pyroelectric cell, it generates a current according to the temperature, which can be measured. Since the temperature equalizes quickly, it is necessary to use sweeping sensors. The advantage of these sensors is high resistance to electrostatic discharge [17]. The basic principle of sensors based on the technology described above is shown in Figure 2.5.

2.5.3 E-field technology

The sensor consists of a finger drive ring and a matrix of antennas as shown in Figure 2.6. The drive ring generates a sinusoidal radio frequency signal, and the matrix receives that

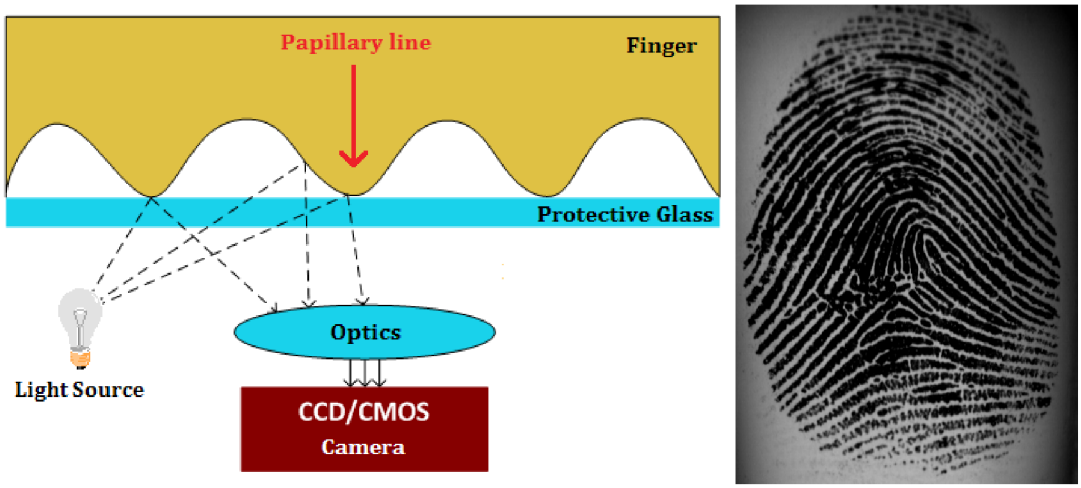


Figure 2.4: Optical sensor principle (taken from [17]), and the example of the fingerprint acquired by an optical sensor (taken from [4] DB2_A_dataset).

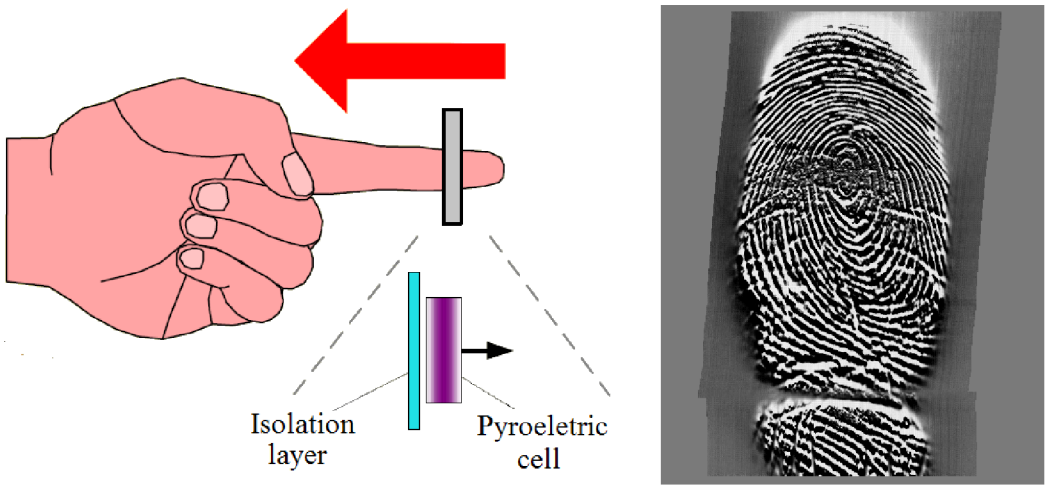


Figure 2.5: Thermal sensor principle (taken from [17]), and the example of the fingerprint acquired by a thermal sensor (taken from [4] DB3_A_dataset).

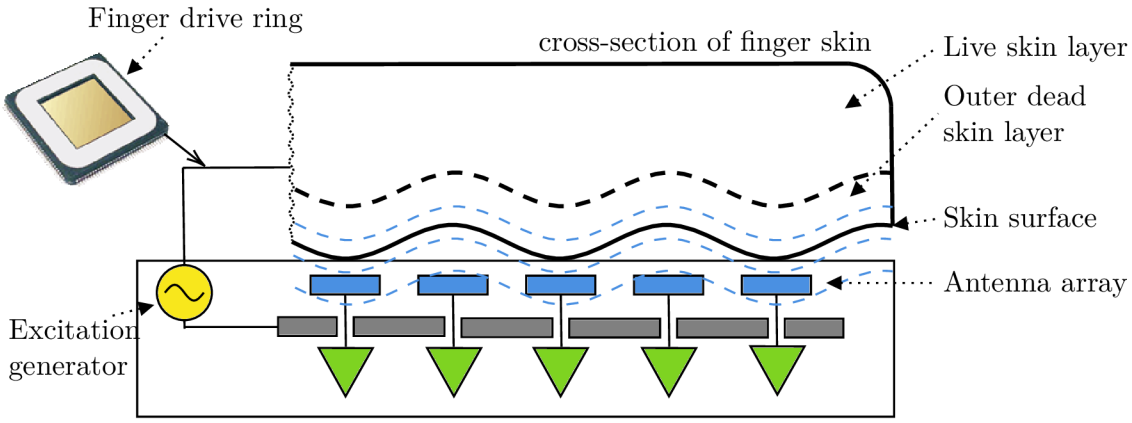


Figure 2.6: E-field sensor principle (inspired by [2]).

signal modulated by skin structure (dermis layer, because the electric field passes the upper layer of skin). This technology is resistant to fake fingers and dirt. The disadvantages of these sensors are high sensitivity to electrostatic charges and possible sensitivity to a disturbance in its RF modulation [17, 2]. An example of a fingerprint acquired by an e-field sensor can be seen in Figure 2.7.

Figure 2.7: The example of the fingerprint acquired by an e-field sensor (taken from [4] DB1_A_dataset).

2.6 Synthetic fingerprints

As the popularity of fingerprint recognition technologies is rising, many new methods need to be invented to be more resistant to impostors. These methods need thorough testing, which can be a problem because of the lack of enough fingerprints in the database. It is because acquiring such a number of fingerprints is very time and money consuming. Since collecting such a database is a very tiresome process for technicians and users, it is also easy to make a mistake. Once such a database is available, it is problematic to share it because of privacy legislation that protects personal information [17, 25].

When no large databases are available for testing, developers have to work with small databases, making it easy to make algorithms data-dependent. That led to the invention of synthetic fingerprint generators, which allow for the creation of large fingerprint databases [17].

2.6.1 Generation of synthetic fingerprints

There are several methods used for this purpose; however, most of them are based on the same principle [17]. The typical representative of tools for this purpose is SFinGe, which stands for "Synthetic Fingerprint Generator". It is the oldest and most common method for generating synthetic fingerprints. Beside creating realistically looking fingerprints, SFinGe brings several other advantages such as low costs of the fingerprint database and the possibility of producing large databases. All the mentioned factors result in easy testing and optimization of recognition algorithms [25].

The latest version of this method is 5.0, which upgraded the algorithms of previous versions and came with the new parameter, which can set up the probability of generating a low-quality fingerprint [41].

The generation process of SFinGe consists of four steps that result in the so-called master fingerprint (perfect fingerprint) [17, 25]. Other steps then make this perfect fingerprint more realistically looking. At first, the fingerprint shape is determined. The basic shape is oval and can be changed in all directions to create the required shape. In the second step, a class of fingerprint is chosen together with defining a number and positions of cores and deltas. Based on this information, a consistent direction field is generated. For the arch class, SFinGe uses a sinusoidal function whose frequency and amplitude control the arch curvature [25]. One can notice that the density of papillary lines varies over the entire area of a fingerprint. That is solved by the third step of the process when a density map is created. The density map is generated based on the positions of cores and deltas. The last step of the generation chain is a ridge pattern generating. It combines all previous steps with some initial seeds. Gabor filters then refine the image. Minutiae are generated in random places with random types. After that, the master fingerprint is generated [17, 25]. Figure 2.8 shows the example of master fingerprint generation and application of scratches to it.

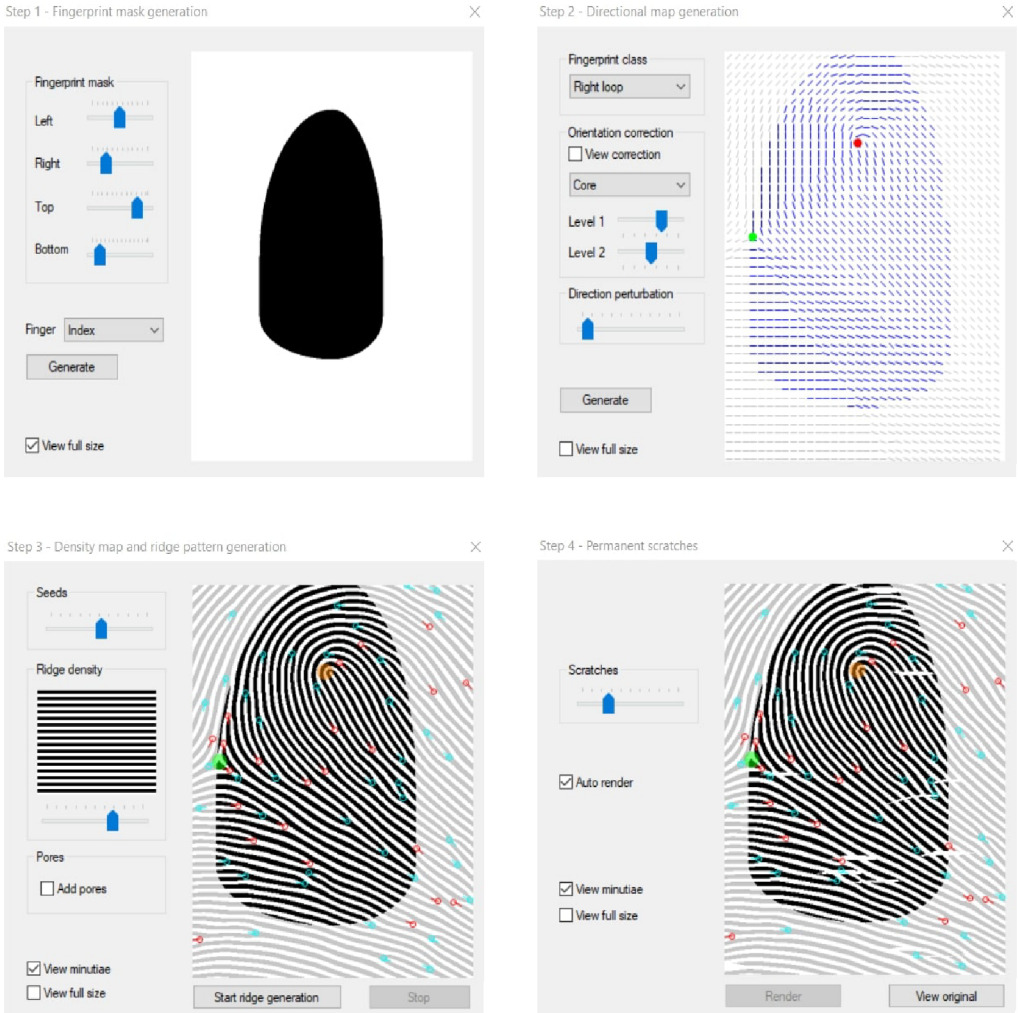


Figure 2.8: Images show four steps of the generation chain (taken from [37]).

The process of making the master fingerprint more realistically looking starts with the selection of the contact region. The ridge patterns are translated to simulate different placements of the finger on the sensor; however, the fingerprint's position and shape are not changed in this step [17].

The second step modifies ridge thickness, which simulates skin dampness and finger pressure. Wet skin or higher pressure lead to thicker ridges, and in that case, the dilatation operator is applied. Otherwise, the erosion operator is applied to simulate dry skin or low pressure [25]. This step is followed by the phase of the fingerprint non-linear distortion, which simulates skin deformation according to different finger placements on a sensor. For this distortion, the Lagrangian interpolation is used [17, 25].

The next step is noising and rendering, which simulates many small adjustments. These include non-uniform pressure of the finger, different contact of ridges with a sensor, small cuts or abrasions on the fingertip, presence of small pores within the ridges, and other noise. The process continues with translation and rotation. That simulates that the finger was not placed on the sensor precisely, so it translates or rotates the whole fingerprint.

The final step is the generation of a realistic background generated randomly from the existing set of background images transformed by mathematical methods to create new backgrounds. Different background models can be created to simulate different acquisition technologies (e.g. optical, capacitive – as mentioned in Section 2.5) [17, 25].

Chapter 3

Artificial neural networks

Artificial neural networks (ANNs) are a set of algorithms based on the operating principle of a mammalian brain. Neurons are its fundamental units of computation. In the brain, neurons are connected with synapses in more complex structures creating networks used to process data. Each neuron receives input signals from its dendrites and generates output signals along its axon. The axon branches out and connects through synapses to dendrites of other neurons [19]. People learn and improve their capacities to process data by establishing reconnections between neurons [26].

ANNs can be used for a wide range of information processing tasks. They can learn to recognize structures in a set of training data and generalize what they have learned to other datasets, which means they can handle *supervised learning* problems [26]. They also work well in analyzing large sets of high-dimensional data, where it could be challenging to determine which features are important. They can detect clusters and other structures in the input data. In this case, it is talked about *unsupervised learning* problems [26]. In many problems, some information about targets is known but is incomplete. In such cases, algorithms that combine both supervised and unsupervised learning are used. That approach is called *reinforcement learning* [26].

3.1 Transformation of biological principle into a computational model

The ANN algorithms use significantly simplified neuron models compared with real neurons, as can be seen in Figure 3.1; however, the basic principle is still the same. In the computational model, the signals that travel along the axons are called inputs (e.g., x_0, x_1). These signals interact with the dendrites of the other neuron based on the synaptic strength at that synapse. This connection with a given strength is called weight (e.g., w_0, w_1), and the operation caused by the interaction with the signal has a character of multiplication (e.g., $w_0 x_0$). The idea of reconnections mentioned above is represented by learnable weights which control the strength of the influence of one neuron on another one. Then, the signals get to the cell body, where they all get summed together with the bias b. An activation function f is applied to the final result, which decides whether the neuron should fire or not [19]. More detailed information about activation functions are provided in Section 3.3.1. A model which utilizes just a single neuron described above, is called a *perceptron* [1], which can be used as a simple binary classifier; however, such classifier works well just on linearly separable data.

As mentioned in the introduction to this chapter, neurons in a brain are interconnected into complex networks. With a single neuron, we can express binary information, which is based on whether a neuron fires or not; however, to express more than just a binary value, a structure needs to be extended by additional interconnected neurons just like in the brain. These neurons are connected in a parallel manner, as shown in Figure 3.2. This structure, called a layer of neurons, is an essential building block for multilayer networks described in Section 3.3.

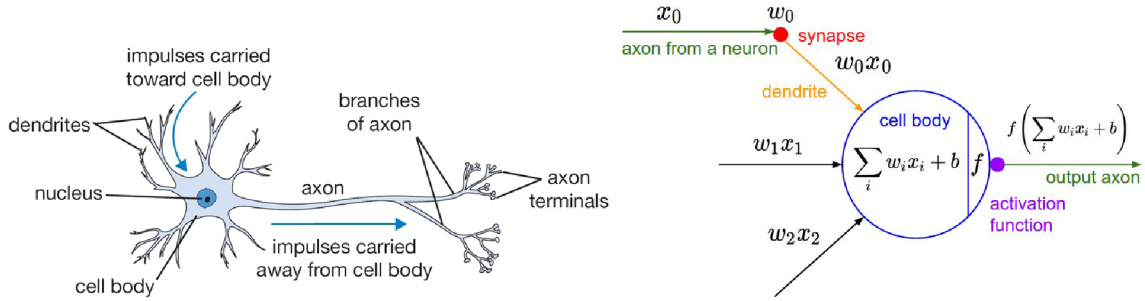


Figure 3.1: Comparison between a biological neuron (left) and its common mathematical model (right) (taken from [19]).

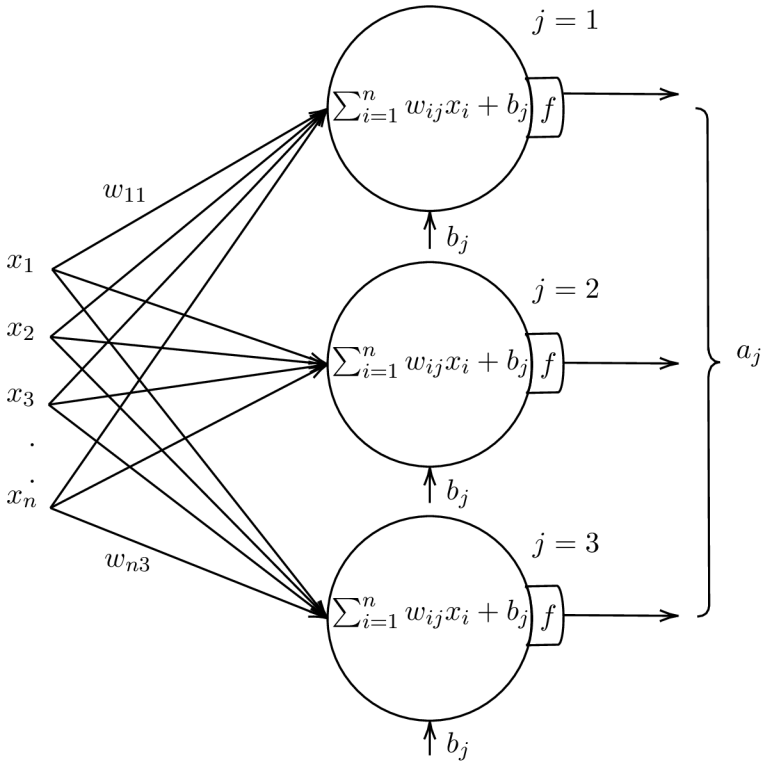


Figure 3.2: Multi-neuron neural network diagram.

3.2 Discriminative vs. generative models

Three essential steps are needed to create a synthetic fingerprint generator based on real data samples, as can be seen in Figure 3.3. At first, the algorithm needs to go through the existing fingerprint images and learn their characteristics and appearances. Specifically, it needs to learn a distribution throughout the dataset, so it knows how to represent a fingerprint image. In the end, the model needs to generate a new sample from the distribution it has learned. This process corresponds to the principle of generative models, which is a subclass of machine learning algorithms. This section describes the principle of both generative models and discriminative models, which are their opposite.

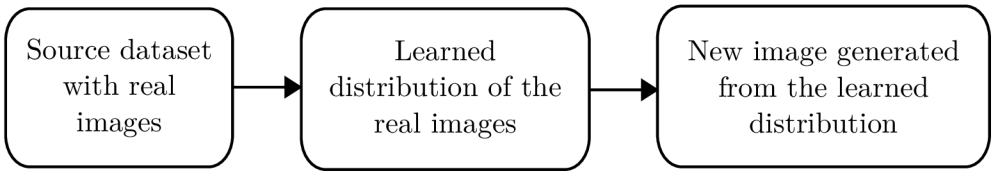


Figure 3.3: The basic principle of a generative model.

The goal of supervised learning is to learn a mapping function $x \rightarrow y$, where x represents a piece of data, and y represents a target variable (label) [23]. Examples of supervised learning tasks can be classification, regression, or semantic segmentation. Discriminative models are models for supervised learning. These models estimate a posterior probability distribution $p(y|x)$ [13]. When x is an input image, and y is a label of a class, then the distribution reveals the extent to which the model calculates the image to be representative of a particular class.

On the contrary, unsupervised learning aims to learn some underlying structure of data x even when there are no labels available [23]. Examples of unsupervised learning are clustering, density estimation, or dimensionality reduction. Generative models are models primarily for unsupervised learning but can also be used in a supervised setting. These models address a density estimation, which is one of the core problems in unsupervised learning. Generally, their goal is to learn a distribution $p_{model}(x)$, which is as similar as possible to a distribution of training data $p_{data}(x)$ [23].

It is useful to demonstrate a difference between the approach of generative and discriminative models on a task that they both can be used for – classification. As mentioned above, the goal of a discriminative classifier is to learn a posterior distribution $p(y|x)$, while the goal of a generative classifier is to learn a joint probability $p(y, x)$. This joint probability can be learned directly, or by computing it using a chain rule as $p(y, x) = p(y|x) \cdot p(x)$. That shows the relation between a posterior probability learned by discriminative models and a joint probability learned by generative models. Generative models need to learn a density function $p(x)$ to be able to represent the input data well.

In comparison, generative models have a more difficult task, since their goal is significantly more complex. As can be seen in Figure 3.4, discriminative models learn boundaries between data classes, while generative models learn the distribution of individual classes. However, the advantage in learning the distribution of training data is that generative models have significant additional value at generating new samples similar to training ones.

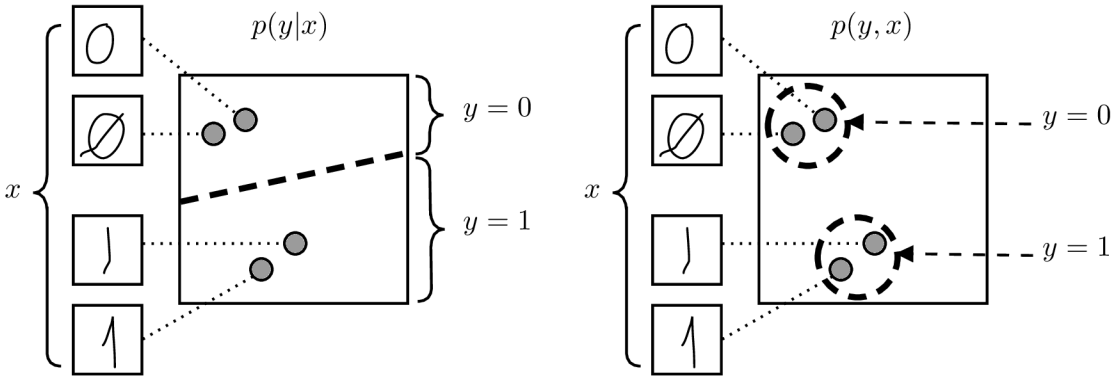


Figure 3.4: Example of discriminative and generative models' goals in classification task represented in the graph (inspired by [13]).

3.3 Deep feedforward networks

When a perceptron model, mentioned in Section 3.1, is described as a layered structure, there are two layers - the input layer and the output layer. The input layer transmits the data to the output layer, which is a computational one. Multilayer networks improve the architecture by at least one additional computational layer between input and output layers, referred to as a hidden layer. Based on the increasingly used multilayer structure in neural networks, they are known as deep learning models. The specific kind of architecture of multilayer networks is a feedforward network, which means that outputs of neurons from one layer are fed as inputs to the successive layer. Therefore, the whole structure represents an acyclic graph [1, 24]. Since deep feedforward networks (DFNs), also known as multilayer perceptrons, are essential models of deep learning, it is convenient to describe all neural networks' fundamental components in this section.

3.3.1 Activation function

An activation function is a mathematical function, that allows to decide whether a neuron is activated or not. Below is the basic formula for a single neuron from Figure 3.1.

$$a = \sum_i w_i x_i + b \tag{3.1}$$

It is clear that $a \in \mathbb{R}$, so there is no information about value bounds. That makes it impossible to decide whether a neuron should fire (be activated) or not. For this purpose, the activation function $f(a)$ transforms the value a into a given range, making it possible to decide whether the neuron should be considered activated. All of the activation functions described in this section are based on the information from [10] and [24].

The most straightforward activation function is a step function. This function works well with a perceptron model as a binary classifier. However, it cannot be used for more complex tasks, where the structure consists of multiple neurons.

$$f_{step}(x) = \begin{cases} 1 & \text{for } x > threshold, \\ 0 & \text{otherwise.} \end{cases} \tag{3.2}$$

A binary output problem can be solved by a linear activation function, which scales its input by a constant c. On the other hand, it brings other problems. Using this function,

it is not possible to use backpropagation (see Section 3.3.3) to train the model. Since backpropagation uses gradient descent, and the derivative of a linear function is a constant, the gradient has no relation to the input. Also, the model loses the strength of multiple layers because they all collapse into a single layer. That is because a linear combination of linear functions is still a linear function. A range of output values causes one more problem. As already mentioned, it is convenient to transform values into a specific range; however, with a linear activation function, output values can still be in a range $(-\infty; \infty)$. It is possible to utilize these values, for example, for classification using a function $max()$ or $softmax()$, and make a final decision based on the result. However, it does not allow generating probabilistic scores from neurons.

$$f_{linear}(x) = cx \tag{3.3}$$

It is necessary to use non-linear activation functions in deep learning models to avoid the problems mentioned above. One of the well-known is a sigmoid function. It squashes the output values into range $(0; 1)$, which allows generating probabilistic scores from neurons. Also, the gradient is steep near the origin and saturates as going along the x-axis. That helps in classification tasks because the function tries to keep output values close to either zero or one. The main problem with this activation function is its small gradient as getting further from the origin. That can cause a vanishing gradient problem, which leads to slow learning of the network, or eventually, a stop of learning.

$$f_{sigmoid}(x) = \frac{1}{1 + e^{-x}} \tag{3.4}$$

Similar to sigmoid function is the Tanh function, which squashes the output values into range $(-1; 1)$, instead of $(0; 1)$. It is a scaled sigmoid function, which has a steeper gradient. However, the main problem with vanishing gradients remains the same.

$$f_{tanh}(x) = \frac{2}{1 + e^{-2x}} - 1 \tag{3.5}$$

Currently, the most widely used activation function is the ReLu function. The main advantage against functions above is that ReLu outputs 0 for all negative input values and therefore deactivates neurons that produce negative values. Most of the neurons fire in an analog way with functions like sigmoid or Tanh, so most of the activations need to be processed to contribute to the network output. ReLu helps to make these activations more sparse, resulting in a lighter network. Additionally, ReLu is significantly less computationally expensive because it involves simpler mathematical operations then sigmoid or Tanh.

$$f_{relu}(x) = max(0, x) \tag{3.6}$$

The problem of ReLu function is the zero output for values in range $(-\infty; 0)$. Since a gradient in that range is also a zero, weights are not adjusted for these activations during backpropagation. That leads to a dying ReLu problem when a part of the network can become passive. A modified version of this function was proposed to address this problem, which reduces the effect of neurons by a factor c in negative range instead of deactivating them. This modified version is called a leaky ReLu.

$$f_{leaky\_relu}(x) = \begin{cases} x & \text{for } x \geq 0, \\ cx & \text{for } x < 0. \end{cases} \tag{3.7}$$

3.3.2 Loss function

Once a model predicts the output value y during training, it needs to be evaluated against the expected output value $\overline{y}$. For this purpose, a loss function is used, which defines the error of the model. In other words, it reflects how far from the correct output, the generated output is. The result is then used to tune the network parameters during backpropagation (see Section 3.3.3). The information about loss functions provided in this section was obtained from [24].

Loss functions are generally separated into two categories - regression loss and classification loss. In regression tasks, both the expected output values and predicted output values are direct. Therefore, it allows calculating the loss based on the difference between these two values.

The most straightforward loss function would then be a direct difference between the expected and predicted outputs, as shown below. However, this allows positive as well as negative results, which is undesirable.

$$L(w) = \overline{y} - y \tag{3.8}$$

There is a couple of regression loss functions; however, the two most common similar loss functions are used to avoid the problem with a sign of the result - Mean squared error (MSE) and Mean absolute error (MAE). MSE loss function squares the difference between predicted and expected values, which makes all values positive, and then computes the mean to normalize the result. For n predicted values y and corresponding expected values $\overline{y}$, the MSE is defined as shown below.

$$E = \frac{\sum_{i=1}^{n} (\overline{y_i} - y)^2}{n} \tag{3.9}$$

The problem with MSE is that it is prone to outliers in the data. Any sample, which outputs value far from the expected value, contributes significantly to the loss. When many outlying data are present in the dataset, MSE causes a problem. That can be solved by the MAE loss function, which computes the absolute value of error instead of squaring it. This approach is not as sensitive to outliers as MSE; however, calculating absolute values makes the loss function non-differentiable, which is a problem during backpropagation.

Classification tasks do not predict direct values as regression tasks. Instead, they predict a discrete class label, which is generally a positive integer value. Since both the expected value and predicted value are integers from a set of discrete values, computing the difference between those two numbers does not provide any useful information about the loss. Therefore, classification loss functions are developed over probability distributions.

Cross-entropy is one of the most common loss functions for classification models that output probability values between 0 and 1. It compares the probability distribution of prediction with the expected labels. Cross-entropy loss is minimal when these two distributions are the same, and increases as they diverge from each other. The formula is defined for the expected label $\overline{y}$ and predicted label y as:

$$H(\overline{y}, y) = -\sum_{i=1}^{n} (y_i \log \overline{y_i}) \tag{3.10}$$

3.3.3 Backpropagation process

In order to effectively train a neural network model, there is a need to update its weights (in a biological analogy – a need for establishing reconnections between neurons), which reflects how a neural network learns. Once a neural network is constructed with its initial weights, a forward pass is performed. That means inputs from a training set are fed into the network and passed through until the network eventually generates an output. After that, the loss function is computed, which reflects the error of the model as described in Section 3.3.2. Once the information about the error is known, it needs to be reflected in the network structure. That is the point where a backpropagation algorithm takes place. If a function is differentiated, we get the gradient of that function. The gradient represents the direction along which the function increases/decreases the most. Backpropagation calculates partial derivatives, going back from the error function to a specific neuron and its weight. The backpropagation results in a set of weights that minimize the error function [28].

This process could be repeated for every sample in the training set; however, this would be ineffective. Typically, a couple of samples are grouped in one batch. The whole batch is then passed through the network, and the backpropagation is performed on the aggregated result. The batch size and the number of batches, called iterations, are two hyperparameters that can be optimized [28]. Once an entire dataset is passed through a neural network, it is referred to as one epoch. So, the number of iterations in one epoch can be computed by the equation below:

$$iterations_{epoch} = \frac{size\ of\ a\ training\ dataset}{batch\ size} \tag{3.11}$$

3.3.4 Adam optimizer

In practice, the backpropagation algorithm is used by more sophisticated algorithms that use the backpropagation for computing the gradient of the error function and they control the update of weights. One of the common optimization algorithms for this purpose is called Adam [8]. The size of the step in the direction of the decreasing gradient of the error function is defined by the hyperparameter called a learning rate. The learning rate has a significant impact on neural network performance and it is hard to set. Adam is one of the algorithms that automatically adapt the learning rate throughout learning. It uses a momentum, which accumulates exponentially decaying moving average of past gradients and continues to move in their direction, which optimizes the learning process of neural networks [11].

3.3.5 Neural network hyperparameters

In the last section, a term hyperparameter was mentioned. This section gives a closer look at its meaning. Model parameters are internal properties of training data that are learned during training. The objective of the network training is to learn the values of the model parameters. Hyperparameters, on the other hand, are external parameters set by the programmer. Different values of hyperparameters can have a significant impact on network performance [28].

Hyperparameters that are related to the network structure can be, for example, number of hidden layers in the network architecture, an activation function, that determines the output of each element in the neural network, initialization values of weights, or a dropout (one of the methods to avoid network underfitting). Other hyperparameters related to the

training algorithm are, for example, a learning rate, a batch size, number of iterations, or an optimizer algorithm [28].

Optimizing the hyperparameters of a neural network model means retraining the network using each set of hyperparameters and evaluating the results. There are a few more or less sophisticated methods to do this. Manual search requires just testing of hyperparameters that the operator chooses. Unless the operator is experienced, this method can be a dead end. A more systematic approach brings a method called grid search, which involves systematically testing different hyperparameters' values and retraining the model for each combination [28]. However, James Bergstra's and Yoshua Bengio's paper [3] showed that it is more efficient to use random hyperparameter values than using the manual search or grid search. The last commonly used optimization method is Bayesian optimization [34]. The idea is to train the model with different hyperparameters values and observe the shape of the function generated by these values. The method then predicts the best possible hyperparameters values, which provide higher accuracy than a random search [28].

3.4 Convolutional neural networks

Since the great success of convolutional neural network (CNN) architecture in 2012 [21], the concept of CNNs became highly used primarily for computer vision tasks (e.g., image classification, face recognition, image processing in robots and autonomous vehicles). CNN scans an image one area at a time, identifies and extracts essential features that are used for image classification [18].

Plain neural network model, in which layers are fully connected (meaning that all neurons in one layer are connected with all neurons in the following layer), is inefficient when it should process extensive high-dimensional data, such as images or videos. Since an image with hundreds of pixels and three color channels (RGB) results in millions of model parameters, there is a high chance of overfitting. To limit the number of parameters, CNN uses a structure in which each set of neurons analyzes just a part of the image. The general form of this structure can be written as $height \times width \times depth$, where depth relies on the color channel of the image. When working with grayscale images, this structure is a single matrix with values in a range from 0 (black) to 255 (white) for each pixel. In a case of the RGB image, the structure would contain three matrices, each having values in a range from 0 to 255 [20].

3.4.1 CNN architecture

Three main layers of a network are used to build the CNN architecture – convolutional layer, pooling layer, and fully-connected layer. The fully connected layer works as a classical neural network. It is usually at the end of the CNN and outputs the vector of probabilities as in the standard classification task. The process that makes CNNs different and very powerful happens in the unique architecture between the input and the output layer.

The convolutional layer's parameters are a set of filters that map each of the neurons in the convolutional layer on a spatially small area of the input volume. The basic idea is indicated in Figure 3.5. That reduces the total number of model parameters needed. The size of the filter is a hyperparameter called the receptive field of the neuron. During the forward pass, the filter moves across the input height and width axis, and products are computed using a convolution operation. Three hyperparameters are important for the size of the convolutional layer's output – depth, stride, and zero-padding. The depth

hyperparameter corresponds to the number of filters that a developer wants to use. Since each neuron learns to find a different feature in the input, this hyperparameter is important to optimize how detailed the convolutional layer should be. The value of stride defines how the filter is shifted. The stride of value one means that the filter moves one pixel at a time. When the value of stride is increased, the convolutional layer will produce smaller outputs spatially. Zero-padding is used to pad the input with zeros around. Its size is a hyperparameter. In the CNNs, the zero-padding is usually used to ensure that the input and output volume will have the same size [20].

With the mentioned values, the size of the output volume can be computed using the following formula:

$$\frac{(W - F + 2P)}{S} + 1, \tag{3.12}$$

where W is the input volume size, F is the receptive field size, S is the value of stride, and P is the amount of zero-padding used [20].

The pooling layer is commonly inserted between convolutional layers in the CNN architecture to reduce the spatial size of the input, and together with that, the number of model's parameters. It works on each depth slice of the input and resizes it by applying some operation on values within the filter. The most common operations that the filter applies are max-pooling, average pooling, or L2-norm pooling. The average pooling is not commonly used nowadays since, in practice, the max-pooling proved to work better. The principle of max-pooling is shown in Figure 3.6. The usage of pooling layers turned out to be a little bit controversial since there are models that showed that pooling layers are unnecessary, and to reduce the size of input data using higher stride in the convolutional layer works well [20].

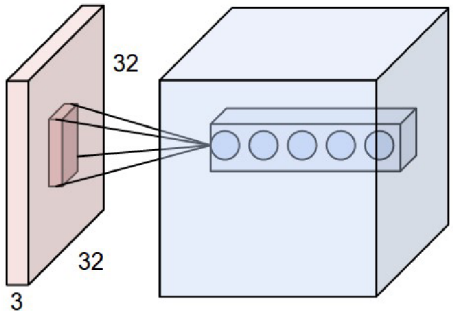


Figure 3.5: Example of neurons in the convolutional layer connected to a local region of the input data. These neurons that are connected with the same regions are referred to as depth column (taken from [20]).

3.5 Generative adversarial networks

Models described up to this section (DFNs, CNNs) are the common types of discriminative models. However, based on the information in Section 3.2, it is clear that the generative model is needed for the purpose of this work, so that new data samples can be generated. In 2014, one, called generative adversarial networks (GANs) [12], was proposed, which became a state-of-the-art generative model for the following years.

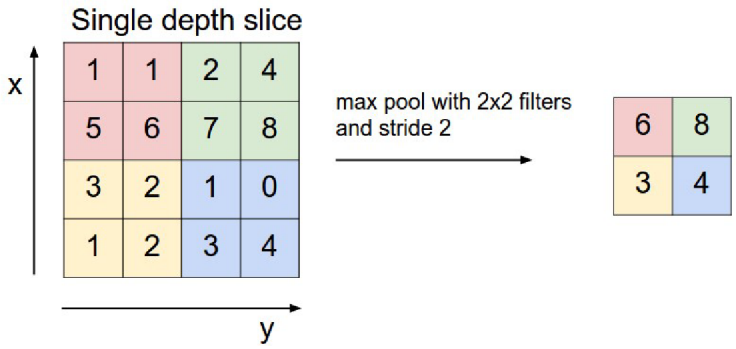


Figure 3.6: Example of how the pooling layer applies a filter using the max operation. (taken from [20]).

GANs belong to the class of direct implicit density generative models. Therefore, their goal is to sample from the probability density function $p_{model}(x)$ without explicitly defining it [23].

Even though generative adversarial networks are generative models because their goal is to learn a data distribution of the training dataset, they also use a discriminative model in their architecture. The generative model is referred to as a *generator* and discriminative model as a *discriminator*. The goal of a generator is to learn how to produce data as similar to real images as possible. The goal of a discriminator is to evaluate its input data and decide whether they look real or fake. These two models are adversaries in the zero-sum minimax game. This game has a nash-equilibrium when a generator learns the training data distribution, and a discriminator is not able to distinguish whether the input sample is real or fake, which means that $p_{real}(x) = \frac{1}{2}$ and $p_{fake}(x) = \frac{1}{2}$, where x is the input sample. The basic structure of generative adversarial networks is shown in Figure 3.7.

$$\min_G \max_D V(D,G) = \mathbb{E}_{x\sim p_{data}(x)}[\log D(x)] + \mathbb{E}_{z\sim p_z(z)}[\log(1 - D(G(z)))] \tag{3.13}$$

The equation above shows a value function of the minimax game between a generator and a discriminator [12]. The mapping of the input noise features defined by prior $p(z)$ into a data space is given by a differentiable function $G(z, \theta_g)$ represented by a generative model with parameters θ_g. The discriminative model with parameters θ_d defined as $D(x, \theta_d)$ outputs a single scalar value representing the probability that data x came from the p_{data} rather than p_g. The discriminator is trained to maximize the probability of assigning correct labels to both real samples and samples coming from the generator. The generator is trained to minimize $\log(1 - D(G(z)))$. That works well theoretically; however, in practice, training a generator to maximize $\log D(G(z))$ is preferred instead of minimizing $\log(1 - D(G(z)))$, which provides significantly stronger gradients in the early stage of training and therefore helps generator to learn well [12].

The reason why GANs are so popular in processing high-dimensional data is the idea behind them. The problem is that there is no direct way to sample from a complex, high-dimensional distribution of training data. This model allows to sample from a simple distribution and learn transformation to the training distribution using a neural network [23].

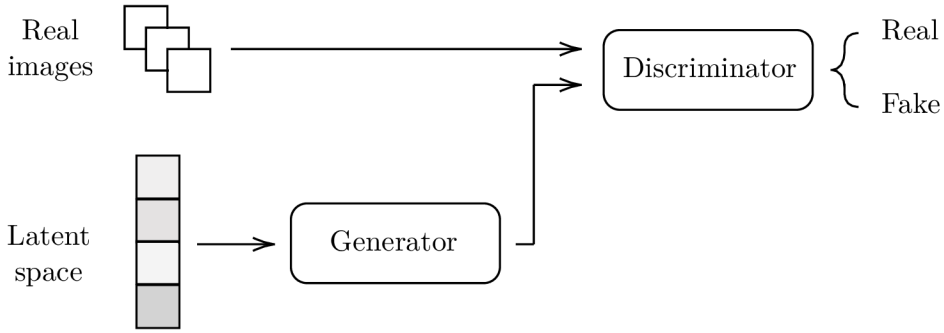


Figure 3.7: Basic schema of generative adversarial networks.

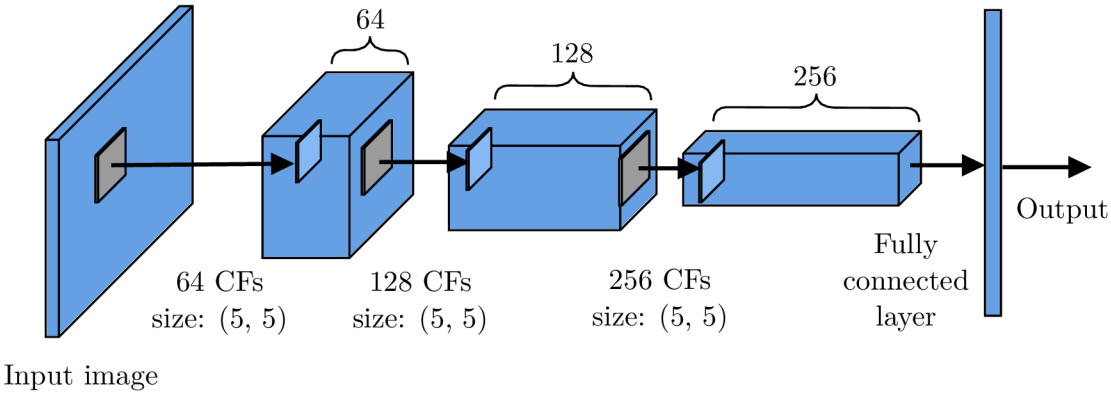
3.6 Deep convolutional generative adversarial networks

The original model of GAN proposed by [12] used a multilayer perceptron for both a discriminator and a generator. However, as mentioned in Section 3.4, to process high-dimensional data, it is convenient to use a convolutional neural network, which significantly reduces the number of parameters of the model. In paper [31], a modified version of GAN was proposed, which uses convolutional layers for a discriminator, and fractionally-strided convolutional layers for a generator. This model, called a deep convolutional GAN (DCGAN), proved to work well and became one of the most popular generative models for high-dimensional data.

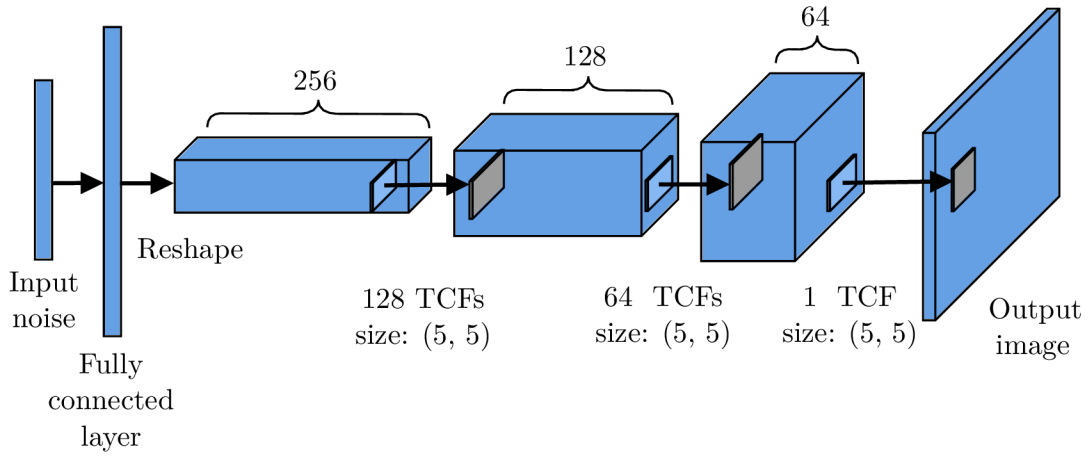
3.6.1 DCGAN architecture

As already mentioned, the discriminator network in DCGAN uses convolutional layers that downsample input data. The last convolutional layer is flattened and then fed into a single sigmoid output. The generator network then reshapes an input noise first and then uses fractionally-strided convolutional layers, also called transposed convolutional layers. These layers upsample input data into the desired shape, which needs to be the same as the shape of real samples. The example of generator and discriminator networks is shown in Figure 3.8.

Alongside the updated architecture of the original GAN model using CNN architectures, paper [31] also proposed several changes that lead to a stable training of a DCGAN. These changes include replacing any pooling layers with strided convolutional and fractionally-strided convolutional layers, which allows discriminator and generator networks to learn their own spatial downsampling and upsampling, respectively. The other proposed change is to apply a batch normalization to all layers, except the input layer of a discriminator, and the output layer of a generator, which stabilizes learning by normalizing the input to each unit to have zero mean and unit variance. The third change is removing any fully-connected hidden layers since they can reduce convergence speed. The last proposed change deals with activation functions. The recommended activation for a discriminator is Leaky ReLu for all layers. For a generator, using ReLu activation is recommended for all layers except the last one, which uses Tanh. All activation functions are described in Section 3.3.1.



(a) Example of a discriminator network (CF = convolutional filter).



(b) Example of a generator network (TCF = transposed convolutional filter).

Figure 3.8: Example of networks in DCGAN model (inspired by [24]).

Chapter 4

Solution design

Designing a synthetic fingerprint generator requires two main steps. The first one is the acquirement of a sufficient dataset. Since the model is designed with respect to the input data, this step is crucial, and any later change can affect the model architecture. In Section 4.1 are described two popular fingerprint datasets for research purposes. Both of them provide a significant amount of data; however, the SOCOFing dataset was chosen for the purpose of this work due to its higher complexity. The second step is then the design of a generator model. In Section 4.2 is presented the proposed model based on a DCGAN architecture together with methods implemented to improve its performance.

4.1 Datasets

As mentioned in Section 2.6, one of the main problems with biometrics databases is privacy protection. Therefore, it is not easy to share a database once it is acquired. In order to create a synthetic fingerprint generator, it is essential to get a sufficient database that our model can train on. Luckily, for non-commercial purposes, there are still a few accessible databases with anonymized data, usually owned by academic institutions. This section presents two popular fingerprint databases meant for research purposes – SOCOFing [35, 36] and FVC2006 [4].

4.1.1 FVC2006 dataset

Even though this dataset was initially designed for a fingerprint verification competition, it became a popular source of fingerprint images for research purposes. Also, the original paper [27] used the FVC2006 dataset in their solution. FVC team grants access to four distinct subsets DB1, DB2, DB3, and DB4, each of them acquired by a different method. Each subset is divided into two parts: A and B. Part A contains the first 1,680 fingerprint images from 140 fingers. Part B contains the other 120 fingerprint images from 10 fingers. In total, there are 1,800 fingerprints acquired by four different methods. The example samples from the FVC2006 dataset can be seen in Figure 4.1.

The image format is BMP, and the images are in 256 shades of gray. Image resolution varies depending on the subset. Table 4.1 shows a resolution, an image size and a sensor type for each subset.

Figure 4.1: Examples of FVC2006 datasets in the following order: DB1, DB2, DB3, and DB4.

Table 4.1: Properties of the FVC2006 datasets (taken from [40] and modified).

Subset	Sensor Type	Image Size	Resolution
DB1	Electric field sensor	96×96	250 dpi
DB2	Optical sensor	400×560	569 dpi
DB3	Thermal sweeping sensor	400×500	500 dpi
DB4	SFinGe v3.0	288×384	about 500 dpi

4.1.2 SOCOFing dataset

Sokoto Coventry Fingerprint Dataset (SOCOFing) is designed specifically for academic research purposes. An extensive dataset of fingerprint images also contains useful information about gender, hand, and finger name for each image. The dataset contains two parts. In the first part, 6,000 fingerprints acquired from 600 African individuals are located. The second part provides over 49,000 altered fingerprint images in total. The altered images are further separated into three parts by the STRANGE framework's level of alteration - easy, medium, and hard. STRANGE is a framework for the generation of realistic alterations on fingerprint images. In the SOCOFing dataset, samples are altered by obliteration, central rotation, and z-cut. Information about the applied alteration is incorporated in the image title. The SOCOFing dataset was chosen for the purpose of this work due to its complexity.

All images have the same resolution of 1×96×103 (gray channel × width × height), and all real images were acquired using an optical sensor. Table 4.2 contains information about the number of images in each part of the dataset. In Figure 4.2, different levels of alteration are shown. The first column contains a real image example, and each successive column contains a higher level of an alteration.

Table 4.2: Number of fingerprint images in each part of the SOCOFing dataset.

Subset		Number of images
Altered	**Easy**	17,934
	Medium	17,067
	Hard	14,272
Real		6,000

Figure 4.2: Examples of altered images from the SOCOFing dataset. Each line includes one type of alteration - central rotation, obliteration, and z-cut in the respective order.

4.1.3 Data augmentation

The common method to reduce overfitting of convolutional neural networks is applying data augmentation. That means applying transformations on the original data to increase the generalizability of the model, especially when working with small datasets. Since the proposed DCGAN model contains a discriminator, which is a convolutional neural network, and the real part of the SOCOFing dataset contains just about six thousand images, it is a convenient use case of data augmentation. The altered images from the SOCOFing dataset were not used during training due to their quite rough modifications of the original fingerprints. Instead, the custom data augmentation was implemented using Keras framework, which is further described in Section 5.4.1. Examples of augmented samples that were used to train the proposed solution are shown in Figure 4.3.

Figure 4.3: Examples of augmented samples from the SOCOFing dataset.

4.2 Proposed model

DCGAN model, which was described in Section 3.6, was chosen for this work. It also proved to work well in paper [27], where it was used for the same purpose as a synthetic fingerprint generator. The basic DCGAN architecture that was used for this work is shown in Figure 4.4. The discriminator consists of four convolutional layers. Each of them has a kernel size of 5×5, and stride is set to value two. That means that each layer downsamples data to half of their size. Data get flattened at the end of the network, and then they are fed into a dense layer with sigmoid activation function, which generates a single scalar value representing how much the discriminator believes that given data came from the training dataset. The generator contains five fractionally-strided convolutional layers. Each of these layers upsamples data to double their size spatially. A dense layer precedes these layers. Output values from this dense layer are simply reshaped into the selected input shape of the network. The last layer in the generator network is a convolutional layer, which changes the data dimension to the required number of channels. In this case, where images are in greyscale, the number of channels is one.

In Section 3.6 were mentioned several tips proposed by [31] for a stable training of DCGANs. However, the research of techniques for stable training models based on GAN architecture is still an active field, and many methods were proposed to improve and stabilize their performance. Some of them were implemented in this work and are described further in this section. In Section 5.3, changes to the proposed DCGAN model after implementation of these methods are described.

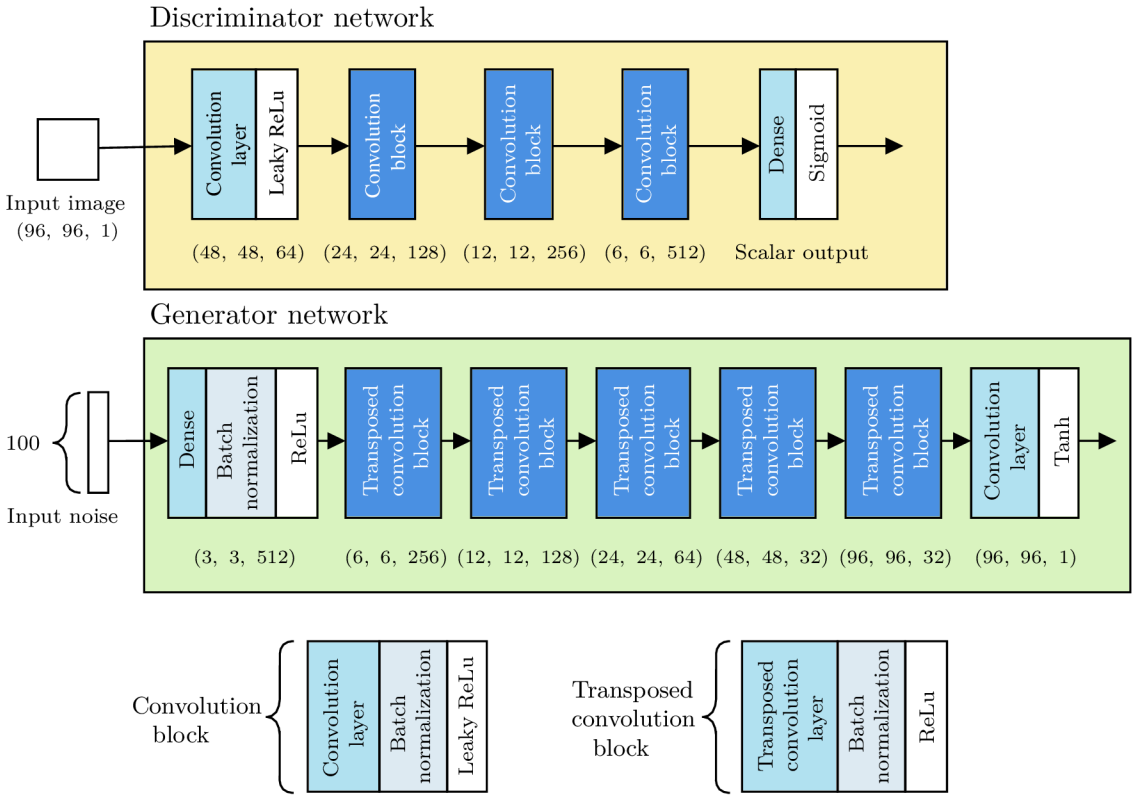


Figure 4.4: The proposed architecture of discriminator (top) and generator (bottom) models.

4.2.1 Label smoothing and noisy labels

When training the discriminator model, real images are generally represented by label '1' and fake images by label '0'. However, these hard labels are good to be replaced by smooth labels, which proved to work well against discriminator being overconfident about its predictions. That can have a regularizing effect when training GANs. In [32] is recommended using one-sided label smoothing when only positive labels are smoothed, and negative labels are kept at value '0'. Given the parameter α, values of positive labels were randomly transformed into range $\langle 1 - \alpha, 1 \rangle$.

Another method related to labels is using noisy labels [5]. That means flipping some real and fake labels when training the discriminator, which can be represented as $y_{new} = 1 - y_{old}$. That introduces an error to those labels and helps to reduce discriminator's overconfidence. In this work was used a 5 % probability of flipping the label during training.

4.2.2 Minibatch discrimination

To address the mode collapse, which is one of the main failure modes in training GANs, [32] proposed a method called minibatch discrimination. The mode collapse is a state where the generator produces the same output for different input data. The problem is that the discriminator processes each sample separately and therefore has no way to inform the generator to produce more distinctive outputs. The minibatch discrimination method is based on a simple principle allowing the discriminator to look at multiple samples simultaneously. However, it is focused primarily on similar samples within the batch – the similarity of generated samples increases, when the mode starts to collapse. The similarity $o(x_i)$ is computed between the image x_i and all other images in the same batch. This similarity is then appended to one of the intermediate layers of the discriminator, and it can use this score to detect generated samples and penalize the generator. The example of the discriminator architecture together with extended dense layer by the similarity at the end of the network is shown in Figure 4.5.

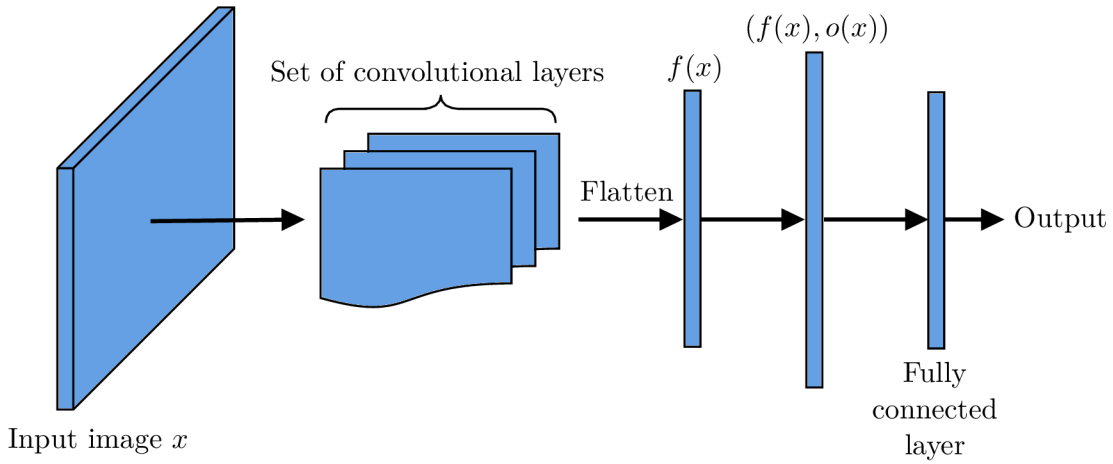


Figure 4.5: Example of the discriminator network with similarity appended to the intermediate dense layer.

4.2.3 Spectral normalization

GANs have a general problem with the unstable training process. One of the main challenges is controlling the performance of a discriminator. When the support of the learned distribution p_{model} and the support of the real distribution p_{data} are disjoint, the discriminator, which can perfectly distinguish these distributions, exists. Such discriminator then leads to a stop of the generator's training, because its derivative with respect to the input turns out to be 0. That leads to a need for a restriction on the choice of a discriminator. In 2018 was proposed a method for weight normalization called spectral normalization [29], which stabilizes the training of the discriminator.

The method restricts the choice of the discriminator to the set of Lipschitz continuous functions, assuring the boundedness of those functions [29]. In practice, the method computes the spectral norm $\sigma(W^l)$ for each layer l, which is the largest singular value of the layer's weights W. The spectral norm $\sigma(W)$ could be computed using a singular value decomposition; however, that showed to be computationally heavy. Therefore, the method uses the power iteration method [29] to estimate $\sigma(W)$, which results in a short computational time compared with the overall computational cost of the GAN training.

4.2.4 Experience replay

The mode collapse problem can also be addressed by mitigating the opportunity of the discriminator to overfit for a particular time instance of data batches. One generated data sample is preserved in each training step. When it reaches the given number of steps, the discriminator is fed by recently generated images together with the current batch. This method is based on a stability trick proposed for reinforcement learning problems in [30].

Chapter 5

Implementation and results

This chapter provides an overview of the implementation details of the proposed solution and its results evaluation. The solution was implemented in Python language as it is one of the most popular languages for creating deep learning models. Python provides simplicity by its own syntax, but also by supporting many deep learning platforms specifically designed to make the development of deep learning models even more comfortable. In Section 5.1, some of the most popular deep learning platforms are discussed. Section 5.2 provides details about the hyperparameters of the proposed model. In Section 5.3, the models that were used for results evaluation are discussed.

5.1 Python deep learning platforms

Python supports many deep learning frameworks and libraries that make it significantly more comfortable to create deep learning models. Some of the popular ones are described in this section. Since most of the described libraries were initially released after 2015, it is clear that Python's deep learning environment grows very fast. Keras framework was used to implement the proposed solution because it provides a high-level easy-to-learn API with additional benefits: real-time data augmentation, image preprocessing features, pre-trained models, and many more.

5.1.1 Theano

Theano [38] is a Python library that provides efficient ways of working with multi-dimensional arrays. Since deep learning problems involve large amounts of data, this feature is crucial. Theano also works as an optimizing compiler, it can compile parts of an expression graph into CPU or GPU instructions, which improves computational performance. Theano can also find some of the numerically unstable expressions and compute them using more stable algorithms. Additionally, it supports an efficient symbolic differentiation.

This library provides many advantages, most of all its flexibility. On the other hand, since Theano is a lower-level API, it requires good knowledge to write effective code. Therefore, Theano is used to power some of the other deep learning frameworks that work on a higher level, such as Keras.

5.1.2 TensorFlow

TensorFlow is an open-source library that provides multiple levels of abstraction. In cooperation with Keras API, it allows a high-level approach to developing deep learning models. However, it still offers a lot of flexibility. TensorFlow's basic data structures are tensors that are multi-dimensional arrays. Therefore, TensorFlow effectively allows us to work with multi-dimensional arrays, which provides excellent support for the development of deep learning models. The same code can be computed on either CPU or GPU, which can accelerate the computational process. Computations in TensorFlow are described by computational graphs, where each edge represents data (tensor), and each node represents a mathematical operation performed on the input data. The computational graph is built in advance of running the computational process, allowing, for example, using placeholders in code, which represent data that are not known before running the model and can be added during runtime from external resources. Since Google LLC backs TensorFlow, there is a large community of developers that share their knowledge and experience, which is a significant benefit of using this library. TensorFlow provides mainly computationally well-performing platform; however, it does not provide many additional features that would help with the whole process, such as data preprocessing and others. It is more convenient to use the Keras high-level framework with TensorFlow as its backend.

5.1.3 PyTorch

With the initial release in 2016, PyTorch [39] is the youngest deep learning framework discussed in this section. Many of its features are similar to TensorFlow and Theano. For example, it works with tensors that can be computed on both CPU and GPU, which can significantly increase the computational performance. Another similar feature, yet the one that makes the most significant difference, is building a computational graph. PyTorch is designed to handle dynamic computational graphs, which is not possible in either TensorFlow or Theano platforms. It provides features of both high-level and low-level APIs, which provides flexibility and simplicity at the same time. Keras was chosen for the implementation of the proposed solution because there is still a higher number of relevant resources that use TensorFlow or Keras for development in the time of creating this work.

5.1.4 Keras

The Keras Python library is a popular deep learning API, which was developed to run mainly on top of the TensorFlow and Theano machine learning platforms. The Keras API provides two core data structures – layers and models, which makes it fast and easy to create and train the proposed solution. There are two approaches to create models in Keras - sequential [6] or functional [7]. The sequential model is a linear stack of layers, which works well for simple models. More complex structures require using the Keras functional API. For example, it allows defining models with multiple inputs or outputs, models with shared layers, and much more.

5.2 Hyperparameters of the proposed model

The proposed model was trained for 300 epochs on an Nvidia Tesla T4 GPU, with a batch size of 128. Weights were initialized from a zero-centered Gaussian distribution with a

standard deviation of 0.02, as recommended in [31]. The input noise to the generator network was sampled from 100-dimensional Gaussian distribution with zero mean and unit variance. ADAM optimizer was used to optimize the loss function with the learning rate of 0.0002 and the momentum of 0.5 [31]. The factor α of label smoothing was set to 0.1 with a 5 % probability of flipping the label during training. Experience replay happens after each 32 training steps (batches). For minibatch discrimination, 100 discrimination kernels were used, which results in the extension of the flattening layer's output's dimensionality by 100. The similarity of samples is computed in the 30-dimensional space.

5.3 Implementation of the proposed model

As mentioned in Section 5.1, the proposed solution was implemented using Python's deep learning framework called Keras. TensorFlow version 2.1.0 was used as its backend because it proved to be the most stable version in the time of creating this thesis. For the implementation of the purposed model, using sequential models was sufficient. The proposed model consists of three sequential models – the discriminator and generator networks and the GAN model, which combines both of them. The reason for creating the combined model is that the discriminator and generator networks are trained separately; however, the generator needs to access the discriminator to receive the information about its error, which is then reflected in the update of weights. Keras's trick is to achieve this behavior by setting the model's weights as not trainable. That means the discriminator itself is trained separately, and the generator is trained using the combined model, which sets the discriminator's weights as not trainable. The implementation of the combined model is shown in Listing 5.1.

In Section 5.1.4 was mentioned, that the core data structures in Keras are models and layers. Therefore, the generator and discriminator models were implemented as a sequence of layers already provided by the Keras framework. Visualizations of the generator and discriminator Keras models are shown in Figure 5.1 and Figure 5.2 respectively.

```
def define_gan(g_model, d_model, adam_learning_rate=0.0002):
    // sets the discriminator's weights as not trainable
    d_model.trainable = False

    // defines the combined sequential model
    model = Sequential(name='GAN-model')
    model.add(g_model)
    model.add(d_model)

    // sets the Adam optimizer
    opt = Adam(lr=adam_learning_rate, beta_1=0.5)

    // creates the model
    model.compile(loss='binary_crossentropy', optimizer=opt)
    return model
```

Listing 5.1: Implementation of the GAN model which combines the generator and discriminator networks.

Conv2D_1	Input:	(None, 96, 96, 1)	Output:	(None, 48, 48, 64)
Leaky Relu_1	Input:	(None, 48, 48, 64)	Output:	(None, 48, 48, 64)
Conv2D_2	Input:	(None, 48, 48, 64)	Output:	(None, 24, 24, 128)
Batch normalization_1	Input:	(None, 24, 24, 128)	Output:	(None, 24, 24, 128)
Leaky Relu_2	Input:	(None, 24, 24, 128)	Output:	(None, 24, 24, 128)
Conv2D_3	Input:	(None, 24, 24, 128)	Output:	(None, 12, 12, 256)
Batch normalization_2	Input:	(None, 12, 12, 256)	Output:	(None, 12, 12, 256)
Leaky Relu_3	Input:	(None, 12, 12, 256)	Output:	(None, 12, 12, 256)
Conv2D_4	Input:	(None, 12, 12, 256)	Output:	(None, 6, 6, 512)
Batch normalization_3	Input:	(None, 6, 6, 512)	Output:	(None, 6, 6, 512)
Leaky Relu_4	Input:	(None, 6, 6, 512)	Output:	(None, 6, 6, 512)
Flatten	Input:	(None, 6, 6, 512)	Output:	(None, 18432)
Dense with sigmoid	Input:	(None, 18432)	Output:	(None, 1)

Figure 5.1: Visualization of the proposed discriminator model implemented in Keras.

5.4 Implementation of methods for performance improvement

Several methods were proposed in Section 4.2 to improve the performance of the DCGAN model. Their implementation is discussed in this section. At first, Section 5.4.1 provides information about the implementation of data augmentation described in Section 4.1.3. Implementation of spectral normalization and minibatch discrimination methods is described in Section 5.4.2. These two methods share the same feature – they could be implemented as layers in Keras. Other methods that were implemented outside the implemented model itself are described in Section 5.4.3.

5.4.1 Implementation of data augmentation

Since the solution was coded using the Keras framework, the most convenient way to incorporate data augmentation to the solution is via the `ImageDataGenerator` class. One can choose from many possible transformations that are then applied to the original dataset. Choosing transformations is an important step because it can significantly affect the generated images. For example, in case of fingerprints, it would not make sense to use a vertical

Dense	Input:	(None, 100)	Output:	(None, 4608)
Batch normalization_1	Input:	(None, 4608)	Output:	(None, 4608)
Relu_1	Input:	(None, 4608)	Output:	(None, 4608)
Reshape	Input:	(None, 4608)	Output:	(None, 3, 3, 512)
Conv2DTransposed_1	Input:	(None, 3, 3, 512)	Output:	(None, 6, 6, 256)
Batch normalization_2	Input:	(None, 6, 6, 256)	Output:	(None, 6, 6, 256)
Relu_2	Input:	(None, 6, 6, 256)	Output:	(None, 6, 6, 256)
Conv2DTransposed_2	Input:	(None, 6, 6, 256)	Output:	(None, 12, 12, 128)
Batch normalization_3	Input:	(None, 12, 12, 128)	Output:	(None, 12, 12, 128)
Relu_3	Input:	(None, 12, 12, 128)	Output:	(None, 12, 12, 128)
Conv2DTransposed_3	Input:	(None, 12, 12, 128)	Output:	(None, 24, 24, 64)
Batch normalization_4	Input:	(None, 24, 24, 64)	Output:	(None, 24, 24, 64)
Relu_4	Input:	(None, 24, 24, 64)	Output:	(None, 24, 24, 64)
Conv2DTransposed_4	Input:	(None, 24, 24, 64)	Output:	(None, 48, 48, 32)
Batch normalization_5	Input:	(None, 48, 48, 32)	Output:	(None, 48, 48, 32)
Relu_5	Input:	(None, 48, 48, 32)	Output:	(None, 48, 48, 32)
Conv2DTransposed_5	Input:	(None, 48, 48, 32)	Output:	(None, 96, 96, 32)
Batch normalization_6	Input:	(None, 96, 96, 32)	Output:	(None, 96, 96, 32)
Relu_6	Input:	(None, 96, 96, 32)	Output:	(None, 96, 96, 32)
Conv2D	Input:	(None, 96, 96, 32)	Output:	(None, 96, 96, 1)
Tanh	Input:	(None, 96, 96, 1)	Output:	(None, 96, 96, 1)

Figure 5.2: Visualization of the proposed generator model implemented in Keras.

flip, because the images could cause a problem during their evaluation. It is more convenient to use a horizontal flip, which keeps images realistically looking. Another transformation that can be used is the featurewise normalization. It includes `featurewise_center`, which sets input mean to zero over the dataset, and `featurewise_std_normalization`, which divides inputs by standard deviation of the dataset. To be able to apply standard normalization, the `ImageDataGenerator` needs to learn the mean and the standard deviation of the dataset before generating the actual data.

5.4.2 Implementation of spectral normalization and minibatch discrimination

The implementation of spectral normalization and minibatch discrimination methods caused a few changes in the structure of the discriminator's Keras model. The updated model can be seen in Appendix B.

Spectral normalization described in Section 4.2.3 provides a great advantage, because its definition allows the implementation within a model's layer. Therefore, spectral normalization was implemented using the updated version of existing Keras layers used in the discriminator, as visualized in Figure 5.3.

As mentioned in Section 4.2.2, the minibatch discrimination method computes the similarity of samples within the batch and appends it to the dense intermediate layer of the discriminator. Custom Keras layer was implemented and added to the model structure right after the flattening layer, as shown in Figure 5.4.

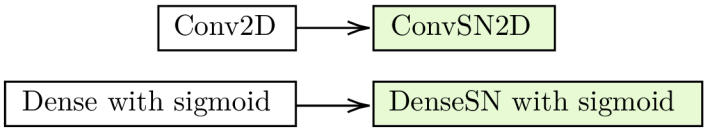


Figure 5.3: Changes in the discriminator model caused by the spectral normalization method. Elements highlighted by green color show the updated layers that implement the spectral normalization.

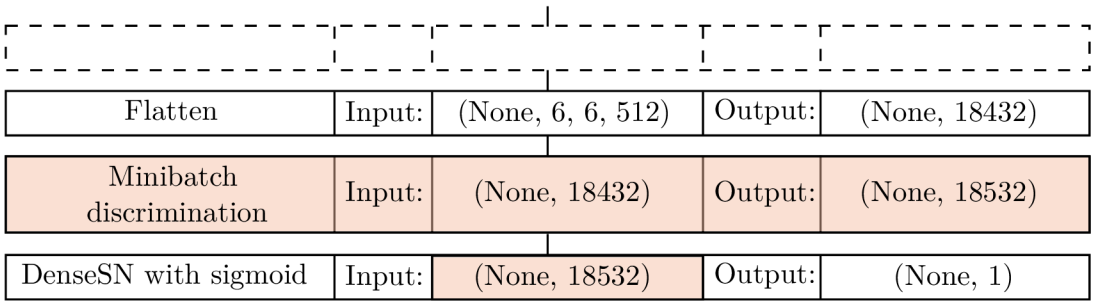


Figure 5.4: Changes in the discriminator model caused by the minibatch discrimination method. Elements highlighted by red color show newly added layer that is responsible for minibatch discrimination, which changed the input data shape of the subsequent dense layer.

5.4.3 Implementation of methods that did not affect the model itself

The methods that are not a part of the discriminator model itself are incorporated in the training loop. To be able to better describe the implementation of these methods, Listing 5.2 shows the training loop written in pseudocode. Each `step` represents training on one batch of data.

Smoothing the labels and making them noisy takes place during the preparation of a batch of data, as can be seen in rows 6–8 and 12–13 of the training loop. Since the label smoothing is one-sided, only the real data labels are smoothed; however, both the real and fake data labels are randomly flipped.

Making the labels noisy is implemented as a function that takes the input vector of labels y and the probability of flipping the labels. It computes the number of labels that should be inverted based on the probability using the following formula:

$$num\_to\_flip = flip\_prob \cdot input\_vect\_length, \tag{5.1}$$

where $flip\_prob$ is the probability of flipping the label, and $input\_vect\_length$ stands for the length of the input vector. The given number of indices is then randomly chosen, that are inverted using the rule $y_{new} = 1 - y_{old}$.

Label smoothing is implemented as a function, which takes the vector of randomly flipped real labels y_{noisy} and the value of factor α, and returns the updated vector of values in range $\langle 1 - \alpha, 1 \rangle$.

Experience replay happens after the training of the discriminator, as can be seen in rows 17–21 of the training loop. Each training step, one of the generated samples is added to the buffer. Once the given number of training steps defined by `replay_step` is reached, the discriminator is trained of the batch of samples from the buffer. Then the buffer is cleared and on the following training step it starts to save the generated samples again.

```
1   def train(dataset, batch size, number of epochs, replay_step):
2       variables initialization
3       train_steps = (size of dataset / batch size) * number of epochs
4
5       for step in train_steps:
6           x_real, y_real = prepare real data and labels
7           make real labels noisy
8           smooth real labels
9
10          train the discriminator on real data
11
12          x_fake, y_fake = prepare fake data and labels
13          make fake labels noisy
14
15          train the discriminator on fake data
16
17          if step == replay_step:
18              train discriminator on samples in the buffer
19              empty the buffer
20          else:
21              add one random fake sample into the buffer
22
23          prepare data and train the generator
```

Listing 5.2: Pseudocode of the training loop.

5.5 Results evaluation

One of the common methods for evaluating DCGAN models is called Fréchet Inception Distance [14]. It measures both the quality of generated images and their diversity. However,

since this thesis aims to generate fingerprints, there is a considerable advantage in using other tools specifically designed for measuring the quality of fingerprint images. One of those tools is a publicly available software from NIST (National Institute of Standards and Technology), called NFIQ (NIST Finger Image Quality). The performance of the proposed model during training is discussed in Section 5.5.1. The results of the proposed model evaluated by both methods mentioned above are provided later in this section.

5.5.1 Training evaluation

As already mentioned, training GANs often leads to problems with stability. That is the reason why the proposed solution implements additional methods that proved to be able to help with this issue. In Figure 5.5, losses tracked during the training of the proposed model visualized in Figure 4.4, are shown. This model does not implement any of the additional methods. Further in this work, this model will be referred to as "DCGAN_BASE". One can see that the loss of the discriminator remains relatively stable during the training; however, the generator's loss is significantly unstable and collapses to zero several times during training. That suggests that the generator is not able to generate fake samples in a consistent way, and it is easy for the discriminator to identify the generated samples.

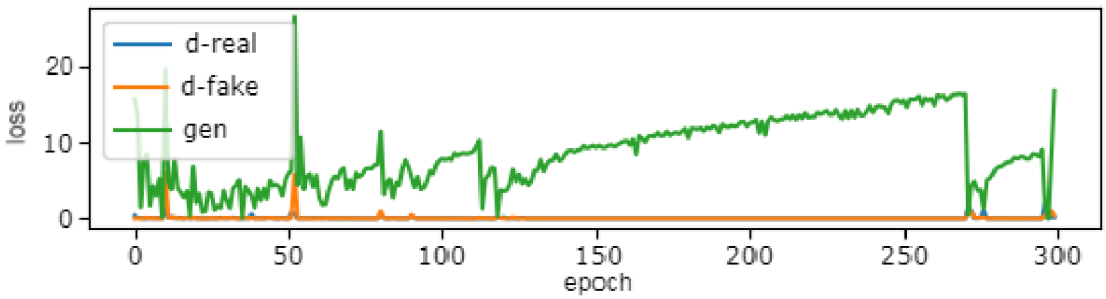


Figure 5.5: Losses tracked on each epoch during the training of the proposed model without any additional methods for performance improvement.

To stabilize the training, label smoothing and noisy labels methods were implemented, because they can have a regularizing effect as mentioned in Section 4.2.1. Together with the mentioned methods, the spectral normalization, which is used to stabilize the training of GANs, was implemented. This model will be referred to as "DCGAN_EXT". As can be seen in Figure 5.6, the implemented methods had a significant impact on the training process of the proposed model.

However, even though the training became more stable by implementing the methods mentioned above, the mode-collapse showed to be a big problem. To address this problem, the minibatch discrimination and experience replay methods were implemented. From now on, this model will be referred to as "DCGAN_FULL". In Figure 5.7, one can see that apart from the noise at the start of the training, the process remained stable.

5.5.2 Fréchet Inception Distance

One of the conventional methods to measure the performance of GANs is the Inception Score. It uses an inception model pre-trained on the ImageNet dataset, which classifies the generated images and predicts the conditional probability $p(y|x)$, where y is the label and x is the generated data. The idea behind measuring the quality of images is that the

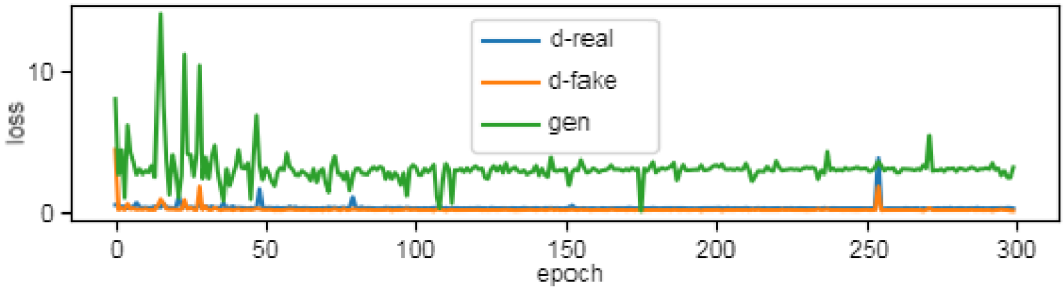


Figure 5.6: Losses tracked on each epoch during the training of the proposed model after the implementation of methods for training stabilization: spectral normalization, label smoothing, and noisy labels.

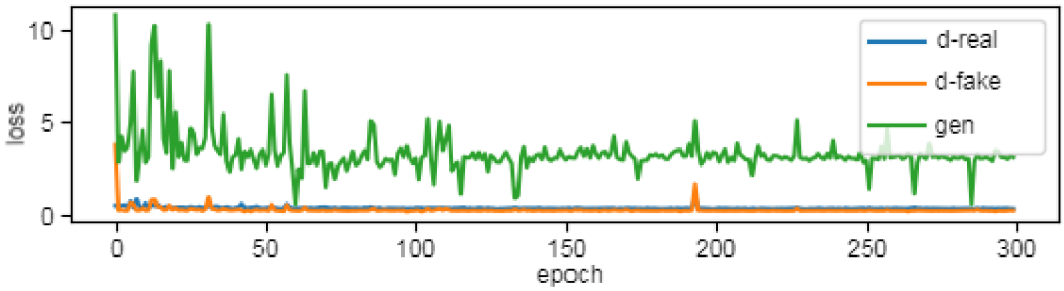


Figure 5.7: Losses tracked on each epoch during the training of the proposed model after the implementation of methods that reduce the mode-collapse: experience replay, and minibatch discrimination.

probability $p(y|x)$ should have low entropy, which reflects that the images belong to just a few classes. On the other hand, the entropy across those images should be high, which reflects the diversity of the images. The problem with the Inception Score is that it does not use real samples statistics for their comparison with the generated ones.

The improved method based on the Inception Score is called the Fréchet Inception Distance (FID) [14]. This method uses features from an intermediate layer of the previously mentioned inception model. For the given features, FID uses a multi-dimensional Gaussian distribution with the mean μ and the covariance Σ. The difference of gaussian distributions of synthetic and real images is measured by the Fréchet distance d. The formula for computing the FID is the following [14]:

$$d^2((\mu_g, \Sigma_g), (\mu_x, \Sigma_x)) = \|\mu_x - \mu_g\|_2^2 + \mathrm{Tr}(\Sigma_x + \Sigma_g - 2(\Sigma_x\Sigma_g)^{\frac{1}{2}}), \tag{5.2}$$

where μ_x and Σ_x represent a Gaussian distribution of real samples, and μ_g and Σ_g represent a Gaussian distribution of generated samples. Tr stands for the sum of the diagonal elements.

The average, minimal, and maximal FID score for each model discussed in Section 5.5.1 is shown in Table 5.1. Figure 5.8 presents the progression of the FID score during the training. All models tend to have a worse FID score after about 100 training epochs. That suggests that there are either samples with low quality or that the diversity of samples decreases due to the mode-collapse. In Figure 5.9 can be seen that the FID score corresponds to the visual quality of images. The third row, which is highlighted by a red frame, represents samples after 60 epochs. Their quality is decent, and they also have significant

variance. However, then the quality and variance start to get worse (and the FID score rises). Therefore, Table 5.2 provides the information about the FID score values just for the first 100 epochs. In the rest of this thesis, only the model trained to the 100th epoch is considered.

Table 5.1: The Fréchet Inception Distance of the proposed model over 300 epochs.

Model/Dataset	**FID**		
	Average	**Min**	**Max**
DCGAN-BASE/SOCOFing	71.47	38.93	101.04
DCGAN-EXT/SOCOFing	66.71	38.56	121.94
DCGAN-FULL/SOCOFing	54.56	34.21	98.41

Table 5.2: The Fréchet Inception Distance of the proposed model over 100 epochs.

Model/Dataset	**FID**		
	Average	**Min**	**Max**
DCGAN-BASE/SOCOFing	65.44	41.96	93.19
DCGAN-EXT/SOCOFing	54.94	38.56	85
DCGAN-FULL/SOCOFing	48.52	34.21	93.05

Since the original paper [27] reached FID score 70.5, it is clear that the solution proposed by this work achieves better results. Their achieved FID score is mostly comparable with the average score of the proposed DCGAN-BASE model, which does not implement any of the methods for performance improvement. That suggests, that the additional methods proposed in this work have a significant impact on the quality of generated fingerprints.

5.5.3 NIST Finger Image Quality

The original version of NFIQ was developed back in 2004 as the first publicly accessible fingerprint quality assessment tool. NFIQ was the first tool to allow a universal interpretation of fingerprint quality, which resulted in better fingerprint recognition systems by identifying appropriate samples for their testing. Later, in 2011, started the collaboration between several institutions, including NIST, Federal Office for Information Security, Federal Criminal Police Office, and others, which resulted in a new version of this tool called NFIQ 2.0. It became the reference implementation of ISO/IEC 29794-4 Biometric sample quality – Part 4: Finger image data standard. According to that standard, it provides a quality score ranging from 0 to 100, where 100 is the best result. NFIQ 2.0 was explicitly developed for images captured at 500 dpi using optical sensors or scanned from inked cards. That suits great for the SOCOFing dataset because it satisfies both requirements. However, to be able to get a quality score of the real samples, their bits-per-pixel values needed to be converted from 32 bits to 8 bits. NFIQ 2.0 is also not able to identify minutiae for SOCOFing and FVC2006 DB1 datasets, which suggests that the smaller image size leads to problems with minutiae identification since both datasets have a smaller image size than the other datasets. Table 5.3 shows the average quality score for each dataset. The size of images was then doubled just to get more meaningful information about SOCOFing and FVC2006 DB1 datasets. Table 5.4 shows the average NFIQ 2.0 score for those resized images.

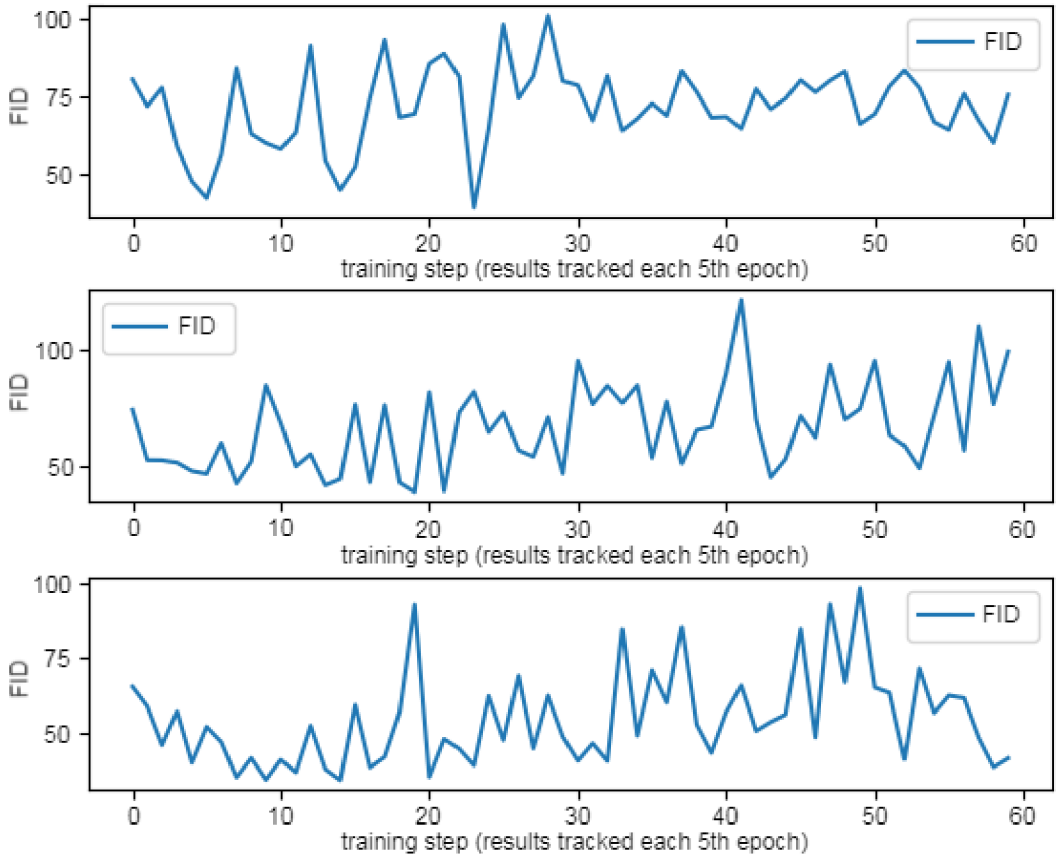


Figure 5.8: The FID score tracked on each 5th epoch during the training of the proposed models: DCGAN-BASE, DCGAN-EXT, and DCGAN-FULL in the respective order.

Based on the results from Section 5.5.2 and based on the evaluation of the visual quality of the generated fingerprint images, models with the following training phases were selected for evaluation by NFIQ 2.0: DCGAN-BASE trained for 80 epochs, DCGAN-EXT trained for 100 epochs and DCGAN-FULL, which was trained for 60 epochs. Table 5.3 provides information about the average NFIQ 2.0 quality score obtained over the generated dataset of 100 samples from each of the models. The obtained score proves that even for newly generated data samples, NFIQ 2.0 has a problem with identifying minutiae. Therefore, the generated samples were also resized to double their original size. The average score reached by the resized images is shown in Table 5.4. One can notice, that the NFIQ 2.0 quality score is even higher for the generated images than it is for those from the original dataset. This was a surprising result. In the case of samples generated by the DCGAN-BASE model, this could be caused by a significantly higher number of minutiae detected. However, the results of the DCGAN-FULL model present a higher quality score with almost the same number of minutiae detected. These results proved, that the proposed model can generate samples highly similar to those from the original dataset.

Table 5.3: Average NFIQ 2.0 quality score and minutiae count of samples from each dataset. The generated samples were produced by models trained on the SOCOFing dataset.

Dataset	**NFIQ 2.0 Score**	**Minutiae count**
FVC2006-DB1	2.35	0
FVC2006-DB2	39.2	71.3
FVC2006-DB3	48.33	80.26
FVC2006-DB4	29.96	43.7
SOCOFing	3.4	0
Generated samples		
DCGAN-BASE	4.13	0
DCGAN-EXT	5	0
DCGAN-FULL	4.42	0

Table 5.4: Average NFIQ 2.0 quality score and minutiae count of 500 randomly selected samples from FVC2006 DB1 and SOCOFing datasets. The generated samples were produced by models trained on the SOCOFing dataset. All images were resized to double of their original size.

Dataset	**NFIQ 2.0 Score**	**Minutiae count**
FVC2006-DB1	23.24	32.72
SOCOFing	15.33	37.34
Generated samples		
DCGAN-BASE	16.21	83.52
DCGAN-EXT	17.65	74.09
DCGAN-FULL	18.21	41.26

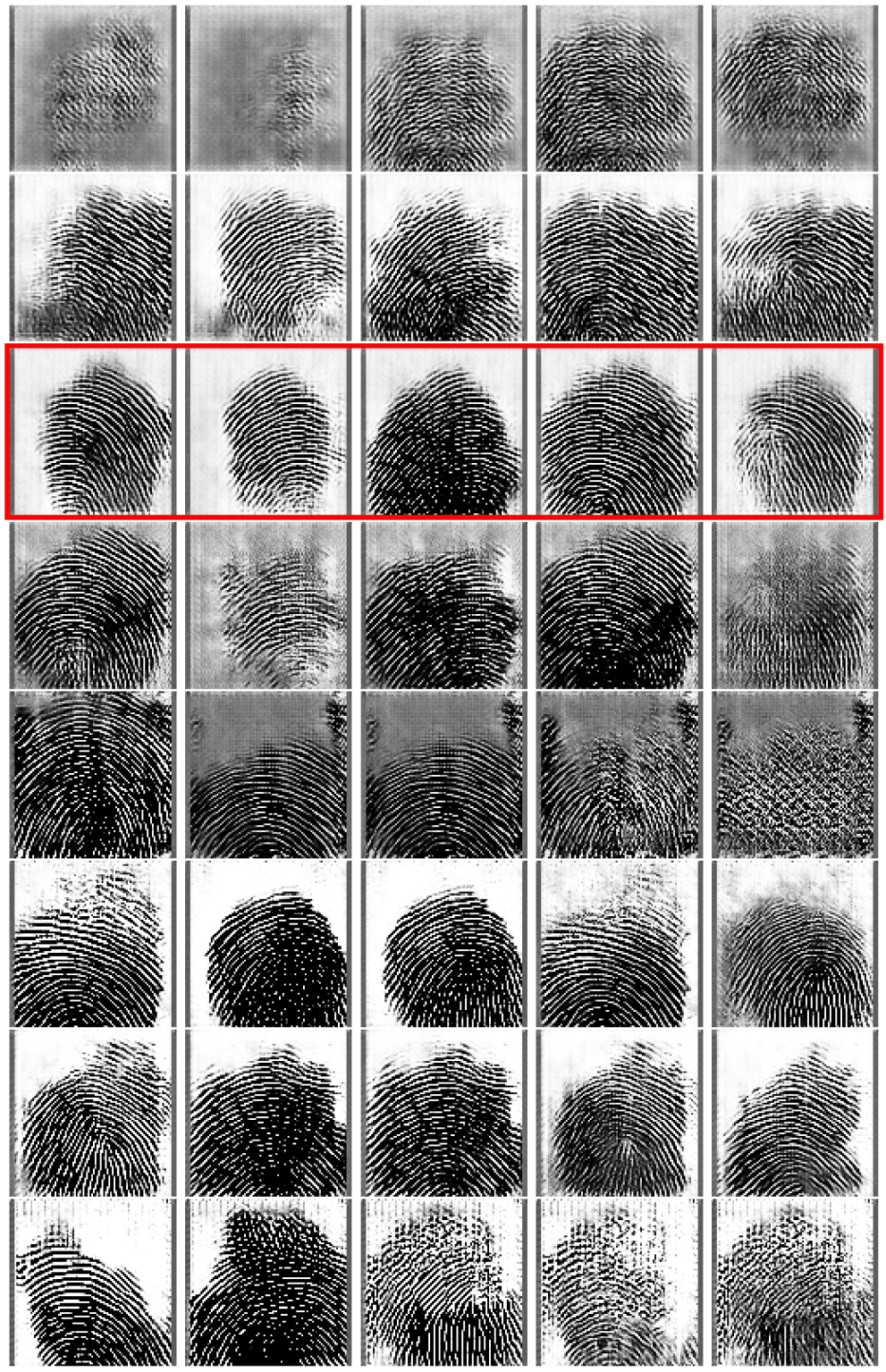


Figure 5.9: The generated fingerprint images for five input latent vectors generated using the DCGAN-FULL model. The first row contains the results after 20 training epochs. Each subsequent row contains samples after another 20 epochs, except for the last row, which contains a more advanced phase of training after 220 epochs.

Chapter 6

Conclusion

The goal of this thesis was to create a synthetic fingerprint generator based on the principle of generative adversarial networks (GANs). The proposed solution proved that the deep convolutional GAN (DCGAN) model is capable of generating fingerprints that are highly similar to samples from the original dataset. Alongside the proposed DCGAN architecture, there are several methods discussed, that were implemented to increase the stability of the training process and the quality of generated fingerprints. Compared with the results of the work Finger-GAN [27], which originally purposed the DCGAN model for generating synthetic fingerprints, the additional methods implemented within this work proved to have a significant impact on the quality of the generated fingerprints.

The results were evaluated using the Fréchet Inception Distance (FID), and the NIST Finger Image Quality 2.0 (NFIQ 2.0). By implementing the proposed methods, the average FID score over 100 training epochs was improved by 25 % and the NFIQ 2.0 score by 12 %, compared with the results of the plain model, which does not implement any of these methods. The best results were achieved with the model that implements all of the proposed methods. Fingerprint images generated after 60 epochs were evaluated as the best results. At this point, the model achieved the FID score of 36.8 and the average NFIQ 2.0 quality score of 18.21.

Even though the proposed solution proved the model's capability of generating a complex structure of fingerprints, it is still affected by the mode-collapse problem. Therefore, future work should be focused on reducing its impact. One of the options to improve the results is using fingerprint labels from the dataset and consequently creating a *conditional GAN* model. Another improvement could be reached by the implementation of *unrolled GAN* model, which is focused primarily on the reduction of the mode-collapse problem.

In Chapter 2, this thesis provides basic information about biometrics with emphasis on fingerprints and describes the principle of SFinGe. Chapter 3 describes the common principles of artificial neural networks, together with a closer look at GAN and DCGAN models. In Chapter 4 is described the proposed model, including the methods implemented for the improvement of its performance. This chapter also provides an overview of accessible fingerprint datasets. Finally, Chapter 5 describes technical details of the implementation of the proposed solution and provides the evaluation of results on the generated datasets. The surprising result was that the average NFIQ 2.0 quality score over the generated database of 100 samples was even higher than for the original samples from the SOCOFing dataset. Given the information above, the main aim of the thesis has been reached.

Bibliography

[1] Aggarwal, C. C. *Neural Networks and Deep Learning.* Cham, Switzerland: Springer, 2018. ISBN 978-3-319-94463-0.

[2] AuthenTec, Inc. The Fundamentals. *TruePrint^{TM} Technology* [online]. AuthenTec, Inc., June 2002 [cit. 2019-10-15]. Available at: http://www.zvetcobiometrics.com/Documents/Trueprinttechnology.ppt.

[3] Bergstra, J. and Bengio, Y. Random Search for Hyper-Parameter Optimization. *Journal of Machine Learning Research* [online]. JMLR, Inc. February 2012, vol. 13, p. 281–305, [cit. 2019-01-17]. ISSN 1533-7928. Available at: http://www.jmlr.org/papers/volume13/bergstra12a/bergstra12a.pdf.

[4] Cappelli, R., Ferrara, M., Franco, A. and Maltoni, D. Fingerprint verification competition 2006. *Biometric Technology Today* [online]. Elsevier. July 2007, vol. 15, no. 7, p. 7–9, [cit. 2019-12-15]. DOI: 10.1016/S0969-4765(07)70140-6. ISSN 0969-4765. Available at: https://doi.org/10.1016/S0969-4765(07)70140-6.

[5] Chintala, S. et al. *How to Train a GAN? Tips and tricks to make GANs work* [online]. GitHub, Inc., ©2020. revised 2020-03-05 [cit. 2020-04-15]. Available at: https://github.com/soumith/ganhacks.

[6] Chollet, F. The Sequential model. *Keras* [online]. Keras Team, Google, April 2020 [cit. 2020-06-16]. Available at: https://keras.io/guides/sequential_model/.

[7] Chollet, F. The Functional API. *Keras* [online]. Keras Team, Google, April 2020 [cit. 2020-06-16]. Available at: https://keras.io/guides/functional_api/.

[8] Diederik P. Kingma, J. B. Adam: A Method for Stochastic Optimization. *ArXiv.org* [online]. version 9. January 2017, revised 2017-01-30, [cit. 2020-06-07]. arXiv:1412.6980. Available at: https://arxiv.org/abs/1412.6980.

[9] Drahanský, M. et al. *Biometrie.* Brno: Computer Press, s.r.o, 2011. ISBN 978-80-254-8979-6.

[10] Ertel, W. *Introduction to Artificial Intelligence.* 2nd ed. Cham, Switzerland: Springer International Publishing, 2017. ISBN 978-3-319-58486-7.

[11] Goodfellow, I., Bengio, Y. and Courville, A. *Deep learning.* Cambridge, MA: MIT Press, 2016. ISBN 9780262035613.

[12] Goodfellow, I. J. et al. Generative Adversarial Networks. *ArXiv.org* [online]. June 2014, [cit. 2019-01-07]. arXiv:1406.2661. Available at: https://arxiv.org/abs/1406.2661.

[13] Google. *Generative Adversarial Networks: Background: What is a Generative Model?* [online]. USA: Google. Last updated 2019-05-24 [cit. 2020-06-05]. Available at: https://developers.google.com/machine-learning/gan/generative.

[14] Heusel, M., Ramsauer, H., Unterthiner, T., Nessler, B. and Hochreiter, S. GANs Trained by a Two Time-Scale Update Rule Converge to a Local Nash Equilibrium. *ArXiv.org* [online]. version 6. January 2018, revised 2018-01-12, [cit. 2019-06-07]. arXiv:1706.08500. Available at: https://arxiv.org/abs/1706.08500.

[15] International Organization for Standardization. *International Standard ISO/IEC 2382-37 Information technology – Vocabulary – Part 37: Biometrics*. 2017. Retrieved from https://standards.iso.org/ittf/PubliclyAvailableStandards/.

[16] Jain, A. K. Technology: Biometric recognition. *Nature* [online]. London: Nature Publishing Group. September 2007, vol. 449, no. 7158, p. 38–40, [cit. 2019-01-17]. DOI: 10.1038/449038a. Available at: https://doi.org/10.1038/449038a.

[17] Kanich, O. *Fingerprint damage simulation: a simulation of fingerprint distortion, damaged sensor, pressure and moisture*. Saarbrücken: Lambert academic publishing, 2014. ISBN 978-3-659-63942-5.

[18] Karn, U. An Intuitive Explanation of Convolutional Neural Networks. *The data science blog* [online]. August 2016 [cit. 2019-01-17]. Available at: https://ujjwalkarn.me/2016/08/11/intuitive-explanation-convnets/.

[19] Karpathy, A. Neural networks. *CS231n Convolutional Neural Networks for Visual Recognition* [online]. Stanford University [cit. 2019-12-15]. Available at: https://cs231n.github.io/neural-networks-1/.

[20] Karpathy, A. Convolutional neural networks. *CS231n Convolutional Neural Networks for Visual Recognition* [online]. Stanford University [cit. 2019-01-24]. Available at: https://cs231n.github.io/convolutional-networks/.

[21] Krizhevsky, A., Sutskever, I. and Hinton, G. E. ImageNet Classification with Deep Convolutional Neural Networks. *Communications of the ACM*. New York: Association for Computing Machinery. May 2017, vol. 60, no. 6, p. 84–90. DOI: 10.1145/3065386. ISSN 0001-0782.

[22] Kücken, M. and Newell, A. C. Fingerprint formation. *Journal of Theoretical Biology*. Elsevier. 2005, vol. 235, no. 1, p. 71–83. ISSN 0022-5193.

[23] Li, F., Johnson, J. and Yeung, S. *Lecture 13: Generative Models* [online]. Stanford, CA: SVL Lab, Stanford University, May 2017 [cit. 2020-05-25]. Course slides. Available at: http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture13.pdf.

[24] Liu, Y. and Mehta, S. *Hands-On Deep Learning Architectures with Python*. Birmingham, UK: Packt Publishing Ltd., April 2019. ISBN 978-1-78899-808-6.

[25] Maltoni, D. et al. *Handbook of Fingerprint Recognition*. Dordrecht: Springer, 2006. ISBN 9780387954318.

[26] Mehlig, B. Artificial Neural Networks. *ArXiv.org* [online]. version 2. February 2019, revised 2019-02-01, [cit. 2019-12-15]. arXiv:1901.05639. Available at: https://arxiv.org/abs/1901.05639.

[27] Minaee, S. and Abdolrashidi, A. Finger-GAN: Generating Realistic Fingerprint Images Using Connectivity Imposed GAN. *ArXiv.org* [online]. December 2018, [cit. 2019-12-15]. arXiv:1812.10482. Available at: https://arxiv.org/abs/1812.10482.

[28] MissingLink.ai. The Complete Guide to Artificial Neural Networks: Concepts and Models. *Neural Network Concepts* [online]. MissingLink.ai [cit. 2019-12-15]. Available at: https://missinglink.ai/guides/neural-network-concepts/complete-guide-artificial-neural-networks/.

[29] Miyato, T. et al. Spectral Normalization for Generative Adversarial Networks. *ArXiv.org* [online]. February 2018, [cit. 2020-05-20]. arXiv:1802.05957. Available at: https://arxiv.org/pdf/1802.05957.pdf.

[30] Pfau, D. and Vinyals, O. Connecting Generative Adversarial Networks and Actor-Critic Methods. *ArXiv.org* [online]. version 2. January 2017, revised 2017-01-18, [cit. 2020-06-13]. arXiv:1610.01945. Available at: https://arxiv.org/abs/1610.01945.

[31] Radford, A. et al. Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks. *ArXiv.org* [online]. version 2. January 2016, revised 2016-01-07, [cit. 2019-01-17]. arXiv:1511.06434. Available at: https://arxiv.org/abs/1511.06434.

[32] Salimans, T. et al. Improved Techniques for Training GANs. *ArXiv.org* [online]. June 2016, [cit. 2020-05-20]. arXiv:1606.03498. Available at: https://arxiv.org/pdf/1606.03498.pdf.

[33] Shaffer, D. *FIRS IAFIS — FBI: Privacy Impact Assessment for the Fingerprint Identification Records System (FIRS) Integrated Automated Fingerprint Identification System (IAFIS) Outsourcing for Noncriminal Justice Purposes - Channeling* [online]. USA: Federal Bureau of Investigation, May 2008 [cit. 2020-05-30]. Available at: https://www.fbi.gov/services/information-management/foipa/privacy-impact-assessments/firs-iafis.

[34] Shahriari, B. et al. Taking the Human Out of the Loop: A Review of Bayesian Optimization. *Proceedings of the IEEE*. IEEE. January 2016, vol. 104, no. 1, p. 148–175. ISSN 0018-9219.

[35] Shehu, Y. I., Ruiz Garcia, A., Palade, V. and James, A. Detection of Fingerprint Alterations Using Deep Convolutional Neural Networks. In: *Proceedings of the International Conference on Artificial Neural Networks (ICANN 2018)* [online]. Cham, Switzerland: Springer International Publishing, October 2018 [cit. 2020-01-20]. DOI: 10.1007/978-3-030-01418-6_6. ISBN 978-3-030-01418-6. Lecture Notes in Computer Science.

[36] Shehu, Y. I., Ruiz Garcia, A., Palade, V. and James, A. Sokoto Coventry Fingerprint Dataset. *ArXiv.org* [online]. July 2018, [cit. 2020-01-20]. arXiv:1807.10609. Available at: https://arxiv.org/abs/1807.10609.

[37] Svoradová, V. *Generování onemocnění kůže do syntetických otisků prstů z SFinGe*. Brno, 2019. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology.

[38] Theano Development Team. Theano: A Python framework for fast computation of mathematical expressions. *ArXiv.org* [online]. May 2016, [cit. 2020-06-20]. arXiv:1605.02688. Available at: http://arxiv.org/abs/1605.02688.

[39] Torch Contributors. PYTORCH DOCUMENTATION. *PyTorch* [online]. Torch Contributors, ©2019 [cit. 2020-06-16]. Available at: https://pytorch.org/docs/stable/index.html.

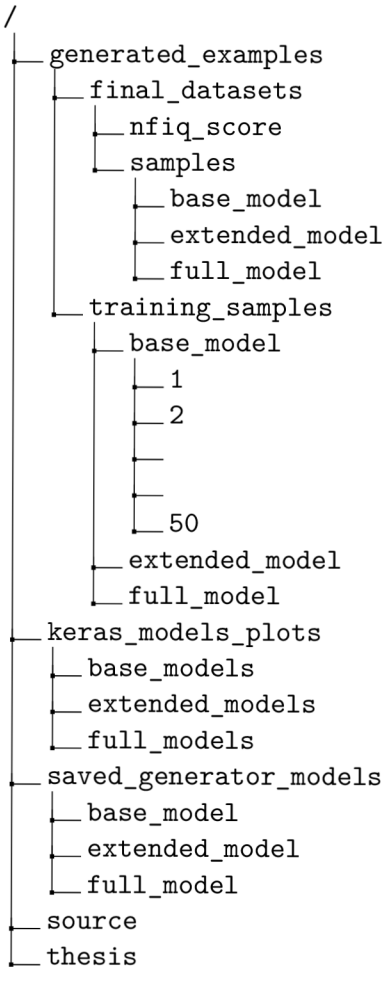
[40] University of Bologna. Databases. *FVC 2006: Fingerprint Verification Competition* [online]. University of Bologna, ©2006 [cit. 2019-12-15]. Available at: http://bias.csr.unibo.it/fvc2006/databases.asp.

[41] University of Bologna. Fingerprint Generation. *Biometric System Laboratory - Webpage* [online]. University of Bologna, ©2020 [cit. 2019-12-15]. Available at: http://biolab.csr.unibo.it/research.asp?organize=Activities&select=&selObj=12&pathSubj=111%7C%7C12&Req=&.

Appendix A

Contents of the included storage media

Directory tree of the included storage media looks like the following:



- File “README.TXT”
  - This file, located in the root directory, contains a similar text to this one for clarification of the contents of the included storage media.

- Folder “generated_examples”
  - This folder contains exported examples by the proposed Keras models. In the subfolder „final_datasets“, 100 samples exported from the proposed models after the certain number of epochs are located. DCGAN-BASE model was trained for 80 epochs, DCGAN-EXT model for 100 epochs and DCGAN-FULL model for 60 epochs. The subfolder contains also the csv files with NFIQ 2.0 score of all of the results presented in Section 5.5.3.
  - The second subfolder „training_samples“ provides samples generated from 50 different input noise vectors by each of the proposed models – DCGAN-BASE, DCGAN-EXT and DCGAN-FULL. These images were saved each 20th epoch over 300 training epochs in total.
- Folder “keras_models_plots”
  - This folder contains exported visualizations of the proposed Keras models. It includes subfolders for base models that do not implement any of the proposed additional methods, as well as extended models implementing some of the proposed methods and final models that implement all of the methods described in this thesis.
- Folder “saved_generator_models”
  - The saved weights of the generator models that provide the best results as discussed in Section 5.5.3 are provided in this folder. One can easily use these pretrained models to generate new data and create more extensive datasets.
- Folder “source”
  - This folder contains all of the source files. There is also the license file “LICENSE”, which belongs to the file “SpectralNormalizationKeras.py”, which implements Keras layers for spectral normalization. The folder also contains the “README.TXT” file, which provides information about running the implemented solution. The reader is encouraged to install the packages defined in the file „requirements.txt“ to his local system, which assures that the solution works correctly.
- Folder “thesis”
  - This folder contains electronic version of this text.

Appendix B

Proposed model with extensions implemented in Keras

ConvSN2D_1	Input:	(None, 96, 96, 1)	Output:	(None, 48, 48, 64)
Leaky Relu_1	Input:	(None, 48, 48, 64)	Output:	(None, 48, 48, 64)
ConvSN2D_2	Input:	(None, 48, 48, 64)	Output:	(None, 24, 24, 128)
Batch normalization_1	Input:	(None, 24, 24, 128)	Output:	(None, 24, 24, 128)
Leaky Relu_2	Input:	(None, 24, 24, 128)	Output:	(None, 24, 24, 128)
ConvSN2D_3	Input:	(None, 24, 24, 128)	Output:	(None, 12, 12, 256)
Batch normalization_2	Input:	(None, 12, 12, 256)	Output:	(None, 12, 12, 256)
Leaky Relu_3	Input:	(None, 12, 12, 256)	Output:	(None, 12, 12, 256)
ConvSN2D_4	Input:	(None, 12, 12, 256)	Output:	(None, 6, 6, 512)
Batch normalization_3	Input:	(None, 6, 6, 512)	Output:	(None, 6, 6, 512)
Leaky Relu_4	Input:	(None, 6, 6, 512)	Output:	(None, 6, 6, 512)
Flatten	Input:	(None, 6, 6, 512)	Output:	(None, 18432)
Minibatch discrimination	Input:	(None, 18432)	Output:	(None, 18532)
DenseSN with sigmoid	Input:	(None, 18532)	Output:	(None, 1)

Figure B.1: The proposed model architecture after the implementation of spectral normalization and minibatch discrimination methods. Elements highlighted by green color are layers that were changed to equivalent layers that implement spectral normalization. Elements highlighted by red color show newly added layer that is responsible for minibatch discrimination, which changed the input data shape of the subsequent dense layer.

Appendix C

Generated samples

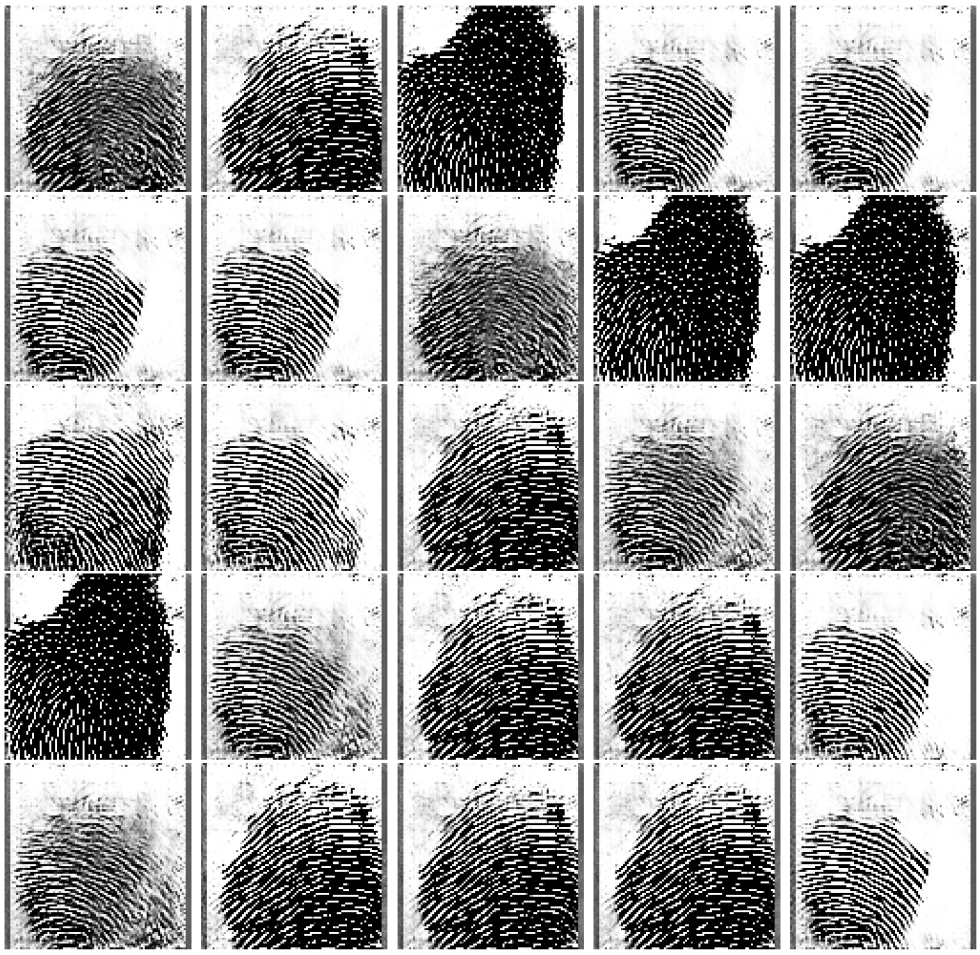


Figure C.1: The generated fingerprint images by the DCGAN-BASE model after 80 training epochs. Each sample was generated from a random latent vector. It is clear that this model is significantly affected by the mode-collapse problem.

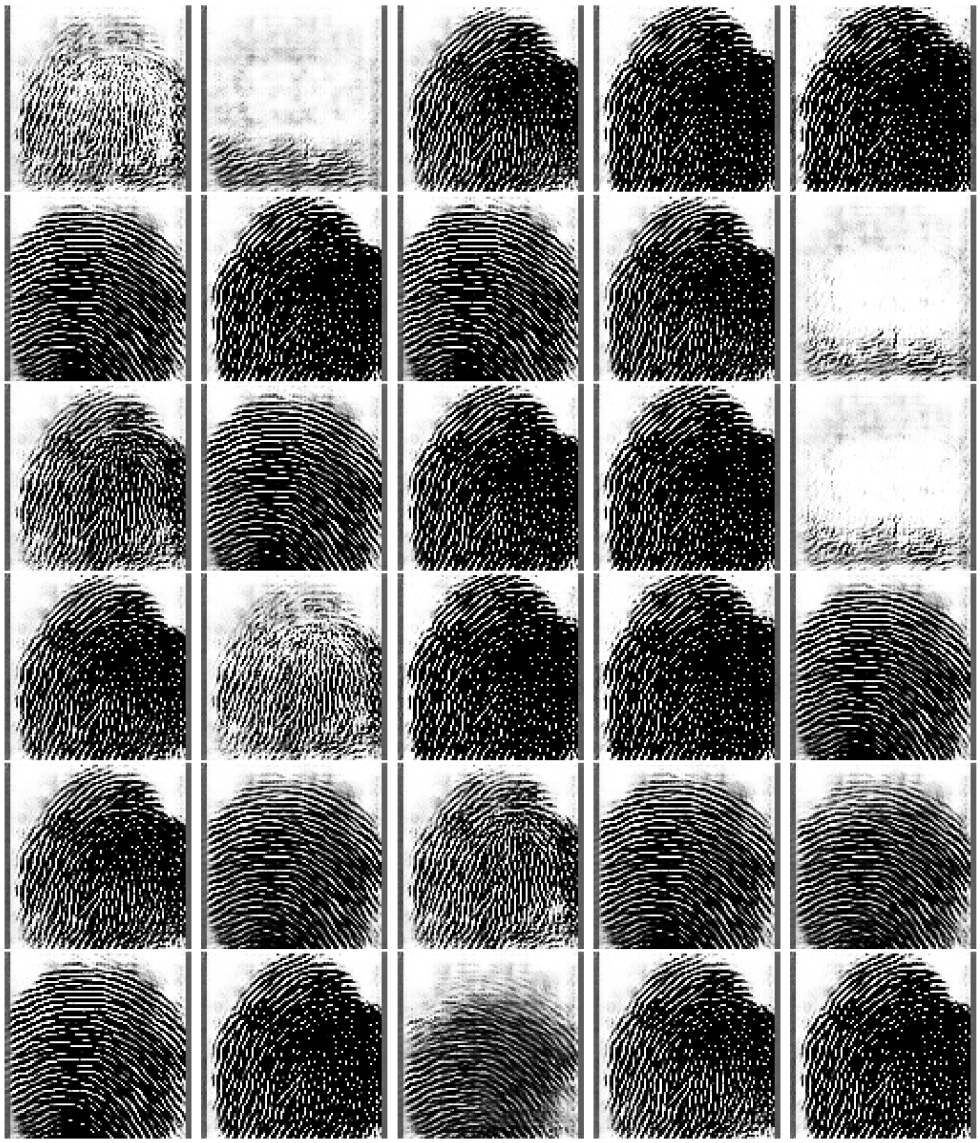


Figure C.2: The generated fingerprint images by the DCGAN-EXT model after 100 training epochs. Each sample was generated from a random latent vector. The mode-collapse problem still persists.

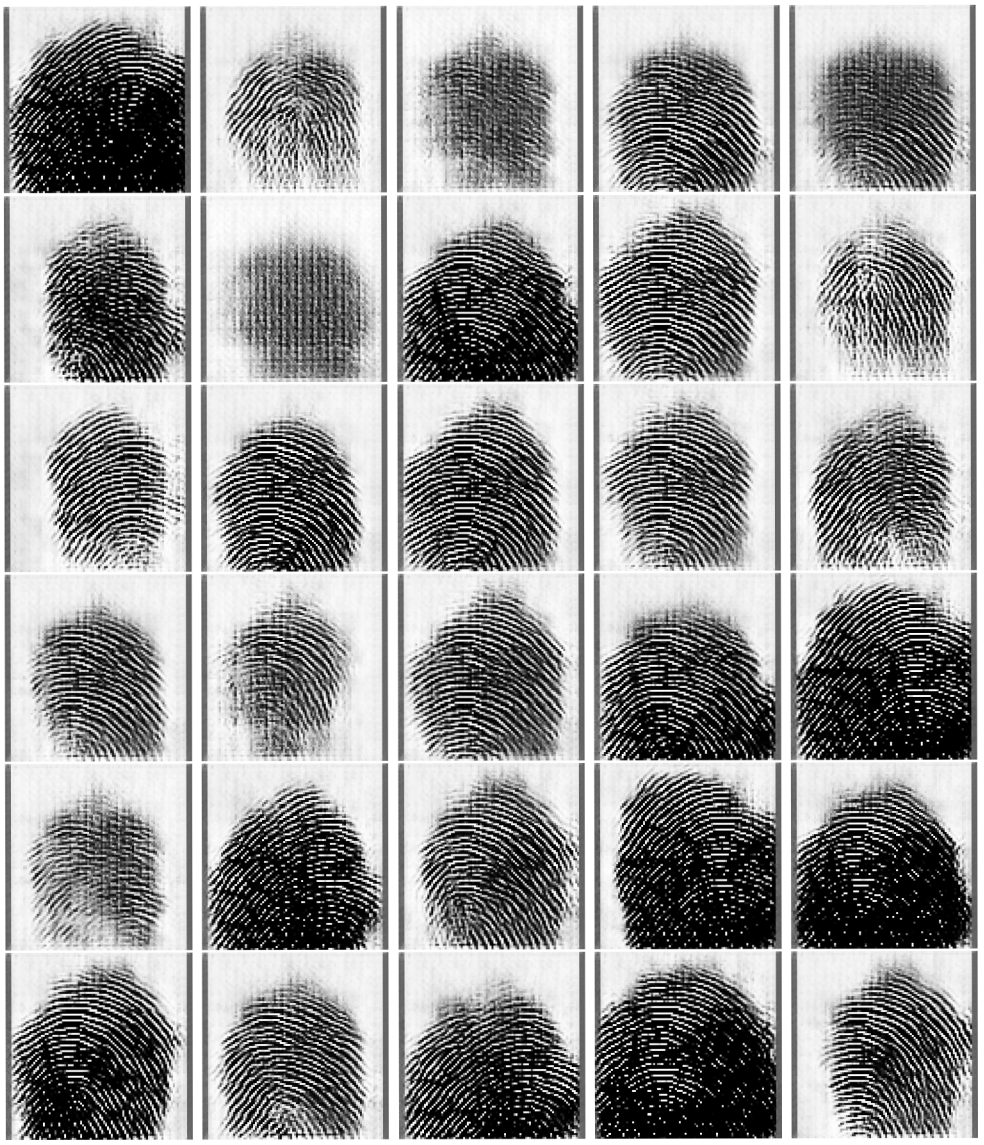


Figure C.3: The generated fingerprint images by the DCGAN-FULL model after 60 training epochs. Each sample was generated from a random latent vector. The implemented methods significantly reduced the mode-collapse.