BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

SIMULATION OF SKIN DISEASES EFFECT USING GAN

SIMULACE PROJEVU KOŽNÍHO ONEMOCNĚNÍ S VYUŽITÍM GAN

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR AUTOR PRÁCE	Bc. ADAM BAK
SUPERVISOR VEDOUCÍ PRÁCE	Ing. ONDŘEJ KANICH, Ph.D.

BRNO 2021

Department of Intelligent Systems (DITS) Academic year 2020/2021

Master's Thesis Specification

24097

Student:	**Bak Adam, Bc.**
Programme:	Information Technology Field of study: Intelligent Systems
Title:	**Simulation of Skin Diseases Effect Using GAN**
Category:	Image Processing

Assignment:

1. Study the literature regarded to use of fingerprints in biometric systems, focus on synthetic fingerprint generation. Study the damage caused by skin diseases on the fingerprint. Acquaint yourself with different types of Generative Adversarial Networks (GANs) used for image generation.
2. Propose a method using GANs to generate synthetic fingerprints with symptoms of the skin diseases. Use available database with fingerprints influenced by skin diseases.
3. Implement the proposed method from the previous point.
4. Create a dataset of minimally 200 synthetic fingerprints for each simulated disease, evaluate their quality using several fingerprint quality measurement methods.
5. Summarise and discuss achieved results.

Recommended literature:

- Maltoni, D., Maio, D., Jain, A.K. and Prabhakar, S.: *Handbook of Fingerprint Recognition*. Springer, 2009, p. 512. ISBN 978-1-8488-2254-2.
- Minaee, S., Abdolrashidi, A.: *Finger-GAN: Generating Realistic Fingerprint Images Using Connectivity Imposed GAN*, Preprint, 2018.
- Drahanský, M.: *Hand-Based Biometrics: Methods and technology*, IET 2018, p. 430, ISBN 978-1-78561-224-4.

Requirements for the semestral defence:

- Items 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, 2020
Submission deadline:	May 19, 2021
Approval date:	November 11, 2020

Abstract

The aim of this master's thesis is to generate a dataset of synthetic fingerprint images that display symptoms of skin disease. The thesis deals with damage caused by skin disease in the fingerprint images and synthetic fingerprint generation. The diseased fingerprints are generated using a model based on Wasserstein GAN with gradient penalty. A unique diseased fingerprint database created at FIT BUT was used for training of the GAN model. The model was trained on three types of skin disease: atopic eczema, psoriasis vulgaris and dyshidrotic eczema. The generator network of the trained WGAN-GP model was used to generate datasets of synthetic fingerprint images. The synthetic images were compared with real fingerprint images using the NFIQ and FiQiVi quality assessment tools and by comparing minutiae location and minutiae orientation distributions in the fingerprint images.

Abstrakt

Cieľom tejto diplomovej práce je vygenerovanie datasetu syntetických snímkov odtlačkov prstov, ktoré vykazujú známky kožných ochorení. Práca sa zaoberá poškodením spôsobeným kožnými ochoreniami v odtlačkoch prstov a generovaním syntetických odtlačkov prstov. Odtlačky prstov s prejavom kožných ochorení boli generované s využitím modelu založeného na Wasserstein GAN s penalizáciou gradientu. Na trénovanie GAN modelu bola použitá unikátna databáza odtlačkov prstov s prejavom kožných ochorení vytvorená na FIT VUT. Daný model bol trénovaný na troch typoch kožných ochorení: atopický ekzém, psoriáza a dyshidrotický ekzém. Sieť generátoru z natrénovaného WGAN-GP modelu bola použitá na vygenerovanie datasetov syntetických odtlačkov prstov. Tieto syntetické odtlačky boli porovnané s reálnymi odtlačkami s využitím NFIQ a FiQiVi nástrojov na určenie kvality spoločne s porovnaním rozložení lokácií a orientácii markantov v snímkoch odtlačkov prstov.

Keywords

fingerprints, synthetic fingerprints, fingerprint generation, skin disease, generative adversarial network, GAN, convolutional neural networks, Python, PyTorch

Kľúčové slová

odtlačky prstov, syntetické odtlačky prstov, generovanie odtlačkov prstov, kožné ochorenia, generatívne adversariálne siete, GAN, konvolučné neurónové siete, Python, PyTorch

Reference

BAK, Adam. *Simulation of Skin Diseases Effect Using GAN*. Brno, 2021. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Ondřej Kanich, Ph.D.

Rozšírený abstrakt

Odtlačky prstov sú jednou z najznámejších a najrozšírenejších biometrických vlastností používaných na biometrickú identifikáciu osôb. Ich využitie pre rozpoznávanie jednotlivcov sa dátuje k prelomu 19. storočia, keď odtlačky prstov nahradili antropometrické vlastnosti ako spoľahlivejší spôsob identifikácie osôb v kriminálnych vyšetrovaniach. Odtlačky prstov s odstupom času nachádzali čoraz širšie využitie, až sa v súčastnosti stali časťou každodenného života.

Samotné odtlačky prstov sú reprezentáciou štruktúry tvorenej papilárnymi líniami na vnútornej strane končekov prstov. Tvar a štruktúra odtlačku sú dané kombináciou genetického kódu spolu s vplyvom okolnostných podmienok. Odtlačky prstov sú unikátne pre každého jednotlivca a obvykle zostávajú nemenné v priebehu života, pokiaľ nie sú poškodené v následku nejakého zranenia alebo vplyvom kožného ochorenia.

Kožné ochorenia, ktoré sa prejavujú v oblasti končekov prstov negatívne ovplyvňujú proces získavania a rozpoznávania odtlačkov prstov. Existuje široká škála kožných chorôb, ktoré takto škodlivo pôsobia na štruktúru papilárnych línií. Relatívne veľké percento ľudí je aspoň raz počas svojho života postihnutých nejakou formou kožných chorôb. Časť populácie dokonca trpí chronickými príznakmi, keďže veľa z týchto ochorení je charakterizovaných fázami zhoršenia a zlepšenia. Takýmto ľudom môže byť na dlhšie obdobia znemožnené používať biometrické systémy založené na odtlačkoch prstov, keďže bežné senzory od-tlačkov prstov typicky nie sú dostatočne vybavené na to, aby si poradili s odtlačkami, ktoré prejavujú známky kožných chorôb. Problém s vývojom senzorov a algoritmov, ktoré by si vedeli s týmito problémami poradiť je potreba prístupu k dostatočne veľkým databázam odtlačkov prstov s prejavom kožných ochorení na ich testovanie. Zozbieranie databáz bežných odtlačkov prstov je nielen pracné a časovo náročné, ale zároveň prináša problémy ohľadom ochrany osobných dát jednotlivcov. V prípade odtlačkov s prejavom kožných ochorení je situácia o toľko zhoršená tým, že je potrebné odtlačky zbierať pod dozorom zdravotníckych pracovníkov a taktiež je potrebné nájsť dostatočný počet dobrovoľníkov, ktorí trpia daným kožným ochorením.

Táto diplomová práca sa zameriava na spôsob vygenerovania syntetických odtlačkov prstov, ktoré budú prejavovať rovnaké známky ochorenia ako reálne odtlačky prstov. Používa pri tom databázu odtlačkov prstov s prejavom kožných ochorení predošle zozbieranú na FIT VUT. Generovanie takýchto syntetických odlačkov prstov by potom následne mohlo buď priamo nahradiť databázy reálnych odtlačkov za účelom ochrany biometrických a zdravotných údajov dobrovoľníkov, alebo slúžiť ako spôsob rozšírenia existujúcich databáz pre rozsiahlejšie testovanie senzorov a algoritmov.

Generovanie takýchto syntetických odtlačkov prstov je dosiahnuté s využitím tzv. Generative Adversarial Networks (GANs). Tieto modely strojového učenia sú inšpirované teóriou hier a fungujú na princípe vzájomného súťaženia dvoch konvolučných neurónových sietí nazývaných generátor a diskriminátor. Práca obsahuje návrh modelu GAN siete založenej na Wasserstein GAN s penalizáciou gradientu (WGAN-GP). Model tejto siete je vhodný na trénovanie na malých datasetoch, pretože sieť diskriminátoru využíva Wasserstein vzdialenosť na meranie rozdielu pravdepodobnostných rozložení reálnych a generovaných dát. Trénovanie takejto GAN je menej citlivé na pretrénovanie siete diskriminátoru a vedie na stabilnejšie učenie. Model navrhnutej siete ďalej využíva techniku dropout, aby sa dokázal naučiť robustnejšie črty. Taktiež používa spektrálnu a batch normalizáciu v sieťach diskriminátora a generátora pre stabilizáciu učenia.

Tento model WGAN-GP siete bol následne implementovaný a trénovaný na troch datasetoch odtlačkov prstov s prejavom kožných ochorení. Vybrané kožné choroby boli atopický ekzém, psoriáza a dyshidrotický ekzém. Trénovanie modelu pokračovalo, kým model nedosiahol rozumné výsledky alebo neprestal dosahovať viditeľné zlepšenia v generovaných snímkoch. Natrénovaná sieť generátoru sa následne použila na vytvorenie nových synteticky vygenerovaných datasetov pre jednotlivé ochorenia.

Synteticky vygenerované odtlačky prstov boli relatívne úspešné v napodobnení odtlačkov s prejavom atopického a dyshidrotického ekzému. V prípade psoriázi sa model nedokázal naučiť generovať reálne vyzerajúce snímky odtlačkov. Tieto syntetické odtlačky prstov boli následne porovnané s reálnymi odtlačkami ovplyvnenými kožnými ochoreniami. Porovnávalo sa rozloženie a orientácia markantov v jednotlivých obrázkoch odtlačkov a s využitím nástrojov NFIQ a FiQiVi bola určená kvalita daných odtlačkov prstov.

V závere práce je zhodnotená úspešnosť použitej metódy generovania odtlačkov s prejavom kožných ochorení a ďalšie prípadné možnosti potenciálneho vylepšenia kvality generovaných snímkov.

Simulation of Skin Diseases Effect Using GAN

Declaration

I hereby declare that this master's thesis was prepared as an original author's work under the supervision of Ing. Ondřej Kanich, Ph.D. All the relevant information sources, which were used during preparation of this thesis, are properly cited and included in the list of references.

. . . .
Adam Bak
May 18, 2021

Acknowledgements

I would like to thank my thesis supervisor Ing. Ondřej Kanich, Ph.D. for his advice, guidance and encouragement he provided me while working on this thesis.

Contents

Chapter 1

Introduction

Fingerprints are one of the most widely deployed biometric characteristic in biometric systems. The practice of using fingerprints for personal identification has been in use since the 19. century. Following the work of Francis Galton [22], fingerprints started to be used as a means of identifying people in criminal investigations. Ever since then, fingerprint technologies have been steadily evolving and have now become commonplace in everyday society.

Fingerprint is the representation of the structure formed by papillary lines at the tip of a finger. It is widely accepted that an individual's fingerprint is unique and remains relatively unchanged throughout the course of his life. However, the structure of these papillary lines can be changed and damaged by skin disease. Fingerprint recognition in biometric systems heavily relies on the structure of these papillary lines to determine a person's identity. The influence of skin disease is an important, but oftentimes neglected factor in fingerprint biometric systems. An individual might be prevented from using certain biometric systems when suffering from a skin disease which affects the fingertips.

To be able to develop fingerprint technologies which are able to asses the effects of skin disease on fingerprints and deal with the resulting damage, it is necessary to have access to large amount of diseased fingerprint data. Collecting a database of fingerprints influenced by skin disease is a particularly difficult task. Not only is it expensive and time-consuming, but it also requires the assistance of medical professionals and ability to find willing participants that suffer from a variety of skin conditions.

The goal of this thesis is to attempt to generate a dataset of synthetic diseased fingerprint images, which could be used in place of a real dataset. Generative adversarial networks were chosen as the means of generating these synthetic fingerprints, as they are a relatively new and powerful tool used for image processing and computer vision tasks. Generative adversarial networks are a type of deep generative machine learning model which rose to prominence in the field of image synthesis ever since their introduction in Goodfellow et al. (2014) [26].

The following Chapter 2 provides a brief introduction to the fingerprint as a biometric characteristic and its use in biometrics. It focuses on the basic structural aspects of a fingerprint image and presents some of the already well-established methods for synthetic fingerprint generation. Chapter 3 details the anatomy of the skin and the formation of papillary line structure in the fingertips. This chapter also covers the manner in which the structure of the skin is affected by certain types of skin disease and the different types of damage observed in in the papillary line structure of afflicted fingerprints. As well as, the overview of the diseased fingerprint database created by Faculty of Information Technology at Brno University of Technology. Chapter 4 contains information about convolutional neural networks, their architectural aspects and their relation to processing 2-dimensional image data. Chapter 5 then subsequently covers how the generative adversarial networks (GAN) utilize convolutional networks for synthetic image generation. This chapter explains the architecture of GANs, their training and introduces a number of different types of GAN used for image generation. The following two Chapters 6 and 7 document the design and implementation details of a GAN model for diseased fingerprint generation, which is based on a Wasserstein GAN with gradient penalty. Lastly, Chapter 8 describes the training process of the proposed model, generation of datasets consisting of diseased fingerprint images and quality evaluation and assessment of the generated images in comparison to real fingerprint images affected by skin disease.

Chapter 2

Fingerprints in Biometrics

The ability to uniquely identify individuals and associate personal attributes (e.g. name) with an individual has proved to be crucial in human society [30]. The practice of using fingerprints to identify individual people has been in use for centuries. Fingerprints are one of the most well-known and commonly used biometric characteristics of a human body for recognizing individuals.

This chapter presents a brief overview of the use of fingerprints in biometric systems. It details some of the important properties of biometric characteristics and their relation to use of fingerprints in biometrics. The chapter also covers some relevant structural aspects of the fingerprint images. And lastly, it focuses on synthetic fingerprint generation and some possible methods of creating realistic looking synthetic fingerprint images.

2.1 Biometric Systems

Biometric recognition (or biometrics) refers to the use of distinctive anatomical and behavioral characteristics, called *biometric characteristics* for automatically recognizing individuals. *Biometric systems* are systems that employ these *biometric characteristics* (e.g. fingerprints, palmprint, iris, retina etc.) to be able to uniquely identify individual people. [30][41]

Biometric systems find use in large-scale identity management systems, where it is crucial to be able to accurately determine (or verify) individuals identity. Identity of an individual in these systems is represented by all the available information in the identity management system associated with that particular person. A reason for trying to uniquely distinguish individuals in such a system might be to limit the individual's access to sensitive resources. [31]

There are other ways of establishing a person's identity other than using biometrics. You can use knowledge-based (e.g. password) or token-based (e.g. keycard) mechanisms for identifying individual people. However, these surrogate identity representations come with certain disadvantages, they can be easily lost, stolen, shared, replicated etc. All of these issues would allow unauthorized individuals access to the systems. [30][31]

The advantage of using biometrics is that they increase the security of the systems. They are not forgettable, transferable or easily lost and can be very difficult to replicate. Biometrics also discourage fraud and eliminate repudiation claims (denial of using the system by the user). [30]

2.1.1 Biometric Characteristics

Contents of this subsection are taken from [31][41]. Biometric characteristics (traits) are attributes of human anatomy or behavior that can be quantified and used to determine identity of individual people. These biometric characteristics should represent something inherent to a human being which cannot be easily replicated or shared between individuals. They enable the biometric systems to distinguish individuals based on "who they are" instead of "what they posses". There are several basic features used to determine the suitability of a biometric characteristic:

- *Universality*: every individual attempting to access the system should be in possession of this biometric characteristic.
- *Uniqueness* (distinctiveness): the biometric characteristic should be sufficiently different between any two given individuals so that is is possible to identify them uniquely.
- *Permanence*: determines how well a particular characteristic resists change over time. A biometric characteristic should sufficiently invariant with respect to the matching criteria.
- *Measurability* (collectability): any biometric characteristic should be able to be measured quantitatively. Measurability determines how easy it is to acquire and digitize given biometric characteristic.
- *Performance*: a set of metrics that determine suitability of a biometric characteristic. These include recognition accuracy, speed, resource requirements and robustness.
- *Acceptability*: willingness of individual users to present a given biometric trait to the system for measurement.
- *Circumvention*: ease with which a given biometric characteristic of an individual can be replicated and employed by unauthorized users to circumvent the system.

Every biometric characteristic has its own strengths and weaknesses and no single characteristic is expected to effectively posses all of the above given features. Some biometric characteristics will be more suited to different applications than others. It is also possible to utilize multiple characteristics in conjunction with one another to counteract any downsides of a singular biometric characteristic.

2.2 Fingerprints in Biometrics

Fingerprints are one of the most widely used characteristics in biometrics, so much so in fact that the word fingerprint has become almost synonymous with the word identity. They have been used for personal identification for many decades. Fingerprints are the most commonly used biometric characteristic of the hand [13].

A fingerprint is the representation of the exterior appearance of the *epidermis* of a fingertip. Epidermis is the outermost layer of the skin, this skin exhibits an interleaved, flow-like pattern of ridges and valleys (furrows) on the fingertips called *papillary lines*, see Figure 2.1. Fingerprint is the pattern formed by the structure of these papillary lines. The formation of the papillary lines is given by a combination of the genetic code of the individual and environmental factors. The patterns formed by papillary lines are very distinctive for a given individual, which makes this biometric characteristic rather useful. [13][31][41]

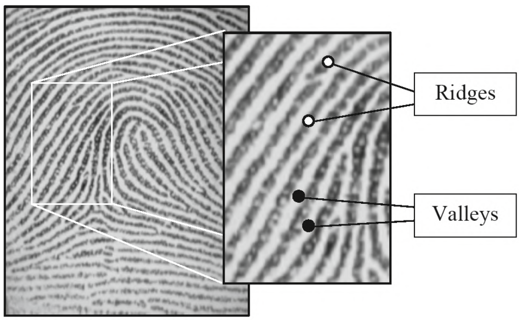


Figure 2.1: Ridges and valleys of a fingerprint image. [41]

The fingerprint, as a biometric characteristic, has a good balance of desirable properties mentioned in Subsection 2.1.1. Almost every human being has fingerprints, with the exception of some hand-related disabilities, missing limbs or extensive skin damage. Fingerprints are very distinctive, patterns formed by the papillary lines are unique for each individual, even in the case of identical twins. Properties of a fingerprint are permanent for a given individual, even if they can change temporarily (e.g. skin disease, cuts, burns) they will eventually heal and revert to their original appearance. As a result of extensive research and funding, fingerprint technologies are becoming continuously more common and affordable. Due to common occurrence of fingerprint systems (e.g. fingerprint sensors in smartphones) the acceptance of an average user to use such systems is relatively high. Although, bear in mind that there might be some stigma associated with fingerprints due to their prevalent use in criminal investigations. Robust biometric systems using fingerprints in conjunction with other security technologies can be prohibitively difficult to circumvent. [13][41]

2.2.1 Fingerprint Classification

Fingerprint classification is the act of assigning a fingerprint image into one of a number of pre-determined classes based on the structural characteristics of the fingerprint. The aim of classifying fingerprints is to serve as an indexing scheme for large databases of fingerprints, where blindly searching for a fingerprint match would be unfeasable. If a fingerprint is of a particular class then it is possible to discard all fingerprints that do not share this class from the search. [65][41]

Fingerprint classification is based on the original *Henry's classification scheme*, which separated fingerprints into five major classes, see Figure 2.2. Fingerprints are separated into these pre-determined classes based on the *global pattern configurations* of the papillary lines at the central point of the fingerprint. These global patterns of configuration are called singular points (regions) which can be broadly separated into two types Figure 2.3. [65][41]

The first one is the *core*. Cores are generally at the very center of a fingerprint, although not necessarily at the center of a fingerprint image. The core is defined as the "north most" point of the inner most ridge line and is usually part of the inner most loop or a whorl [41]. If a fingerprint does not contain a loop or a whorl (e.g. arch fingerprint) then the core is defined as the region of maximal curvature in the fingerprint.

The second one is the *delta*. Delta is a location in a fingerprint where the papillary lines run in three directions [13]. Subsequently, the fingerprint class can be determined using the number of these cores and deltas and the spatial relation between them.

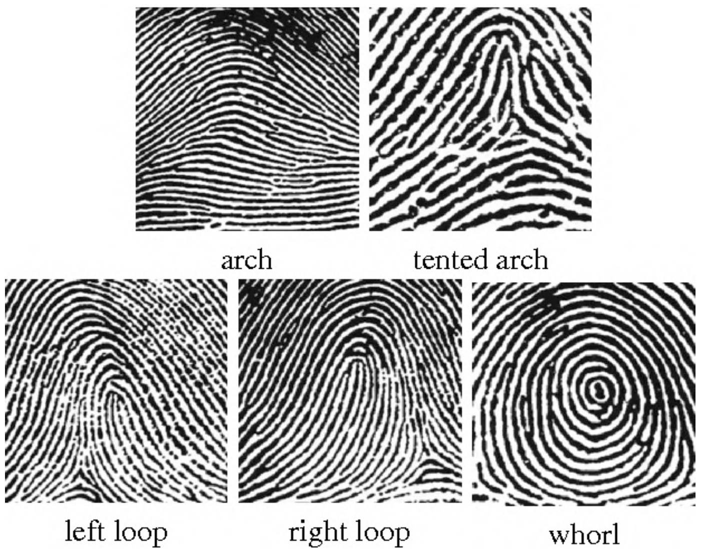


Figure 2.2: Examples of fingerprint images representing each of the five major fingerprint classes. [65]

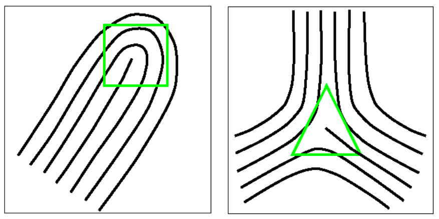


Figure 2.3: Simplified structure of the two main types of a singular point in a fingerprint image – core and delta. [65]

2.2.2 Minutiae

The singular points mentioned in the previous subsection are typically not sufficient for correctly identifying individuals. To properly distinguish fingerprints a set of local structural formations called *minutiae points* are used. Minutia refer to the various ways in which the ridge line of a fingerprint can be discontinuous [41].

There are a number of different basic minutiae which can appear in the structure of a fingerprint. The two that most commonly used in automated fingerprint processing are *line ending* and *bifurcation*. This is mostly to reduce the difficulty of accurately determining the type of minutiae from the fingerprint image. Line ending, as the name suggests, is a location in the fingerprint where the ridge line comes to an end. Bifurcation, on the other hand, is a location in a fingerprint where are single ridge line splits (divides) into two ridge lines. Other than the line ending and bifurcation, there is a wide range of basic minutiae types, some of these are shown in Figure 2.4. [41]

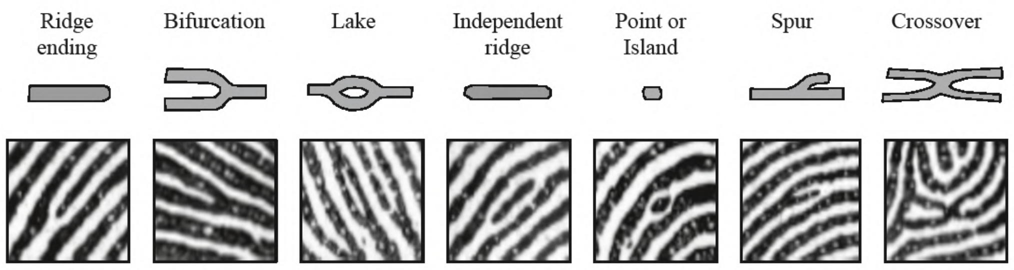


Figure 2.4: An example of seven different select types of minutiae. [41]

2.3 Synthetic Fingerprint Generation

Synthetic fingerprint generation is the process of creating synthetic fingerprint images that imitate fingerprint images acquired by real on-line sensors. One of the reasons for using synthetic fingerprint images in place of real fingerprint images is that the process of acquiring a large enough database of fingerprints can be challenging in some cases. Collecting a large database of fingerprints is cost-prohibitive, time-consuming, tiresome, moreover, there are privacy concerns and legislation issues associated with the use of personal biometric data [9][13][41].

Attempting to collect a database of fingerprint images influenced by skin disease is one such case, where all of these drawbacks are amplified and is one of the main motivations behind this work. It is difficult to collect diseased fingerprint data as you need to employ help of medical experts as well as find willing participants with the specific skin conditions.

Generation of synthetic fingerprint images presents an alternative way to create such collections of data quickly, without privacy issues and at minimal costs [13]. It also makes it possible to tailor the created dataset according to specific needs.

For example, the fingerprint classes mentioned in Subsection 2.2.1 do not occur in human individuals with equal frequency, some are more prevalent than others. There is a possibility that when collecting fingerprint data affected by a specific disease, a few of these fingerprint classes will be underrepresented. Such a dataset might cause issues if it was to be used further, e.g. in machine learning applications. Generation of a synthetic fingerprint dataset could solve this problem, as it is possible to create a dataset containing balanced representation of fingerprint classes.

2.3.1 SFinGe Method

*The **s**ynthetic **fin**gerprint **ge**neration* (SFinGe) is a state-of-the-art fingerprint synthesis algorithm. The SFinGe method was first developed at the university of Bologna in [8] and later improved in [9]. It is currently one of the oldest and most well known methods for generation of realistic looking synthetic fingerprint images.

The simplified idea of the algorithm is that it "inverts" some of the operations used in the process of fingerprint recognition, see Figure 2.5. This method separately generates a random fingerprint area, orientation image and frequency image. These are then assembled together to create a *master-fingerprint.* [41]

The SFinGe algorithm takes fingerprint class, image size, region of interest and singular points as input. SFinGe then generates an orientation field based on modified zero-pole model according to singular points and fingerprint type. Subsequently, Gabor filters are

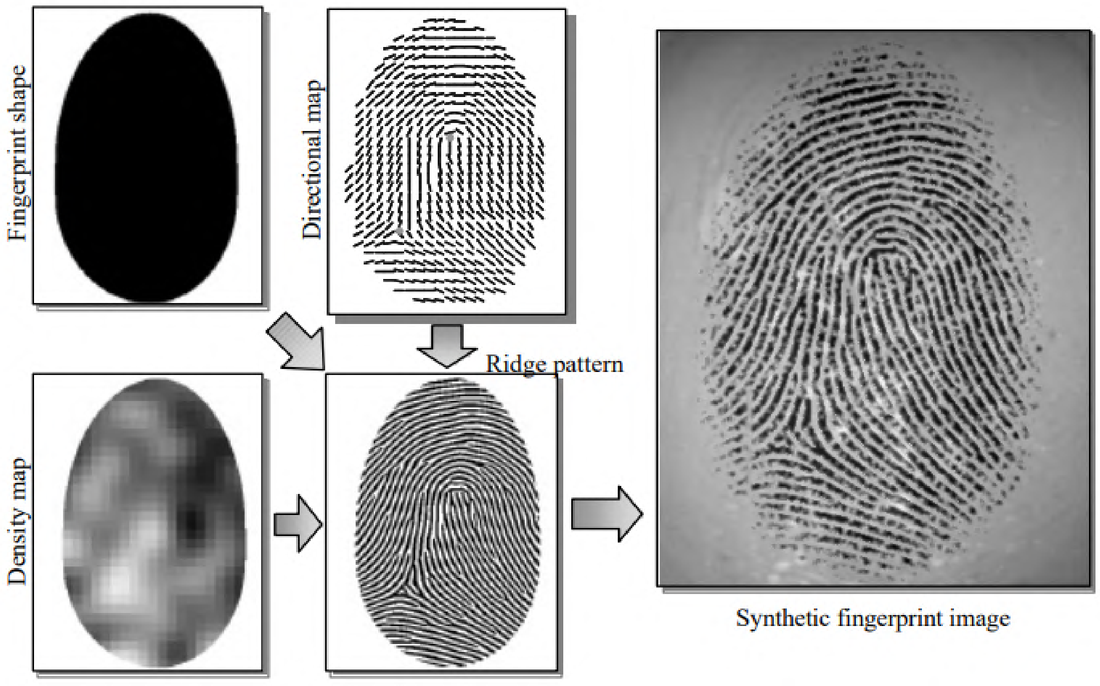


Figure 2.5: The SFinGe method of generating a master-fingerprint representation which can be subsequently used for creation of synthetic fingerprint image versions of the master-print. [9]

applied to a seed image to generate the master-print from based on the orientation field and ridge frequency. [9]

The master-print is and ideal representation of the fingerprint structure. It is further used with a combination of noise and deformities specific to fingerprint sensing to generate a number of different versions of the same fingerprint that look as thought they are produced by a real fingerprint sensor [9].

However, even though the fingerprint images produced by SFinGe are high-quality and look very realistic, there are certain drawbacks associated with the method [9]. First of all, the ridge thickness is constant throughout the entirety of the generated image. This is not necessarily the case in real fingerprint images as the ridge thickness may differ in certain areas of the image.

Secondly, the random noise is distributed uniformly across the generated image. Whereas, both low and high quality regions can be present in a real fingerprint image [41].

Thirdly, the noise generated as a substitution for the intra-ridge noise is random in different representations of the same master-fingerprint. Intra-ridge noise is a very fine level noise caused by natural structure of the skin (e.g. finger pores) and there should be some consistency kept in mind across multiple representations of the same finger.

Lastly, the spatial and orientational distribution of minutiae in SFinGe generated fingerprint images was shown to differ from spatial and orientational distribution of minutiae in real fingerprints [13].

2.3.2 Synthesis based on Statistical Feature Models

The method based on statistical feature models was first introduced in [63]. In this approach to generation of fingerprints, the goal is to synthesize a fingerprint image from a number of prespecified features (e.g. minutiae, orientation field). This model is meant to improve upon the SFinGe model by being able to control the number and location of minutiae in the generated image. [63]

The basic idea of the algorithm is that it samples a number of fingerprint features from random distributions which appear as though they belong to real fingerprints. Subsequently, a fingerprint reconstruction algorithm Feng and Jain [21] is applied to these sampled features to create a synthetic master-print. This master-print is then used to create a varied range of fingerprint impressions by adding distortion and noise.

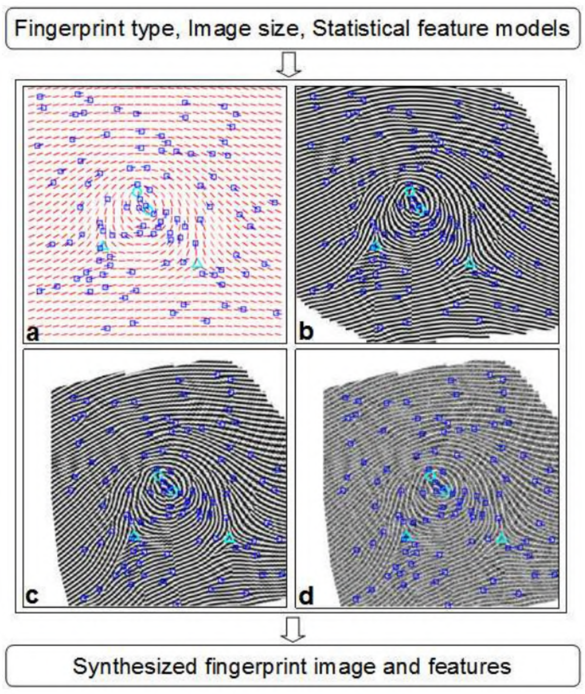

Fingerprint type, Image size, Statistical feature models
a
b
c
d
Synthesized fingerprint image and features

Figure 2.6: Four steps of the fingerprint synthesis method based on statistical feature models: (a) Feature sampling of minutiae, singular points and orientation field from an appropriate feature model (b) Generating master-fingerprint. (c) Generating multiple fingerprint impressions from the master-fingerprint using distortion effects. (d) Rendering synthetic fingerprint image by simulation of finger dryness and addition of noise. [63]

To be able to randomly sample features that will appear in the finalized synthetic fingerprint, five statistical distribution models were created. One model for each of the five major fingerprint classes. These models contain information about the singular points, ridge orientation field, and minutiae of the fingerprint classes. The statistical models are used to more evenly distribute features of the generated fingerprint, to be more in line with it's specified fingerprint class.

One of the drawbacks of this algorithm is that some of the feature information can get lost during the fingerprint reconstruction process. Therefore, some of the sampled features might not be present in the final synthetic fingerprint image.

Chapter 3

Influence of Skin Disease on Fingerprints

A *skin disease* (skin condition) is a medical condition that affects the integumentary system. Integumentary system being the skin and it's appendages. Due to the sheer amount of different skin conditions, it is difficult to determine the percentage of population affected by skin disease during the course of their lives. Part of the affected population suffers from chronic health effects, as there is a number of skin diseases last for a long time and are characterized by periods of exacerbations and remissions. [32]

The presence of skin disease symptoms in the fingertips negatively affects the process of fingerprint recognition and causes problems in biometric systems. Fingerprint sensors are typically not equipped to handle fingerprints that display symptoms of a skin disease. If the quality of fingerprint images is low due to extensive damage caused to the patterns of papillary lines, the affected user might be prevented from using fingerprint biometric systems altogether. [13]

This chapter covers the anatomy of the skin, symptoms of and damage caused by some of the more common types of skin disease. It also details the collection and analysis of the database of diseased fingerprints created by Faculty of Information Technology at Brno University of Technology [3][13][14].

3.1 Anatomy of the Skin

The skin is the largest organ of the body, accounting for about 15% of an adult's body weight. It is a barrier organ that separates the body from the outside environment. It protects the body from microorganisms, regulates body temperature, and facilitates sensory inputs. The skin consists of three layers:

- The *epidermis* is the continually renewing outer layer of the skin. It is the thinnest layer of the skin, it varies in thickness from 0.04 mm to 1.60 mm . The function of the epidermis layer is to insulate and protect the rest of the body from mechanical and environmental damage as well as prevent water loss.

  The epidermis consists of four basic types of cells – keratinocytes, melanocytes, Langerhans cells, Merkel cells. The keratinocytes produce protein keratin, protect tissue from heat. microbes and chemicals. The melanocytes produce pigment called melanin, which protects the skin from UV light and colors the skin. The Langerhans cells rise to epidermis from red bone marrow and provide immune response against microbes in the skin. The Merkel cells are located in the deepest layer of the epidermis and their function is to provide touch sensation

The most unique feature of the epidermis is the *stratum corneum*. The stratum corneum is the outermost part of the epidermis, it consists of a layer of dead cells - keratinocytes (or corneocytes). Keratinocytes are created in lower layer of epidermis and slowly migrate towards the stratum corneum. These cells are periodically shed and replaced, they provide protection to the underlying cells.

Epidermis of the hands, fingers, soles of feet and toes exhibits a flow-like pattern of papillary lines (friction ridges). The formation of the papillary lines of the epidermis is closely related to the structure of dermal papillae at the junction of epidermis and dermis. [13] [37] [42]

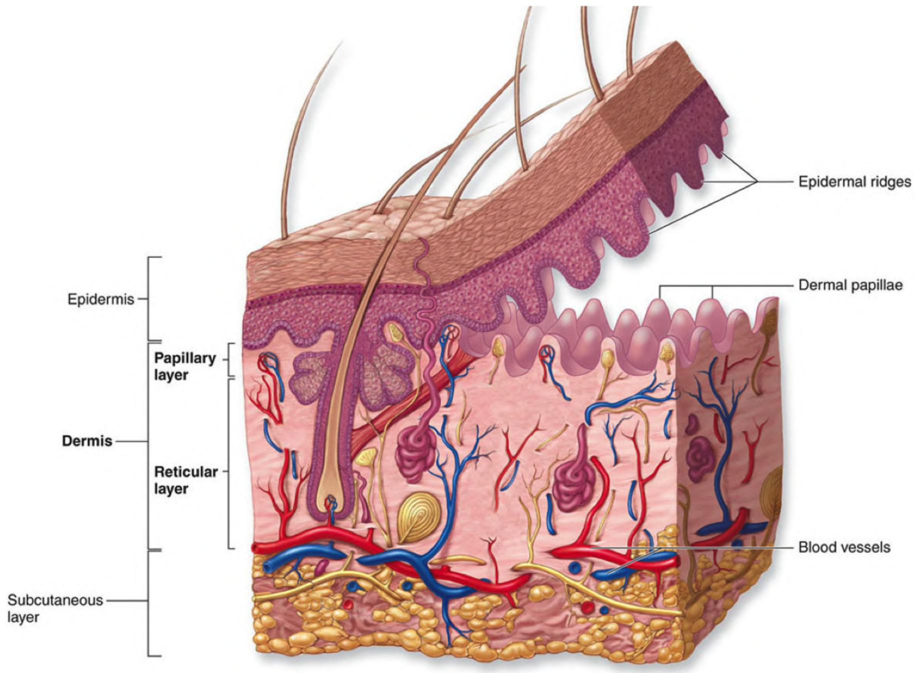


Figure 3.1: Cross-section of the skin and subcutaneous fat layer. [67]

- The *dermis* is the middle layer of the skin located between epidermis and subcutaneous tissue. The thickness of dermis ranges from 1 mm to 4 mm. It is $15 - 40$ times thicker than the epidermis, depending on the location (it is thickest on the back).

  The dermis is comprised mainly of connective tissue containing collagen and elastic fibers. The dermis represents the bulk of the skin structure and provides skin's pliability, elasticity, tensile strength and thermal regulation [37]. There are sensory receptors located inside the dermis.

  The basic cells present in the dermis include fibroblast cells, macrophages and mastocytes. The fibroblast cells provide structural framework to tissues of the skin, they synthesize proteins and are a critical part of healing wounds. The macrophages are a type of white blood cells, they enter the dermis as a response of the immune system to stimuli. The mastocytes are migrant cells of the connective tissue involved in wound healing and immune response.

The *papillary dermis* is the uppermost part of the dermis, it a thin zone immediately below the epidermis. This layer is uneven and contains fingerlike protrusions into the epidermis called *dermal papillae.* These are responsible for the formation of papillary ridges on the surface of the epidermis layer. [13][37][42]

- The *subcutaneous layer* (hypodermis) is a layer of fat cells directly below the dermis layer. Subcutaneous layer is attached to the dermis by collagen and elastin fibers. The function of the subcutaneous layer is to insulate the body and serve as an emergency energy supply. Although it is not truly part of the skin (hypodermis means “beneath the skin”), they are so closely related that the subcutaneous layer is considered part of the skin in pathological processes. [13]

3.2 Damage Caused by Skin Disease

A *skin disease* is a health conditions which affects the integumentary system. Certain skin diseases can be localized on the fingertips or affect fingertips as a part of a larger systemic condition [13]. Such a disease can cause substantial damage to the skin structure of the fingertip, adversely affecting the quality of fingerprint images acquired from the disease-affected finger.

If a skin disease has affected and destroyed the structure of papillary lines in the epidermis and attacked the papillary dermis at the dermoepidermal junction, the papillary lines will regrow in different patterns. In some cases the papillary lines might not regrow at all and such damage caused by the disease is permanent. [13][14]

Common symptoms of skin disease include inflammation, dryness, fibrosis (increase of fibrous connective tissue), atrophy, discoloration, pruritus (itching).

3.2.1 Histopathological Changes of the Epidermis and the Dermis

This type of disease causes problems for most types of fingerprint sensors. They typically change the structure and color of the skin.

Atopic Eczema

Atopic eczema (atopic dermatitis) is a chronic, inflammatory skin disease. It is characterized by dry, cracked, scaly skin and itching.

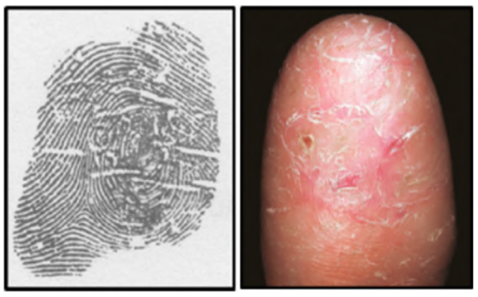
The cause of the disease is unknown, although it is believed that genetic defect predisposes patients to the development of this disease [32]. The condition may occur at any age, but the first symptoms typically appear in childhood. Atopic eczema has phases of exacerbations and remissions. Atopic eczema is very similar to other types of hand eczema, most common being irritant contact dermatitis and allergic contact dermatitis. [13][32][28]

Atopic eczema can cause medium to major damage to the fingerprint. Fingerprints from people with atopic eczema presented with a collection of thin lines, which can cross over one another in different directions, accompanied by a scarce amount of small white spots. [13]

Dyshidrotic Eczema

Dyshidrotic eczema (pompholyx) is a vesicular hand and foot dermatitis. Moderate to severe itching precedes the appearance of vesicles. The vesicles are small, fluid-filled sacs that appear on the palms and sides of fingers. Other symptoms include redness and perspiration. [28]

Dyshidrotic eczema usually covers the fingertip area with irregular blurred shapes, which look like groupings of white spots on the fingerprint image. In addition to the white spots, the fingerprint image typically exhibits a number of thick white lines in the papillary structure. The extent of damage caused to the fingerprint images varies from medium to major depending on the severity of the disease. [13]


(a)

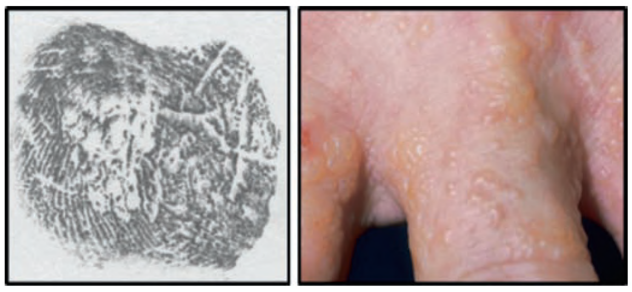

(b)

Figure 3.2: Diseased fingerprints of (*a*) atopic eczema and (*b*) dyshidrotic eczema. [3][13]

3.2.2 Histopathological Changes at the Dermoepidermal Junction

These diseases cause damage even beneath the epidermis, at the dermoepidermal junction. This is the place, where the structure of papillary lines is formed. Damage to this area of the finger causes issues even for ultrasonic fingerprint scanners, as this is the region of the skin where they acquire fingerprints. [13]

Psoriasis Vulgaris

Psoriasis vulgaris is a common, chronic and recurrent inflammatory disease of the skin. It is typically indistinguishable from a very serious form of hand eczema. Psoriasis is characterized by circumscribed, dry, scaling plaques of various sizes. The lesions are typically covered by silvery white scaly skin. [13][14][32]

This disease causes extensive damage to the fingerprints, see Figure 3.3. Most of the fingerprint images acquired from fingertips affected by this disease are completely unusable. A frequent feature of this disease is a large irregular dark spot with a white border. Additionally, dark areas, thickened papillary lines, round spots and oblong spots can be observed. [15]

Verruca Vulgaris

Verruca vulgaris (common warts) begin as smooth, flesh colored papules and evolve into dome-shaped, gray, hyperkeratonic growths with black dots on the surface. The hands are the most common surface on which the warts appear. They are usually few in number, but can become so numerous that they cover large areas of exposed skin. [28]

The extent of damage to the fingerprint image is rather minimal, see Figure 3.3. Outside the areas obscured by warts, the papillary lines flow in the usual way. The papillary lines are only deformed in the area immediately next to the wart. However, the warts can, in some cases, cluster together and obscure larger areas of the fingerprint. [13][28]

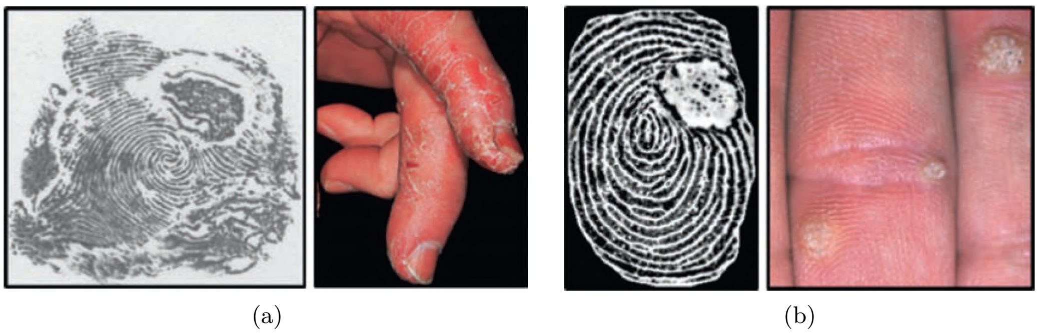


Figure 3.3: Diseased fingerprints of (*a*) psoriasis and (*b*) verruca vulgaris (warts). [3][13]

3.2.3 Skin Discoloration

Skin discoloration, in and of it self, does not affect the papillary structure in any way. The discoloration of the digits may, however, cause problems for optical fingerprint sensors and fingerprint sensors which use color or spectral analysis of the fingertip skin for antispoofing detection. [13]

Raynaud's Phenomenon

Raynaud's phenomenon usually occurs in together with collagen vascular disease or scleroderma. The symptom of Raynaud's phenomenon is sequential pallor, cyanosis (blueness) and rubor (redness). The fingers may fail to regain their normal circulation and remain permanently blue and painful. [28]

Raynaud's phenomenon doesn't cause structural damage to the papillary lines, therefore, the fingerprint images appear undamaged, see Figure 3.4. However, the discoloration of the skin may cause other problems for fingerprint sensors as mentioned above. The damage caused is overall rather minor. [13]

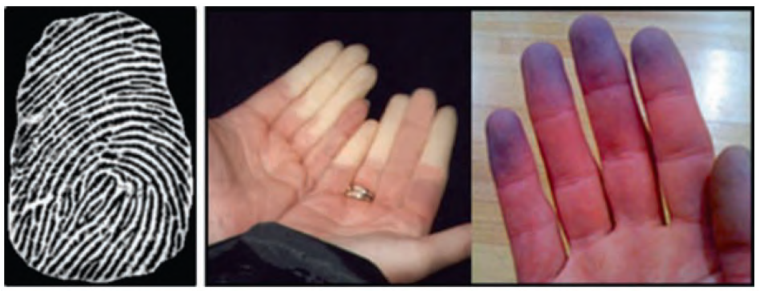


Figure 3.4: Diseased fingerprints of Raynaud's phenomenon. [3][13]

3.3 Database of Diseased Fingerprints

There exists a unique database of diseased fingerprint images created by the Faculty of Information Technology at Brno University of Technology in cooperation with University Hospital Olomouc, St Anne's University Hospital in Brno and a private dermatology clinic in Darmstadt [3][12][13]. To acquire fingerprints from patients suffering from skin disease, medical experts were provided with a set of fingerprint sensors, digital microscope and dactyloscopic cards. The fingerprint sensors included a 3-D touchless and touch optical sensors, sweep and touch capacitive sensors. [13]

A sum total of 2 165 fingerprint images from 44 patients were collected for the database. These fingerprints were influenced by 12 different types of skin disease, see Table 3.1. The fingerprints in the database are associated with anonymized information about the patient, type and severity of the disease.

Table 3.1: Contents of the diseased fingerprints database. [13]

Disease	**No. of fingerprints**	**Percentage**	**No. of patients**
Fingertip eczema	1 107	51.132%	17
Psoriasis vulgaris	326	15.058%	9
Dyshidrotic eczema	247	11.409%	4
Hyperkeratotic eczema	118	5.450%	2
Verruca vulgaris	96	4.434%	4
Scleroderma	50	2.310%	1
Acrodermatitis continua	40	1.848%	1
Colagenosis	36	1.663%	1
Raynaud's phenomenon	9	0.416%	1
Effusion of fingers	35	1.617%	1
Cut wound	18	0.831%	2
"Unknown" disease	83	3.834%	1
Total	**2 165**		**44**

The diseased fingerprint database was used to analyze the influence of and damage caused by these skin conditions on the papillary line structure. There exist common features displayed in the fingerprints affected by each particular disease. During the analysis of the database, the damage caused to the fingerprints was divided into two categories of 7 local features and 5 global features. [13][3]

The local features are:

- straight lines (SL)
- a grid (G)
- small papillary lines disruptions (PLD)
- small "cheetah" spots (CS)
- large oblong spots (ROS)
- large irregular spots (IS)
- dark places (DP)

The global features include:

- blurriness of the image (B)
- significantly high contrast of the image (HC)
- entire fingerprint area affected (EA)
- total deformation of the fingerprint image (TD)
- significantly high quality and health of the fingerprint (HQ)

These global and local features were observed in a subset of the fingerprint image database and each disease present in the database was associated with a percentage score depending on the prevalence of given features in fingerprint images affected by that particular disease, see Tables 3.2 and 3.3. These tables detail the type and extent of certain types of fingerprint damage expected to be found in the fingerprint images.

Table 3.2: Global features observed in the fingerprint damage caused by the skin disease. [13]

Disease	**Percentage of particular features**					**Sum**
	B	**HC**	**EA**	**TD**	**HQ**	
Fingertip eczema	18.01	21.50	40.38	36.36	29.02	572
Psoriasis vulgaris	34.86	27.06	61.93	58.72	18.35	218
Dyshidrotic eczema	30.33	30.33	31.97	29.51	9.84	122
Hyperkeratotic eczema	31.37	29.41	9.80	0.00	37.25	51
Verruca vulgaris	19.05	80.95	7.94	7.94	76.19	63
Scleroderma	0.00	0.00	0.00	0.00	100	23
Acrodermatitis continua	48.57	25.71	100	100	0.00	35
Colagenosis	9.38	40.63	0.00	0.00	25.00	32
Raynaud's phenomenon	0.00	0.00	0.00	0.00	100	8
Effusion of fingers	23.33	16.67	40.00	16.67	3.33	30
Cut wound	37.50	68.75	0.00	0.00	50.00	16
"Unknown" disease	30.00	20.00	90.00	83.33	0.00	30

Table 3.3: Local features observed in the fingerprint damage caused by the skin disease. [13]

Disease	**Percentage of particular features**							**Sum**
	SL	**G**	**PLD**	**CS**	**ROS**	**IS**	**DP**	
Fingertip eczema	72.03	24.65	15.91	12.24	32.34	16.61	15.73	572
Psoriasis vulgaris	40.37	6.42	2.75	12.84	48.17	32.57	62.84	218
Dyshidrotic eczema	63.11	7.38	14.75	18.03	78.69	29.51	32.79	122
Hyperkeratotic eczema	3.92	0.00	66.67	15.69	74.51	3.92	5.88	51
Verruca vulgaris	3.17	0.00	14.29	12.70	74.60	0.00	25.40	63
Scleroderma	0.00	0.00	0.00	0.00	0.00	0.00	30.43	23
Acrodermatitis continua	14.29	0.00	0.00	85.71	60.00	14.29	65.71	35
Colagenosis	100	78.13	0.00	0.00	15.63	0.00	25	32
Raynaud's phenomenon	0.00	0.00	100	0.00	0.00	0.00	0.00	8
Effusion of fingers	10.00	0.00	73.33	43.33	63.33	6.67	13.33	30
Cut wound	93.75	0.00	0.00	0.00	18.75	0.00	12.50	16
"Unknown" disease	100	86.67	0.00	0.00	76.67	30.00	73.33	30

Chapter 4

Convolutional Neural Networks

Convolutional neural networks or CNNs, were first established by LeCun et al. (1998) [39]. Convolutional neural networks have since proven to have very successful applications in image processing, they are designed to recognize visual patterns directly from a pixelated image with minimal preprocessing.

Generative adversarial networks used for synthetic image generation typically consist of a pair of deep convolutional neural networks. It is, therefore, important to understand how convolutional networks function in order to understand how generative adversarial networks work to produce synthetic images.

Convolutional neural network is a specialized type of an artificial neural network designed for processing data that has a known, grid-like topology [25]. An obvious example of grid-structured data is a 2-dimensional image. Image data have a strong 2-dimensional local structure, pixels of an image that are spatially close together are highly correlated. Therefore, neighboring pixels tend to have similar color values and are more likely to be a part of some local structure (e.g. edges, corners, etc.). Another important property of image data is its translation and rotation invariance [1], this means that certain parts of the image data have the same interpretation regardless of their position or rotation in an image.

The convolutional neural networks utilize a mathematical operation called *convolution* to extract information about local features of the input. Convolution is usually subsequently followed by application of a non-linear activation which decides whether the presence of a feature is strong enough to warrant an activation of a neuron. This is then followed by a *pooling* which combine the extracted features into a compact representation, which is mostly invariant to moderate changes in scale, translation, pose. [34]

4.1 Convolution

In mathematics, convolution is an operation on two functions $f(x)$ and $g(x)$ with real-valued arguments that produces a third function $h(x)$. Convolution is defined as the integral over the product of both functions $f(x)$ and $g(x)$ after one is reversed and shifted. The convolution operation is commonly denoted by the asterisk symbol $*$. [25]

$$\begin{aligned} h(x) &= (f * g)(x) \\ (f * g)(x) &= \int_{-\infty}^{\infty} f(y)g(x-y)dy \end{aligned} \tag{1}$$

The convolution operation has a wide range of applications in image processing, data processing, probability theory, electrical engineering etc. To better understand convolution, consider this example from probability theory. Take a simple case of rolling two fair, 6-sided dice. Let X and Y be two discrete random variables describing the process of rolling dice and the possible outcomes. Let $f(X)$ and $g(Y)$ be their discrete probability distributions. The probability distribution function of the sum of these two dice rolls $p(n)$ can be computed as the *discrete convolution* of functions $f(X)$ and $g(Y)$ [48]. Discrete convolution is defined as infinite sum:

$$(f * g)(n) = \sum_{i=-\infty}^{\infty} f(i)g(n-i). \tag{2}$$

For functions with a finite support interval, it is possible to take a finite sum instead.

$$(f * g)(n = d_1 + d_2) = \sum_{i=1}^{n} f(i = d_1)g(n - i = d_2) \quad n \in \{2, 3, .., 12\} \tag{3}$$

Think of this convolution as computing all different permutation of rolling a total sum of $n = d_1 + d_2$ on the two dice. Another way to think about it is as though dragging one distribution function over the other and taking their product where they overlap. If you were to compute the probability of rolling e.g. $p(n = 3)$ it would give the result $\frac{2}{36}$, due to there being two possible ways to roll the sum total of $n = 3$ out of all 36 possible outcomes.

$$\begin{aligned}(f * g)(3) &= f(1)g(2) + f(2)g(1) + f(3)g(0) \\ &= f(1)g(2) + f(2)g(1) + f(3) \cdot 0 \\ &= f(1)g(2) + f(2)g(1) \\ &= \frac{1}{6} \cdot \frac{1}{6} + \frac{1}{6} \cdot \frac{1}{6} = \frac{2}{36}\end{aligned} \tag{4}$$

Convolutional neural networks use 2-dimensional discrete convolution operator at each possible position in the image. This 2-D convolution is a dot-product between a grid-structured inputs and a grid-structured set of weights [1]. The second argument of the convolution is often referred to as the *filter* (kernel) and the output of the convolution is *feature map*. The equation for 2-D correlation with 2-D image I and 2-D filter F is defined as follows:

$$(I * F)(i, j) = \sum_{m} \sum_{n} I(m, n)F(i - m, j - n). \tag{5}$$

This 2-D convolution is a linear transformation that preserves the notion of ordering in the image. It is sparse (only few input units contribute to a given output unit) and reuses parameters (the same weights are applied to multiple locations in the input) [17]. An example of a 2-D convolution can be seen in Figure 4.1.

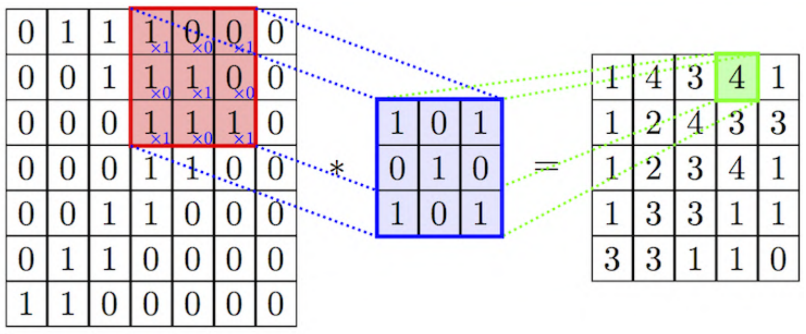


Figure 4.1: Example of 2-dimensional convolution on two grid-structured inputs. The rectangular array on the left could represent a pure black and white input image I. The smaller blue rectangular array in the middle is the filter F, which is being applied to region of I highlighted in red. Result of this step of the convolution operation is output in the green highlighted cell of feature map on the right. This process is repeated for every step in the x-axis and the y-axis, giving the final result [68].

4.1.1 Motivation for Using Convolution

Convolution is used in CNNs because it leverages three important ideas that can help improve a machine learning system: *sparse interaction*, *parameter sharing* and *equivariant representation* [25].

Traditional neural networks e.g. MLP (multi-layered perceptron) with one hidden layer, which is a fully connected architecture, would have to train around 170 000 weights (parameters) for an image of size 244×244 with 3 color channels. In contrast, convolutional neural networks use 2-D convolution with filter that is smaller than the input image. Repeat applications of this filter at certain strides in the input detect small, meaningful features in the local structure of the input image. Each neuron in the next layer following the convolution is only connected to a small number of neurons in the previous layer (given by the size of the filter). In other words, not all neurons in a particular layer are connected indiscriminately to neurons in the previous layer. This makes it possible to store fewer parameters, which both reduces memory requirements of the model, improves its statistical efficiency and requires fewer operations to compute the output. [25]

Parameter sharing is used in convolutional layers of the CNN to limit the number of free parameters in the model. This parameter sharing in CNNs means that the network does not learn a separate set of parameters for every location in the image. Convolutional neural networks assume that if it is useful to compute a certain feature at some position in the input image, then it is very likely that the same feature might be useful to compute at other locations in the image. The basic idea is that a rectangular patch of the image corresponds to a portion of the visual field and should be interpreted in the same way no matter where it is located [1]. Also, another big advantage of sharing parameters in the CNN is further reduction in the number of parameters involved in the network without hindering the networks expressive power.

This sharing of parameters in CNNs causes the convolutional layers of the network be equivariant to translation. Equivariance of a function to translation means that if the input to the function changes, the output changes in the same way [25]. For example, if the input image is shifted in some way $s(I)$, the convolution of the shifted image $h(s(I))$ would be equivalent to the convolution of the original image $h(I)$ after being shifting in the same way $s(h(I))$. Two functions $s(I)$ and $h(I)$ are equivariant if $s(h(I)) = h(s(I))$. That is to say, if an object is moved in the input, its corresponding representation in the feature map is moved accordingly as well.

4.1.2 Zero-Padding of the Input

Zero-padding of the inputs of convolutional layers is important in implementations of CNNs. Without padding, the 2-D input the width and height of the image representation would shrink after every convolutional layer. The image representation would shrink in size by one half of the filter size around the edges, as evidenced by the example of convolution operation in Figure 4.2 (a).

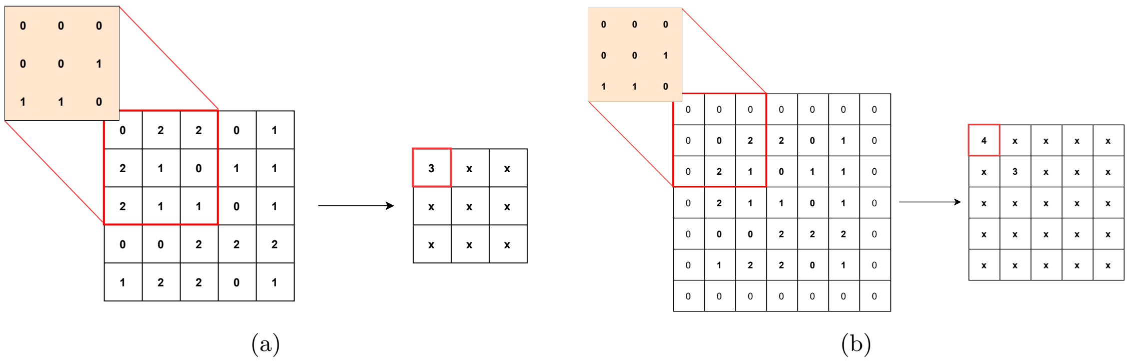


Figure 4.2: The figure shows two examples of feature maps resulting from a convolutional layer of the network. The example (a) is valid padding and the resulting feature map shrinks by $\left\lfloor \frac{f}{2} \right\rfloor$ of the size $f \times f$ of filter F. In the example (b) padding around the edges of the input I equal to the $\left\lfloor \frac{f}{2} \right\rfloor$ is used to prevent spatial reduction in the feature map.

This type of reduction in size is not desirable when using the convolutional layers of the network, because it tends to lose information around the edges of the image (or of the feature map in the case of hidden layers) [1]. Without padding the inputs of the convolutional layer, it would be either necessary to choose small filters or significantly shrink the neural network. Both of these options would limit the expressive capabilities of the network.

There are some extreme use cases of CNNs where there is no padding used in the convolutional layers, in this case the filter is only applied to the areas where the filter is fully inside the input. This is referred to as *valid padding* (filter is only applied to „valid“ positions). Valid padding tends to show worse experimental results and ultimately limits the maximum number of convolutional layers in a network.

Another type of padding used in CNNs the so called *same padding*, shown in Figure 4.2 (b). Its name is derived from keeping the size of the feature map same as the size of the input. The edges of the input are padded by $\left\lfloor \frac{f}{2} \right\rfloor$ for a filter F of size $f \times f$. With this type of padding the network can contain any number of convolutional layers without ultimately reducing the output to size of 1×1.

A different type of padding is referred to as *full padding*. In this case the input is padded by size $f - 1$, almost equal to the size of the filter F. Therefore, at the very edges the filter and the input would only overlap at a single single pixel. This is another extreme case, this type of padding is sometimes used because in the case of same padding (or valid padding) the features around the edges of the input can be underrepresented in the output feature map due to being result of fewer applications of the filter. However, this can mean that learning a single filter that performs well at all positions of the input will be more difficult.

Usually, the optimal amount of zero-padding of the inputs (in terms of expressive power of the network) lies somewhere between valid padding and same padding [25].

4.1.3 Strides

Strides represent a way in which convolution can reduce the spatial dimensions of the inputs. Stride in CNNs is the number of pixels (positions) by which the filter shifts during convolution. The convolution operation mentioned so far performed the dot-product between grid-structured input and filter at every possible point (one pixel at a time). This corresponds to convolution where stride of $s = 1$ is used for both x and y axes. Different stride values can be used for different axes if necessary.

Using higher stride values can reduce the granularity (downsample) of the convolution [1]. Higher stride values (e.g. $s = 2$ or $s = 4$) can be helpful in case the memory-constraints of the CNN pose an issue or the input image is of too high a resolution and could lead to overfitting.

The size of the feature map output by a convolutional layer with stride s, input I with size $w \times h$ (including padding if used) and filter F with size $f \times f$ is given as:

$$w' = \left\lfloor \frac{w - f + s}{s} \right\rfloor + 1 \quad h' = \left\lfloor \frac{h - f + s}{s} \right\rfloor + 1. \tag{6}$$

In practice, strides $s \geq 4$ are used rarely used, the resulting output feature map have smaller spatial dimensions and reduce the overlap of *receptive fields* (regions of input space where the filter is applied). It is possible to try different stride lengths at different convolutional layers of the network to check which gives the best performance on the validation data [45].

4.2 Transposed Convolution

Normal convolution operation used in CNNs typically reduces the spatial dimensions of the input. In convolutional neural networks it is sometimes desirable to upsample the size of the input (e.g. in generators of GANs). This can be achieved by using *transposed convolution*, also called *fractionally strided convolution* or incorrectly referred to as deconvolution [1].

Transposed convolution can be computed as regular convolution on inputs with additional rows and columns of zero values inserted around the actual values of the input [34]. The convolution then outputs a feature map of larger size than the input, effectively upsample the input.

In practice, a faster way of computing the tranposed convolution is used. Same outputs can be achieved by representing the filter of the transposed convolution as a *Toeplitz matrix* [58] and reshaping (unrolling) the input of the transposed convolution to a vector of values. The result of multiplying the Toeplitz matrix and the vector of input values will be another vector, which needs to be reshaped to matrix form. The final result of the transposed convolution is this reshaped matrix. [34]

[1]The term „deconvolution“ is sometimes used in machine learning literature to refer to the operation of transposed convolution. However, deconvolution is defined in mathematics as the inverse operation to convolution which is not the same thing as transposed convolution.

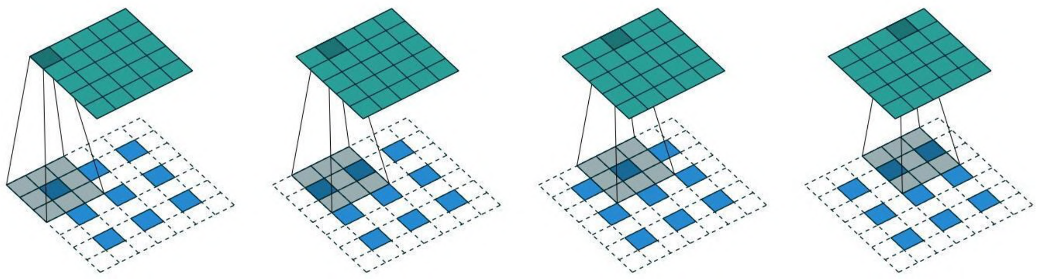


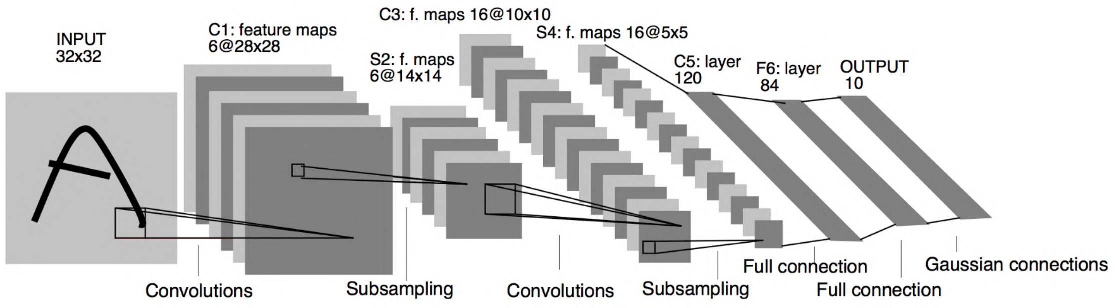
Figure 4.3: An example of a tranposed convolution operation on an input of size 3×3 that is padded by $p = 1$ number of zero values. Regular convolution is performed on the padded input with a filter of size 3×3 resulting in an output feature map of size 5×5, effectively upsampling the input. [66]

4.3 Architecture of Convolutional Neural Networks

A convolutional neural network is a sequence of layers that fulfill essential functions in the network, CNNs are comprised of several different types of layers depending on their function.

Typical CNN includes an input and an output layer (designed in an application specific way) together with a sequence of multiple hidden layers. These hidden layers of a CNN are usually comprised of a combination of convolutional layers, pooling layers and layers with a non-linear (piece-wise linear) activation function [34]. Combined, these different types of layers extract meaningful feature information from the input of the convolutional network. They are usually followed by a sequence of fully connected layers, which function exactly in the same way as a typical feed-forward neural network (e.g. MLP).

Unlike in traditional neural network architectures, neurons within layers of a convolutional network tend to be arranged in 3-dimensional space with limited local connections to the previous layer. Convolutional neural networks take advantage of the fact that an image of width w, height h and number of color channels c can be thought of as a volumetric input (tensor) of size $w \times h \times c$.



(a)

Figure 4.4: An example of a convolutional neural network architecture. This is the original architecture of LeNet-5 network from LeCun et al. (1995) [39], one of the earliest convolutional networks. The network contains a sequence of convolutional layers and subsampling layers (replaced by pooling in more modern networks) followed by a set of fully connected layers [39].

4.3.1 Convolutional layer

Convolutional layers are the most fundamental part of a convolutional network, they learn about the local features in their structured inputs. Convolutional layers are made up from a number of filters which are connected to a local region (receptive field) in the input of the layer. Inputs of a convolutional layer are convolved (detailed in Section 4.1) with this set of filters to produce an output feature map. Each filter in a convolutional layer is associated with its own bias which is added to the convolution and is learned throughout the training of the network.

Filters in convolutional layers are separately applied to the input volume in parallel. Outputs from filters of the same depth produce a 2-dimensional feature map, which are stacked together producing a volumetric output for the next layer of the network.

4.3.2 ReLU Layer

Convolutional layers in CNNs are usually directly followed by application of a nonlinear (or a piece-wise linear) activation function [1]. Although it is not explicitly displayed on the diagrams of network architecture of CNNs, almost every CNN uses a layer of activation functions following the convolutional layers. The application of these activation functions is not very different from their application in traditional neural networks, it does not change the spatial dimensions of the input as it is a one-to-one mapping.

In convolutional neural networks, the most commonly used activation function is currently the *rectified linear unit* (ReLU). It is a simple, fast to compute activation function which returns 0 if the input is negative or returns the value of input if it is positive.

$$f_{ReLU}(x) = max(0, x) \tag{7}$$

The reason for using a nonlinear activation function such as ReLU in the CNN is for the network to be able to nonlinear mappings[34]. Not all CNNs use the basic ReLU function, there are other variants of this activation function e.g. leaky ReLU, noisy ReLU, parametric ReLU etc.

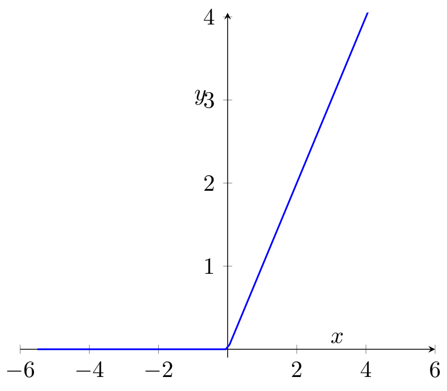


Figure 4.5: Plot of the rectified linear unit or ReLU function commonly used as activation function in CNN on the outputs of the convolutional layers of the network.

Leaky ReLU function is an improved variant of the basic ReLU function, which allows for a small positive gradient when the unit is not active. This function is used to prevent the “dying” ReLU problem, which is a form of the vanishing gradient problem, where the weights of the ReLU neuron could update in such a way that it never becomes active again. Such a neuron could become inactive for the remainder of the training.

$$f_{LeakyReLU}(x) = \begin{cases} \text{x} & \text{if }, x > 0, \\ 0.01\text{x} & \text{otherwise.} \end{cases} \tag{8}$$

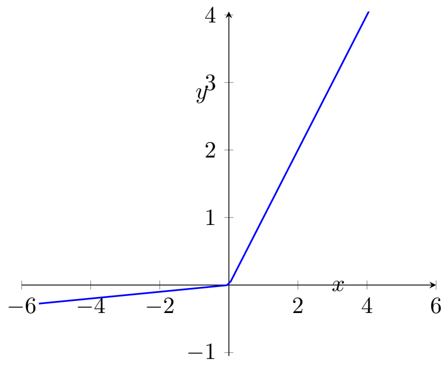


Figure 4.6: Plot of the leaky ReLU function, which has a very small predetermined slope for the negative x values in order to provide well defined gradients.

4.3.3 Pooling layers

Pooling layers operate on the feature map that is output from the previous layers of the CNN to simplify the information. They are typically interleaved with convolutional layers in the network.

The pooling layer applies a pooling function to the inputs of the layer. The pooling function replaces the outputs of the previous layer with a summary statistic of nearby outputs creating a condensed feature map [25]. Popular pooling functions are maximum (max-pooling) and average (average-pooling).

Pooling operates at each input feature map individually to produce another feature map with reduced spatial dimensions. Pooling does not change the number of feature maps, meaning that the output of the pooling layer keeps the same depth as the depth of the input. [1]

Similar to the convolutional layers, it is necessary to specify the size of a region being pooled and stride.

4.3.4 Fully Connected Layers

Fully connected layers of the CNN are typically placed after a sequence of convolutional, ReLU and Pooling layers towards the end of the convolutional network. However, there are network architectures which use fully connected layers at intermediate locations of the CNN and some that don't use fully connected layers at all.

Each neuron in these fully connected layers is connected to every neuron in the input. Fully connected layers in CNNs function in the exact same way as in traditional neural networks (e.g. layer in a MLP). The operation of neurons in this layer is a simple matrix multiplication followed by addition of the bias terms and application of a nonlinear activation function [34]. The fully connected are essentially convolutional layers that use filter size of 1×1.

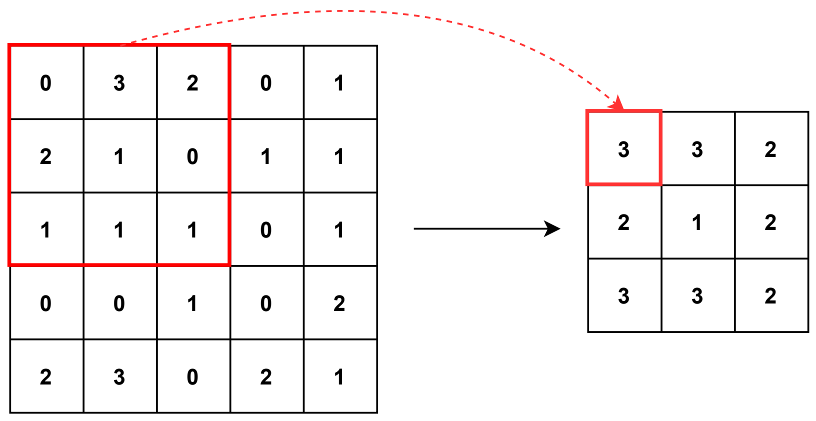


Figure 4.7: An example of max-pooling a feature map of size 5×5 with stride 1. Each feature map from the previous layer of the network is processed in this way.

The fully connected layers provide a way of learning a combination of the high-level features extracted from the final layers of the convolution, they aggregate information from the final feature maps. The spatial grids of the final feature maps are flattened and concatenated to create a high-dimensional representation of the input image which serves as an input to the fully connected layer.

Chapter 5

Synthetic Image Generation Using Generative Adversarial Networks

Generating realistic looking synthetic images has been a long-standing problem in machine learning. There have been a number of innovations made in recent years in the field of deep generative modeling, especially pertaining to generative adversarial networks.

Generative adversarial networks are an emerging machine learning technique for generative modeling which can be used for synthetic image generation. The original design of generative adversarial networks was proposed by Goodfellow et al. (2014) [26]. Their idea was to use a set of two neural networks to compete against one another and iteratively improve each others performance, eventually being able to reproduce training data distribution (e.g. produce images).

This chapter covers the architectural details of generative adversarial networks, their training and variations on the original generative adversarial model which can be used for synthetic image generation.

5.1 Generative Adversarial Networks

Generative adversarial networks (GAN) are a type of deep generative models based on differentiable generator networks introduced by Goodfellow et al. (2014) [26]. The inspiration behind GAN is based on game theory. The idea is to set up a game between two players. The framework of GAN consists of two simultaneously trained *convolutional neural networks* models, a *generator* and a *discriminator*, that are in direct competition with one another.

The generator of a GAN creates new samples of fake data (e.g. synthetic fingerprint image) that are supposed to look as though they were drawn from the same probability distribution as the training dataset (real data). The generator draws noise data from a random probability distribution which it eventually transforms into real looking data samples. Notably, the generator has no direct access to actual data samples, it learns solely through interaction with the discriminator.

The discriminator then receives samples from both the real training data and fake data created by the generator and tries to identify whether the data sample comes from the training dataset or the generator. The discriminator learns using traditional supervised learning methods for classification to classify its inputs into one of the two classes (real or fake) [24].

In this adversarial modeling framework, the goal of the generator is to generate data in such a way that it increases the misclassification rate of the discriminator. The discriminator, on the other hand, tries to minimize its own misclassification rate on the input data it receives. The aim of the generative adversarial network as a whole is to train both models to the point where the discriminator cannot distinguish between real and generated images.

It is, therefore, possible to use GANs to artificially generate entire datasets of synthetic data that mirror the distribution of a real dataset. During the course of training the network, the generator learns a mapping from a randomized vector to a realistic looking fake data sample. Subsequently, the generator of the GAN can be used after the training is finished to create any number of realistic looking samples.

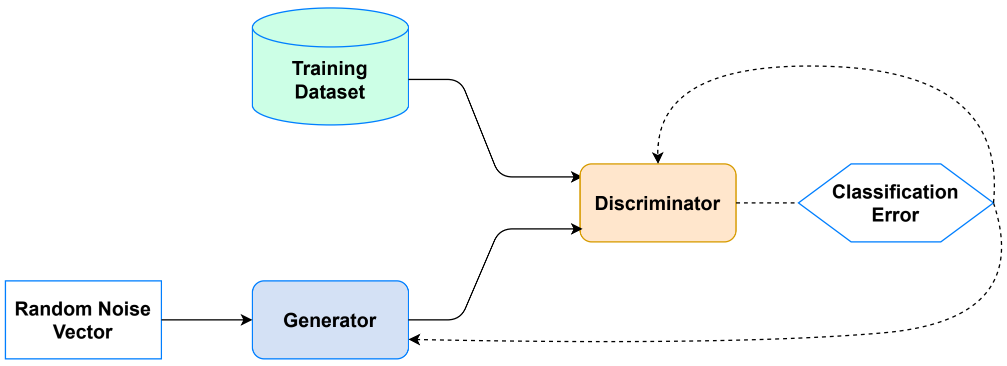


Figure 5.1: Basic architecture of a generative adversarial network (GAN) consisting of two convolutional neural networks: generator and discriminator. The generator takes a randomized vector as an input and tries to create a fake data sample that captures the characteristics of the training data set. The discriminator takes a data sample as an input and decides whether it is drawn from the training data set or created by the generator. Both the generator and the discriminator are then updated according to the classification error of the discriminator.

5.1.1 Generator

The generator of a GAN is a machine learning model that represents a differentiable function $G : R^{|z|} \rightarrow R^{|x|}$, which maps vectors of random values (noise) $z \in R^{|z|}$ from a latent space to an output $x = G(z), x \in R^{|x|}$ that looks like a realistic looking data sample, where $|x|$ is the size of the created data sample. [24][10]

Generators are typically implemented as a CNN that learns this mapping function G during the training process of the GAN. The idea is to try to mirror patterns or probability distributions of the training dataset consisting of real data.

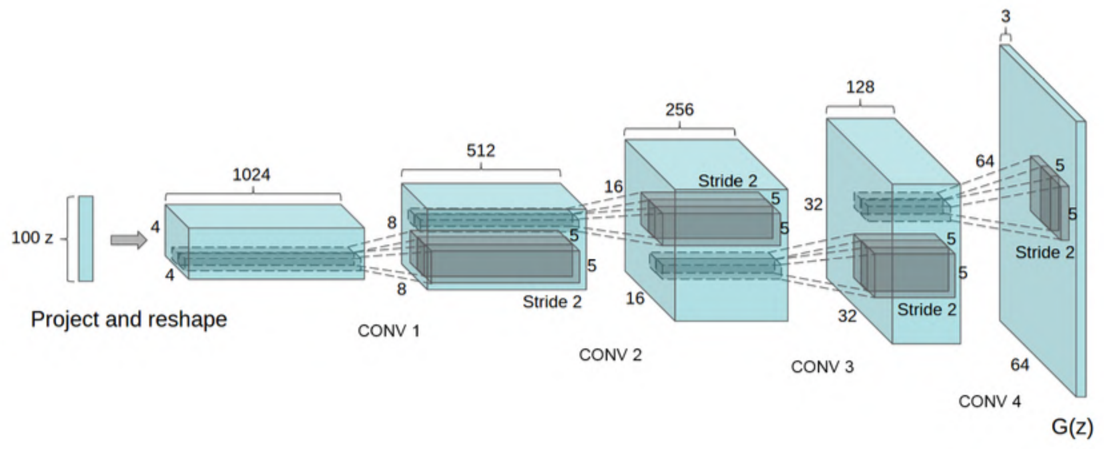
The first step of a generator is to sample a random vector z from some prior probability distribution (e.g. normal distribution) which will be used as an input to the rest of the convolutional network to create a fake data sample $x = G(z)$. Conceptually, the random vector z of size n is drawn from an arbitrarily defined n-dimensional latent space. The vector z then represents latent features of the data sample created by $G(z)$. No intrinsic meaning is assigned to these latent features explicitly, the generator learns to apply meaning to points in the latent space throughout the training process.

The rest of the generator network then takes the random vector z as an input and passes it through a series of transposed convolutions. These layers upsample the vector z

that serves as the seed for the generation of data sample x. The convolutional layers of the generator are typically followed by application of a nonlinear activation function. Multiple different activation functions can be used throughout the networks layers (e.g. DCGAN uses ReLU for the inner convolutional layers and tanh for the last layer) [53].

It is worth noting that not all of the input values to function $G(z)$ must necessarily correspond to the inputs of the first layer of the convolutional network. It is possible to partition the vector z into multiple vectors, e.g. z_1 and z_2, and then feed z_1 as the input to the first layer of the CNN and use z_2 as an input at another point of the CNN.

In general, there are very few restrictions placed on the design of the generator network. The function G must be differentiable and the dimension of z should be at least as large as the dimensions of x. [25]



(a)

Figure 5.2: Generator of a deep convolutional generative adversarial network (DCGAN). The vector z of size 100 is first drawn from a uniform distribution and then projected and reshaped into an input of 1024 feature maps of size 4×4. These feature maps are subsequently put through a series of transposed convolutions (also known as fractional convolutons or deconvolutions) that eventually output a data sample $x = G(z)$ of size 64×64. [53]

5.1.2 Discriminator

The discriminator of a GAN is a binary classification network that takes two types of data samples as an input, real data from the training set and fake data created by the generator. The discriminator then uses traditional supervised learning techniques to separate the inputs into one of two classes – real and fake. In generative adversarial networks, discriminator can be characterized by a function $D : R^{|x|} \to [0, 1]$ that maps an input data samples $x \in R^{|x|}$ to the probability that the data sample is from a real dataset. [10]

The goal of a GAN is to eventually train the generator of the network to the point that it is able to maximally fool the discriminator. A point at which the misclassification rate is approximately 50%. The successfulness of the classification by the discriminator is used to provide feedback to both the discriminator and the generator during the training of the network.

The discriminator of a GAN is typically represented by a CNN. The general structure of the discriminator network includes a series of strided convolutional layers that reduce the spatial dimensions of the input. Again, the convolutional layers are usually followed by the application of an activation function (e.g. ReLU layer).

The use of fully connected layers differs from network to network. Some networks, such as SRGAN [40] from Figure 5.3, utilize a series of fully connected layers at the end of the network to transform the final feature map output by the last convolutional layer into probability of input belonging to one of the two classes.

Other networks, such as DCGAN [53], advocate the elimination of most of the fully connected layers in an attempt to increase the convergence speed of the network. The DCGAN architecture uses a single sigmoid layer as the output of the network. This sigmoid layer directly takes the feature map of the last convolutional layer as the input.

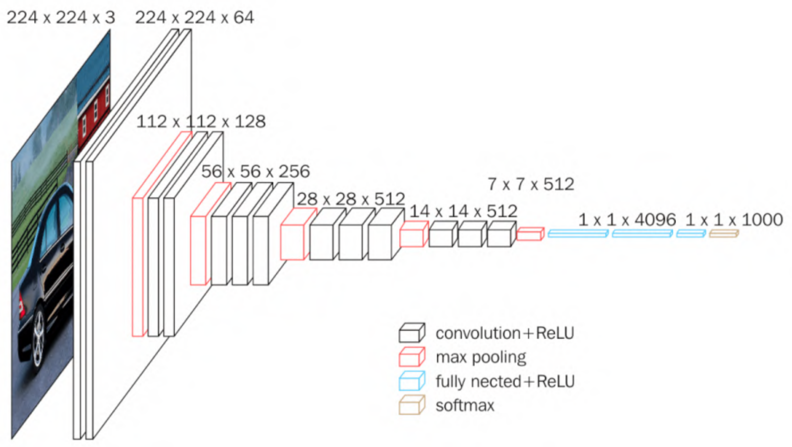


Figure 5.3: The convolutional network model VGG16 [56] used for classification of images. This CNN is the basis for the discriminator of the generative adversarial network used for image super-resolution (SRGAN) [40]. The network consists of a series of convolutional layers with small filter size of 3×3 followed by ReLU layers. The network also contains 5 max-pooling layers with filter size 2×2 and stride 2. The last layers of the convolutional network are a series of three fully connected and ReLu layers followed by the application of the softmax activation function. Softmax being the generalization of the logistic function to multiple classes. [69]

5.2 Training of Generative Adversarial Networks

The process of training a GAN involves finding weights (parameters) of the discriminator network D that maximize its classification accuracy whilst at the same time finding weights of the generator network G that minimize this classification accuracy of the discriminator. This training consists of alternately updating weights of the discriminator and generator models using simultaneous *stochaistic gradient descent* (SGD) [26]. While weights of one model are being update, the weights of the other model remain fixed.

The simplest way of describing the learning in GAN is as a *zero-sum game*. A zero-sum game is a game in which gains (losses) of one player D are directly proportional to the losses (gains) of the other player G. The payoff of the discriminator in this game can be specified by a utility function $U(G, D) = -J^{(D)}$, where $J^{(D)}$ is the discriminators cost function (cost is minimized in discriminator when training). As it is a zero-sum game, the payoff for the generator is defined as $J^{(G)} = -J^{(D)}$. Notably, the generator uses the same cost function as the discriminator in this case. These payoffs are parametrized by the weights (parameters) of the neural networks.

The solution for this zero-sum game problem (convergence of the model) is a Nash equilibrium g^*. A Nash equilibrium for this zero-sum game is a saddle point of function $v(\theta^{(G)}, \theta^{(D)})$. This saddle point is a local minimum with respect to the generators parameters $\theta^{(G)}$ and a local maximum with respect to the discriminators parameters $\theta^{(D)}$. [24][25]

$$g^* = arg \min_G \max_D v(\theta^{(G)}, \theta^{(D)}) \tag{9}$$

This approach was initially used in Goodfellow et al. (2014) [26] and they were able to show that learning in this sort of a zero-sum game is similar to minimizing the Jensen-Shannon divergence between the training data and the generators distribution. Jensen-Shannon divergence being a measure of similarity between two probability distributions (real data and generated data in this case).

5.2.1 Discriminator's Cost Function

The default choice for the cost function of the discriminator is usually the *binary cross entropy error function* [25]. The binary cross entropy function is defined as the negative logarithm of the likelihood function. It is the standard cost function used in models with sigmoid output Figure 5.4.

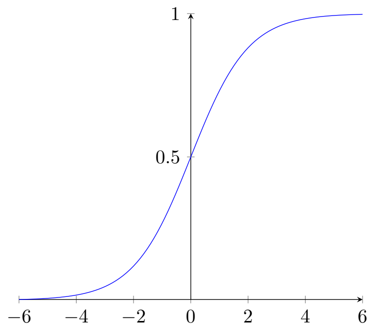


Figure 5.4: Activation function of logistic regression - logistic sigmoid.

$$J^{(D)} = -\sum_{i=1}^{n} t_i log(D(x_i)) + (1 - t_i) log(1 - D(x_i)) \tag{10}$$

Where $D(x_n) \in [0, 1]$ is the outcome of the classification by the discriminator and t_n is the correct class label for data sample x_n (1 for real and 0 for fake). The discriminator tries to minimize this cost function during the training of the model leading it to try to classify input data samples correctly. It is easy to verify that this sum is minimal when the generator correctly outputs 1 for real data and 0 for generated data.

However, the goal of training a GAN model is to teach the generator to create fake data samples which are indistinguishable from real ones. At the convergence of the model the discriminator should therefore output $\frac{1}{2}$ for all the samples, being unable to tell whether a data sample was synthetically generated or not.

5.2.2 Generator's Cost Function

Formulating the training of a GAN in the form of this zero-sum game was shown to be useful for theoretical analysis of the model, but it does not tend to perform particularly well in practice. The results shown in Goodfellow et al. (2014) [26] depend on convexity and the parameter space in the case of convolutional neural networks is not necessarily convex. [24]

Another problem with using the same cost function from Equation 10 for the generator as for the discriminator is that this can lead to the *gradient problem* (gradient saturation) [26]. In the zero-sum game mentioned previously, the discriminator minimizes this cost function and the generator tries to maximize the same cross-entropy function. Unfortunately, the output of this cost function saturates in extreme cases when the discriminator is close to being optimal, meaning the discriminator is able to correctly predict classes with utter certainty. Very high confidence of the discriminator in classifying its inputs leads to vanishing of generator's gradient [25]. In this case the generator is unable to learn in a meaningful way due to the values of the gradient being too small and the training of the model is stunted.

It is possible to use a different *non-saturating cost function* to solve this problem. This proposed cost function is a subtle variation on the minimax cost function. The discriminator tries to increase the probability of misclassification in the discriminator rather than decreasing the probability of correct classification [25].

$$J^{(G)} = -\sum_{i=1}^{n}(1-t_i)logD(x_i) \tag{11}$$

Changing this cost function no longer makes this a zero-sum game as it changes the payoffs of the generator. This slight change in the cost function of the generator allows for large gradients even when the discriminator confidently rejects generated samples. This sort of a cost function provides much stronger gradients earlier in the training process when the generator was not yet able to figure out a way to meaningfully replicate the training data distribution, see Figure 5.5. [25]

5.2.3 Simultaneous Stochaistic Gradient Descent

Stochaistic gradient descent (SGD) is used to learn the parameters for the discriminator and the generator. Stochaistic gradient descent is a gradient-based optimization technique that forms the expectation of a function's gradient based on minibatches of data. Gradient-based optimization methods are currently the most common approach to training neural networks. It is possible to use any gradient-based optimization method for training GAN, e.g. *Adam* optimizer [35]. [1][24]

The gradient update steps in a GAN alternately update the generator G and the discriminator D. It is possible to use $1 \leq k \leq 5$ steps for optimizing the discriminator for every 1 step of the generator, as was originally proposed in [26]. Currently the approach

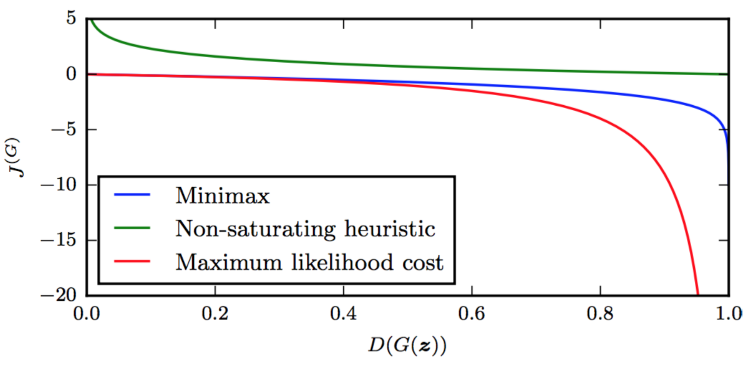


Figure 5.5: Graph of three variations on the cost function of the generator network proposed in Goodfellow et al.(2014) [26]. Notice the difference in the magnitude of non-saturating cost function's gradient when $D(G(z))$ is close to zero, and the discriminator can confidently classifies the sample as fake. [24]

that is thought to work best in practice is to use $k = 1$ for alternate steps of both D and G [54]. The weights update in the estimated gradient step is defined as:

$$\theta = \theta - \frac{\eta}{n} \sum_{i=1}^{n} \nabla_\theta J(x_i, t_i; \theta) \tag{12}$$

where θ are the parameters of the model (discriminator or generator), η is the learning rate, n is the size of the minibatch, $J(x_i, t_i; \theta)$ is the cost function of the model, x_i is the input from a minibatch and t_i the correct class of the input. The alternating steps of the algorithm update $\theta^{(D)}$ to reduce $J^{(D)}$ and $\theta^{(G)}$ to reduce $J^{(G)}$.

5.2.4 Batch Normalization

Since the introduction of DCGAN [53], most architectures of generative adversarial networks have involved *batch normalization* in some form. Batch normalization is a technique proposed by Ioffe and Szegedy (2015) [29] for reducing the internal covariate shift of neural networks. They defined internal covariate shift as the change in the distribution of network activations due to change in network parameters during training [29].

The reason for using batch normalization in deep learning is to improve training of models that use saturating nonlinearities (e.g. sigmoid). The idea is that since a deep neural network consists of numerous layers, small change in some of the initial layers of the network can manifest a large change in the later layers of the network. The issue with this is that such a small change in the parameters of the network can move many dimensions of the layers input into the saturated region of the nonlinear activation function. This significantly slows down the learning process and leads to the vanishing gradient problem. [29]

Batch normalization is defined as follows. Let $z = g(x)$ be an output of a layer in the neural network, where $g(.)$ is a nonlinearity and $x = Wu + b$ is an activation. Here, W and b are the weights and bias of the layer in the network and u is the input to the given layer. Let H be the matrix of activations $x = Wu + b$ of a layer for a minibatch of data. The matrix H is replaced by matrix H' in batch normaliztion.[25][29]

$$H' = \frac{H - \mu}{\sigma} \tag{13}$$

Where μ is a vector containing the mean of each unit and σ is a vector containing the standard deviation of each unit [25].

$$\mu = \frac{1}{n} \sum_{i=0}^{n} H_i \tag{14}$$

$$\sigma = \sqrt{\delta + \frac{1}{n} \sum_{i=0}^{n} (H_i - \mu)^2} \tag{15}$$

Normalizing the activations may reduce the expressive power of the network. For example, the influence of a learned bias parameter b for a layer will be canceled out due to the subtraction of mean. To prevent this, a set of two parameters γ and β are learned for each batch normalization in the network and the normalized H' is scaled and shifted by $\gamma H' + \beta$. [29]

For convolutional layers, the batch normalization should obey the convolutional property. The activations are normalized jointly for the minibatch over all locations (receptive fields) in the input feature map. [29]

However, there is a potential issue when training GAN with smaller minibatch sizes. Smaller minibatch sizes may be used due to limited GPU memory and this may cause increased fluctuation in the above mentioned normalization constants. Subsequently, the model might be influenced more by the change in the normalization constants than by the input itself. This issue was addressed by *virtual batch normalization* for GAN in Salimans et al. (2016) [54]. The virtual batch normalization uses an additional reference batch at the start of the training which remains constant. The activations are normalized by the minibatch and the reference batch. A downside of this approach is that it is computationally expensive, as it is needed to calculate the batch normalization twice. [24][54]

5.2.5 Spectral Normalization

Spectral normalization is a weight normalization technique first introduced in spectrally normalized GANs (SN-GANs) [44]. The aim of spectral normalization is to stabilize the training of GANs by imposing Lipschitz constraints on the function f of the network. Spectral normalization uses the spectral norm $\sigma(W)$ of a weight matrix W, which is equal to the largest singular value of W, to normalize the weights of the matrix W.

$$W' = \frac{W}{\sigma(W)} \tag{16}$$

Spectral norm $\sigma(W)$ of a matrix W normalized in this manner satisfies the Lipschitz constraint of $\sigma(W) = 1$. When each of the weight matrices W^l in the network satisfies this Lipschitz constraint, the function f of the network will be 1-Lipschitz continuous [44].

In practice, calculating the exact spectral norm $\sigma(W)$ used for normalization by using singular value decomposition at every step of the algorithm is very computationally expensive. Instead of calculating the exact value by singular value decomposition, a fast approximation method utilizing the *power iteration method* [23] is used in its place. The power iteration method has proven to be computationally cheap and effective. It relies on the fact that throughout the training of the network, the change in matrix W after an update step should be relatively small. Therefore, the change in the largest singular value of W should also be relatively small. The fast approximation method for spectral normalization relies on this fact and is able to reuse previous calculations in its estimation. [44]

5.2.6 Mode Collapse

Mode collapse is a common problem that plagues the training of GAN models. Mode collapse is a training failure in which the generator network fails to generalize properly and some of the modes (classes) are not well represented in the generated samples. Therefore, some of the samples that have support in the real data distribution will not be present in the samples taken from the generator. This problem consists of the generator learning to map a large subset of vectors z from the latent feature space to a small set of outputs $G(z)$ (partial mode collapse) or in the worst case to a singular output $G(z)$ (complete mode collapse). These outputs or output are subsequently assigned a high probability of being real by the discriminator which leaves the generator with little incentive to try and generate sufficiently different samples. [24][38]

The causes of mode collapse are not yet fully understood. The convergence of GANs proven in Goodfellow et al. (2014) [26] relies on the assumption that the optimization space is convex. However, this convergence does not necessarily hold when the loss function space is highly non-convex, as is the case in practical applications of deep convolutional networks. It has been theorized in [18][36] that mode collapse is linked to convergence to sharp local minima of the loss function. Whether a local minumum is considered sharp is dependent on the loss landscape around the point of convergence. Convergence to a sharp local minuma has been linked to worse over-all performance of GANs (lower inception score) [18]. Gradients around these sharp local minima encourage the generator to map multiple vectors z to a single output or a small number of outputs $G(z)$, leading to mode collapse.

One way of avoiding the mode collapse problem is having the ability to train the discriminator to almost optimality without needing to worry about vanishing gradients and consequently convergence failure. In this case, the discriminator will be able to eventually reject the relatively small set of outputs that the generator stabilizes on. This approach is taken by the Wasserstein GANs [2] (see Subsection 5.3.3), where in their experiments they encountered no evidence of mode collapse.

Another proposed approach to mitigate the issue of mode collapse is to regularize the discriminator in order to constrain its gradient [36]. Penalizing the gradient was shown to result in more stable training in DRAGANs [36] and is likely one of the reasons for the success of Wasserstein GANs with gradient penalty (WGAN-GP).

5.3 Types of Generative Adversarial Networks

Generative adversarial network are still a rather novel approach to deep generative modeling. They are currently an intense field of research. There exists numerous extensions and adjustment to the GAN architecture, as well as to the methods they employ. This section mentions some of these variations of GANs which can be used for synthetic image synthesis.

5.3.1 Deep Convolutional Generative Adversarial Networks

Deep convolutional generative adversarial networks (DCGAN) are an expansion on the original GAN concept proposed by Radford et al. (2016) [53]. Their goal was to improve the stability of GAN training for higher resolution images in most settings. They were the first GAN model to generate high resolution images in a single shot (not using multi-stage generation process) [24].

Their approach was motivated by the *all-convolutional networks* [57]. They proposed a set of constraints on the architecture of the GAN [53]:

1. Replace all pooling functions (e.g. maxpooling) in the generator and the discriminator networks with strided and fractionally strided convolutions instead, when it is necessary to increase or decrease spatial dimensions of the input.
2. Remove all fully connected hidden layers in the networks. The discriminator still has one fully connected sigmoid layer at the end of the convolutional network connected to the last convolutional layer.
3. Use batch normalization in almost all layers of the networks to stabilize the learning. Last layer of the generator and first layer of the discriminator are not batch normalized. This is supposed to improve training in cases of poor initialization of the networks and prevent mode collapse.
4. Use LeakyReLU in the discriminator and ReLU in the generator. With the exception of the last layer in the generator, which uses tanh instead.

5.3.2 Big Generative Adversarial Networks

Big generative adversarial networks (BigGAN) were introduced in Brock et al. (2018) [6]. The name BigGAN stems from the fact that they attempted to train GANs at a largeer scale than ever before, this is both in terms of the model's parameters and batch sizes used for training purposes. BigGAN do not propose any significant modifications to the GAN framework, they instead leverage considerable increase in computational power to dramatically improve performance of GAN. [6]

The architecture of BigGAN is based on ResNet GAN architectures [44][62]. The model employs batch normalization, spectral normalization [44] and class conditional information [16]. *Skip-z connections* were used in the generator to split the latent input vector z and connect parts of z to deeper layers of the generator network. [6]

BigGANs also propose a change to the probability distribution of latent vectors z used in the generator. Typically, z is either sampled from a normal distribution or a uniform distribution. However, BigGAN experimented with a so called *truncation trick.* The truncation trick constitutes using a normal distribution to generate z, but resampling z if the magnitude of z is above a chosen threshold. This has lead to improvements in the individual generated data sample quality at the expense of generated data sample variety. [6]

Despite dramatic improvements in generated image quality, BigGANs still faced challenges of instability in training their networks which were solved using *early stopping.* Early stopping is a workaround that allows the training to collapse, but takes a snapshot of the networks weights before the collapse happens. These weights are usually enough to produce data samples of sufficient variety. [6]

5.3.3 Wasserstein Generative Adversarial Networks

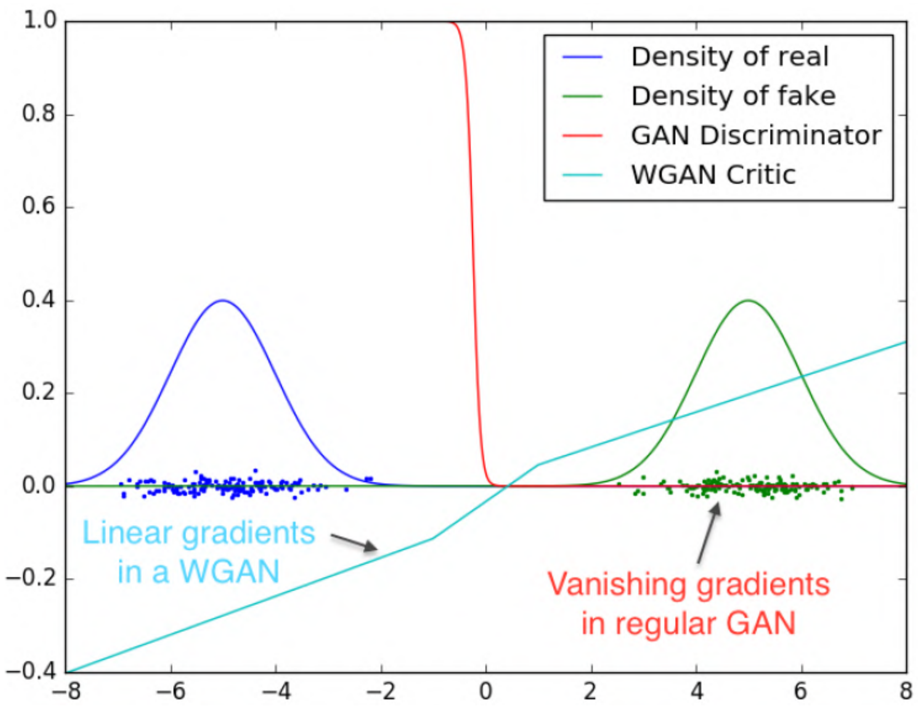
Wasserstein generative adversarial networks (WGAN) are a variation of GAN which use a smoother metric for measuring distance between probability distributions. This new metric is called *Wasserstein distance*, also known as Earth-Mover distance. Informally, Earth-Mover distance can be thought of as the minimum energy cost of moving and transforming a pile of dirt in the shape of one probability distribution into the shape of another probability distribution. [2][61]

In WGAN they refer to the discriminator model as the "critic". This is due to it being used to estimate how real or fake a given data sample looks, instead of classification. The critic acts as an aide for estimating the Wasserstein distance between real and generated data. A function f_w is learned by the critic which is used to compute the Wasserstein distance:

$$W^{(D)} = \frac{1}{m}\sum_{i=1}^{m} f_w(x_i) - \frac{1}{m}\sum_{1}^{m} f_w(g(z_i)) \tag{17}$$

which is the cost function for the discriminator (critic). Where x_i is a real data sample, $g(z_i)$ is a generated data sample and $2m$ is the minibatch size. Similarly to the non-saturating heuristic game, the cost function of the generator is:

$$W^{(G)} = -\frac{1}{m}\sum_{i=1}^{m} f_w(g(z_i)). \tag{18}$$



(a)

Figure 5.6: An example of the difference in the gradients of an optimal discriminator (GAN) and an optimal critic (WGAN). The two disjoint Gaussian distributions represent a real data distribution and a generated data distribution. [2]

One issue in this proposed model is that the f_w must be a K-Lipschitz continuous [61] function. *Weight clipping* is used in the model of WGAN's critic to enforce the Lipschitz constraint on the function f_w. After every gradient update in the critic, the weights are clamped to range $[-c, c]$, where c is a chosen clipping parameter. The authors of WGAN were dissatisfied with the use of weight clipping, an alternative solution to the Lipschitz continuity of f_w in the form of gradient penalty has been proposed instead [27]. [2][61]

Chapter 6

Design of GAN for Generating Skin Diseased Fingerprints

Despite the recent rise to prominence of generative adversarial networks as the state-of-the-art deep generative models used for image synthesis, there are very few applications of GAN for generating fingerprint images. These attempts were met with a varying degree of success [5][7][20][43], but they have demonstrated the possibility of using GAN to generate fingerprint data for biometric purposes.

The aim of this thesis is to try to apply these methods for generating fingerprints using GAN to create realistic looking diseased fingerprint images. The design of a GAN, which will be trained on data samples from the diseased fingerprint database mentioned in Section 3.3, is detailed in this chapter. The chapter covers the process of data augmentation used for the training dataset, the architectural structure of the GAN and the proposed training scheme.

6.1 Data Augmentation

Data augmentation is a method for reducing overfitting of deep learning models by increasing the amount of available training data. It is a popular technique used in medical image analysis, where acquiring more data is a difficult task [19].

Data augmentation artificially inflates the training data set by applying modifications to the existing training data. Image data augmentation is potentially the most well-known type of data augmentation, simpler forms rely on geometric transformations of the training image samples. The idea is to create different appearances of the same image while preserving it's meaning. [55]

I propose to use downscaling and image cropping the central parts of the fingerprint image to reduce the size of the inputs to 256×256. This will not only increase the size of the dataset, but will also make the training of the GAN converge faster, as generating smaller realistic looking images is an easier undertaking.

Additionally, rotation augmentation and horizontal flipping will be applied to the sample image to further increase dataset size. Horizontal flipping is one of the easiest to implement, as it consists of inverting the x axis of the sample image. As for the rotation augmentation, it rotates the image a random amount around an axis. I suggest using a random rotation between $1°$ and $10°$. These transformation can be applied to the training images simultaneously. The combination of these methods will hopefully prove to be sufficient for the training of the GAN as well as help reduce overfitting of the network.

6.2 Generative Adversarial Network Architecture

The generative adversarial network model is based on a WGAN utilizing the gradient penalty (WGAN-GP), similar to the one used in Bontrager et al.(2018) [5], where they were able to generate fingerprint images of size 128 $\times$ 128 using a NIST dataset.

The discriminator (critic) network consists of five downsampling convolutional layers with kernel size 5×5, stride 2 and padding 2, which are followed by spectral normalization, dropout with the rate of 0.4 and LeakyReLU activation function, see Figure 6.1. The last downsampling convolutional layer of the network uses the same kernel size, stride and padding as the previous layers, but differs from them by excluding dropout.

The network uses spectral normalization instead of batch normalization. The reason for this is that batch normalization takes multiple inputs in a batch and introduces correlation between them. This doesn't work well with the gradient penalty which is applied with respect to each individual input. However, the WGAN-GP method works with spectral normalization as it doesn't introduce correlation between training samples. [27] In fact, it introduces another method of enforcing Lipschitz continuity of the functions learned by the discriminator.

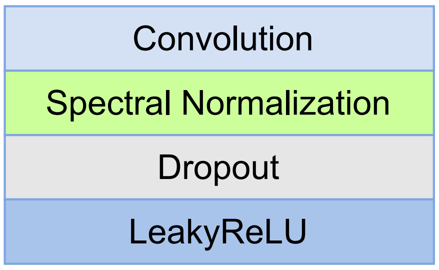


Figure 6.1: A single downsampling convolutional block of the discriminator (critic) network consisting of one convolutional layer, spectral normalization and dropout with a LeakyReLU activation function.

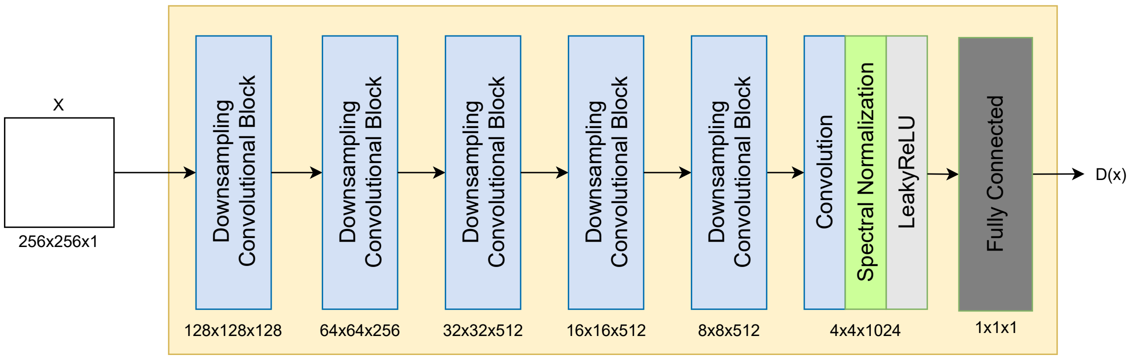

X
256x256x1
Downsampling Convolutional Block
128x128x128
Downsampling Convolutional Block
64x64x256
Downsampling Convolutional Block
32x32x512
Downsampling Convolutional Block
16x16x512
Downsampling Convolutional Block
8x8x512
Convolution
Spectral Normalization
LeakyReLU
4x4x1024
Fully Connected
1x1x1
D(x)

Figure 6.2: Architecture of the discriminator (critic) network in the proposed WGAN-GP.

Dropout is used in the network as a form of regularization which will force the network to learn more robust features. Even though the use of dropout will inevitably increase the number of iterations before the convergence of the GAN, the computational time required necessary for each individual iteration will be lowered as a result.

The feature map output by the last convolutional layer of the discriminator network is flattened and used as an input to a fully connected layer which is not followed by an activation function. The input x of the discriminator network is a 256×256 sized real or generated image. The output of the discriminator $D(x)$ is a single scalar used to estimate the Wasserstein metric.

The generator network takes as an input z a vector of 256 randomly sampled values from a normal distribution $\mathcal{N}(0, 1)$ and transforms it into the output $G(z)$, a 256×256 sized greyscale image. This vector of latent variables is first reshaped by the fully connected layer of the generator, which uses batch normalization and is followed by the application of ReLu activation function.

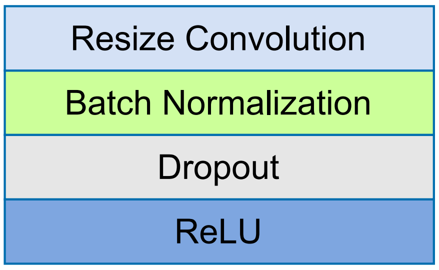


Figure 6.3: A single upsampling convolutional block of the generator network consisting of one convolutional layer, spectral normalization and dropout with a LeakyReLU activation function.

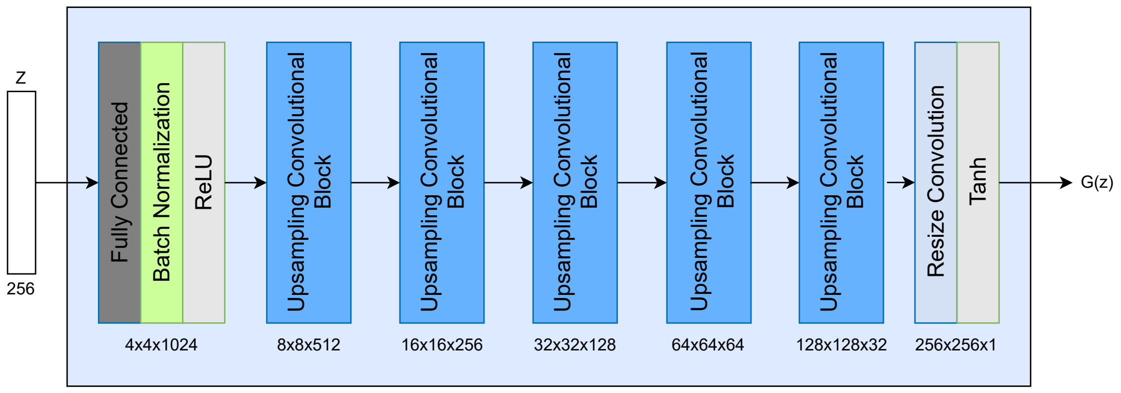

z
256
Fully Connected
Batch Normalization
ReLU
4x4x1024
Upsampling Convolutional Block
8x8x512
Upsampling Convolutional Block
16x16x256
Upsampling Convolutional Block
32x32x128
Upsampling Convolutional Block
64x64x64
Upsampling Convolutional Block
128x128x32
Resize Convolution
Tanh
256x256x1
G(z)

Figure 6.4: Architecture of the generator network in the proposed WGAN-GP.

The result of this first reshape layer is then used as an input to the sequence of five upsampling convolutional blocks, see Figure 6.4. Each upsampling convolutional block consists of *resize convolution*[46] layer followed by batch normalization, dropout with probability of 0.4 and application of the ReLU activation function. The resize convolution is an alternative to transposed convolution that helps to reduce checkerboard artifacting in the generated images. Resize convolution layers consists of nearest neighbor upsampling followed by regular convolution with kernel size of 5×5, stride 1 and padding 2.

Each fully connected and upsampling convolutional layer of the generator network is followed by batch normalization and ReLU activation function, with the exception of the last layer. Directly applying batch normalization can lead to model instability and sample oscillation as was shown in DCGAN [53]. This can be avoided by excluding batch normalization from the output layer. The last layer, therefore, does not include batch normalization and uses hyperbolic tangent (Tanh) as an activation function instead.

6.3 Proposed Training Scheme

Algorithm 1 details the training of the proposed WGAN-GP. In this training scheme, the discriminator D network uses the modified Wasserstein loss function with gradient penalty:

$$\begin{gathered} W^{(D)} = -D(x) + D(\tilde{x}) + \lambda(\|\nabla_{\hat{x}} D(\hat{x})\|_2 - 1)^2 \\ \hat{x} = \epsilon x + (1-\epsilon)\tilde{x} \quad \epsilon \sim U[0,1] \end{gathered} \tag{19}$$

where x is a real data sample, $\tilde{x}$ is a generated data sample, λ is a gradient penalty coefficient and $\hat{x}$ is a point drawn uniformly from lines between x and $\tilde{x}$. This gradient penalty is used instead of weight clipping as a soft constraint on the Lipschitz continuity of the functions learned by the generator, which is a requirement for training WGAN. [27]

The non-saturating cost function mentioned in Subsection 5.2.2 is used for updating the weights of the generator network:

$$J^{(G)} = -D(G(z)). \tag{20}$$

Before the start of the GAN training, it is necessary to initialize the weights of the discriminator $\theta^{(D)}$ and the weights of the generator $\theta^{(D)}$. These weights are randomly initialized using a normal distribution $\mathcal{N}(0, 0.02)$.

The total number of training epochs will be decided by monitoring and assessing the intermediate results produced by the GAN. The number of update steps of the discriminator for each generator update is set to $k = 3$ as a compromise between recommendations from WGAN-GP [27] and Goodfellow (2016)[24]. The minibatch size is set to $m = 64$ data samples, this means that for the training of discriminator a total of 128 data samples are used.

The penalty coefficient is set to $\lambda = 10$, in accordance with the recommendations put forward by WGAN-GP [27]. They found this value worked well across a variety of architectures and datasets.

The hyperparameter values used for the Adam optimizer were the ones recommended by WGAN-GP [27], namely learning rate $\alpha = 0.0001$, exponential decay rate for the first moment estimates $\beta_1 = 0$ and exponential decay rate for the second moment estimates $\beta_2 = 0.9$.

Algorithm 1: Training WGAN-GP

```
   Require: $k, m, \lambda, \alpha, \beta_1, \beta_2$
1  Initialize $\theta^{(D)}$ and $\theta^{(G)}$
2  for number of training epochs do
3      for k steps do
4          for i = 1,...,m do
5              Sample a noise vector $z$ from $\mathcal{N}(0,1)$
6              Sample a random number $\epsilon$ from $U[0,1]$
7              Sample a noise vector $z$ from $\mathcal{N}(0,1)$
8              $\tilde{x} \leftarrow G(z)$
9              $\hat{x} \leftarrow \epsilon x + (1-\epsilon)\tilde{x}$
10             $W_i^{(D)} \leftarrow -D(x) + D(\tilde{x}) + \lambda(\|\nabla_{\hat{x}} D(\hat{x})\|_2 - 1)^2$
11         $\theta^{(D)} \leftarrow Adam(\nabla_{\theta^{(D)}} \frac{1}{m}\sum_{i=1}^{m} W_i^{(D)})$
12     for i = 1,...,m do
13         Sample a noise vector $z$ from $\mathcal{N}(0,1)$
14         $J_i^{(G)} \leftarrow -D(G(z))$
15     $\theta^{(G)} \leftarrow Adam(\nabla_{\theta^{(D)}})\frac{1}{m}\sum_{i=1}^{m} J_i^{(G)}$
```

Chapter 7

Implementation of the Proposed GAN

The following chapter deals with the implementation details of the generative adversarial neural network model for skin diseased fingerprint generation proposed in the previous Chapter 6. The chapter first introduces all of the technologies and frameworks used in the implementation of the neural network models. Then it subsequently mentions the implementation details of the discriminator and generator network modules as well as the GAN as a whole.

7.1 Technologies Used in the Implementation

The following sections details the technologies utilized in the implementation of the GAN model. The programming language of choice is Python v3.7.10. I chose to implement the GAN in Jupyter Notebook v5.2.2 in order to be able to utilize the free cloud computing platform of Google Colab for the training of the network. Google Colab provides users with access to K80, T4, P4 and P100 GPUs, which were used for hardware acceleration of the computations via NVIDIA's CUDA v11.0. The GAN network itself is implemented using the TorchGAN library v0.0.4 and the PyTorch framework v1.8.1, which has cuDNN v10.1 support.

7.1.1 Python

Python is an interpreted, interactive, object-oriented, high-level programming language. It features dynamic typing, high-level data types, late binding, reference counting and cycle detecting garbage collector. Python also supports other programing paradigms other than object-oriented in the form of procedural and functional programming. The programming language is focused on readability and flexibility, making it suitable for rapid prototyping and development.

Python features an extensive and powerful standard library providing a range of standardized solutions to a wide variety of programming problems. In addition to the standard library, the Python ecosystem provides a range of easy-to-use packages and modules for numerical analysis, statistics, data analysis and visualization. All of the above-mentioned properties of the Python language make it well suited for development of machine learning tools and applications.

7.1.2 Project Jupyter

Project Jupyter [52] is an open-source software for interactive scientific computing, research and data science across a variety of programming languages. Jupyter started as a spin-off from the *IPython* project in 2014, after IPython evolved into a generic architecture for interactive computing in any programming language [52]. The language-agnostic parts of IPython (protocol, qtconsole, web application etc.) moved under the new name of Jupyter, while IPython stayed as the Python shell and kernel for the Jupyter projects.

7.1.3 Jupyter Notebook

Jupyter Notebook [33] is a language-agnostic HTML notebook application for Project Jupyter. It is a free, open-source, web-based computing platform for authoring and executing Jupyter Notebook documents. Jupyter Notebook documents combine code with narrative text, mathematical equations, interactive user interfaces, images, videos etc., which integrate together to create an interactive document. These documents are structured data that represent content, code, inputs, outputs and metadata and they are represented as JSON files with the '*.ipynb*' extension. The Jupyter Notebook documents consist of a sequence of three different types of cells:

1. **Code cells** are the primary content of the Jupyter Notebook documents, they allow the user to write and edit code with full syntax highlighting. The programming language used in these code cells is dependent on the choice of Jupyter kernel. The default kernel is IPython kernel and Python programming language and there is a large number of kernels for other supported languages. During the execution of a code cell, the code contained in the cell is sent to the kernel of the Jupyter Notebook. The results of the execution are displayed in the code cell as the cell's output.
2. **Markdown cells** are used to render formatted, stylized text as defined by the Markdown markup language. The Markdown cells can also include attachments, mathematical equations and embedded code. The execution of a Markdown cell results in the corresponding formatted rich text as defined by the cell's contents.
3. **Raw cells** are not evaluated by the Jupyter kernel and the notebook does not render them. Their main purpose is to be included unmodified in the output of Jupyter's NBConvert tool, which is used to convert the notebook to other static formats (HTML, LaTeX, PDF).

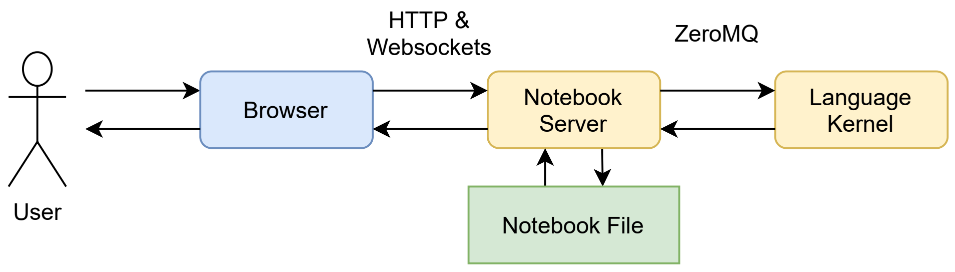


Figure 7.1: Interface of the Jupyter Notebook. Notebook server is responsible for saving loading and editing of the notebook files. The Kernel runs independently and interacts with the server, it gets send cells of code for execution.

7.1.4 Google Colaboratory

Google Colaboratory (or Google Colab) [4] is a product from Google Research that provides hosted Jupyter Notebook service without requiring any additional setup to use. Google Colab stores the Jupyter Notebooks stored in Google Drive and can execute them by connecting to a cloud-based or a local runtime.

The Google Colab is well suited testing and development of machine learning and deep learning applications since the service provides free accelerated computing environments with GPUs and TPUs. Access to the resources provided by Google Colab varies over time, as the service prioritizes users who have used fewer resources in the recent time frame to prevent monopolization of Colab's resources by its users. However, the paid subscription in the form of Colab Pro offers consistent high-priority access to the resources for extended periods of time without limits on the maximum lifetimes of VMs.

7.1.5 PyTorch

PyTorch [51] is an open-source machine learning framework developed by Facebook's AI Research. PyTorch provides imperative, Pythonic programming style with maximum flexibility and speed. The flexibility and ease of use of PyTorch, in comparison to other machine learning libraries, have presently made it the most widely used machine learning framework [64].

PyTorch is based on the Lua machine learning library named Torch. PyTorch is a Python package, but even though the main focus of the project is the Python interface, there are C/C++, Lua and Java APIs available to the users as an alternative.

PyTorch is a dynamic, define-by-run framework. The function to be differentiated in PyTorch is defined by forward pass of the desired computation, as opposed to specifying a static graph structure [50]. This means that PyTorch relies on dynamic computational graphs, which can vary in subsequent iterations.

Pytorch utilizes immediate, eager execution. It never records a so called "forward computation graph". Pytorch only ever runs tensor computations as it encounters them and records solely the information necessary to differentiate the computation. This facilitates the automatic differentiation in PyTorch via the built-in engine called `torch.autograd`, which enables automatic calculation of gradient necessary for back propagation. One disadvantage of not having a forward computation graph is that PyTorch gives up whole-network optimization and batching. [50]

Another feature of PyTorch is its support for GPU-accelerated computation and recently also Cloud TPU computation. To utilize PyTorch with GPU it is necessary to have the GPU version of the framework along with a CUDA-enabled GPU or a GPU that supports ROCm. TPU acceleration is achieved through PyTorch/XLA, which is an integration of the XLA (Accelerated Linear Algebra) with PyTorch that allows users to connect the deep learning framework with Cloud TPUs.

7.1.6 CUDA Toolkit

Compute Unified Device Architecture (CUDA) is a parallel computing platform and programming model developed by NVIDIA. It is used for general computing acceleration through the use of GPUs. In order to utilize CUDA it is necessary to have access to an NVIDIA GPU that is CUDA-enabled. The CUDA platform allows its users to offload computationally intensive computations to a single or potentially multiple GPUs.

Many of the deep learning frameworks, PyTorch included, rely on CUDA for their GPU support. Most of these frameworks utilize NVIDIA's cuDNN deep neural network library, which provides highly optimized implementations of standard computations that are commonly used in deep neural networks (e.g. convolution, normalization, activation functions).

Utilizing GPU-acceleration through use of CUDA is implemented in very straightforward manner in PyTorch. They allow the users to check whether CUDA and a CUDA-enabled GPU are available through `torch.cuda.is_available()`. If at least one is available, it is possible to get an object representing a given GPU device through command `torch .device('cuda:n')`, where n is the GPU ID. Afterwards, it is easy to move the model or data to the GPU by `.to(device)` command. Once the model or data are on a GPU you can do operations on them, which are by default asynchronous and the results are placed on the same device.

7.1.7 TorchGAN

TorchGAN [49] is Python machine learning framework based on PyTorch for designing, implementing and evaluating GANs. The aim of the TorchGAN project is to develop an easy mechanism to combine a variety of techniques from a number of different GAN papers in order to help with quick experimentation and rapid prototyping. TorchGAN provides built-in support for a number of popular GAN building blocks, models, loss functions and evaluation metrics which allow for easy extensibility and customization. Additionally, the framework incurs little to no overhead compared to vanilla PyTorch and can be integrated with it seamlessly.

7.1.8 TensorBoard

TensorBoard is a free open-source visualization and measurement toolkit for machine learning experimentation and development. TensorBoard is intrinsically meant for the TensorFlow library, but it can be integrated with other machine learning frameworks. The visualization toolkit allows tracking of metrics (e.g. loss, performance), model visualization, displaying images and histograms.

To facilitate sharing of the tracked metrics the TensoBoard team launched Tensorboard.dev service. TensorBoard.dev is a managed service that provides interactive web-based dashboard for the data logged over the course of training the machine learning model. TensorBoard.dev runs on App Engine and reads data that users have uploaded to the hosted service, which are subsequently visualized for the users.

7.1.9 TensorBoardX

TensorBoardX enables integration of the TensorBoard visualization tool with the PyTorch framework. This package allows users to utilize simple interface for logging events and metrics of PyTorch machine learning models. TensorBoardX provides a high-level API for creation and asynchronous update of summaries and event log files through its SummaryWriter class. The SummaryWriter creates log files in the TensorBoard log file format, which can be subsequently visualized using TensorBoard.dev web server or TensorBoard.

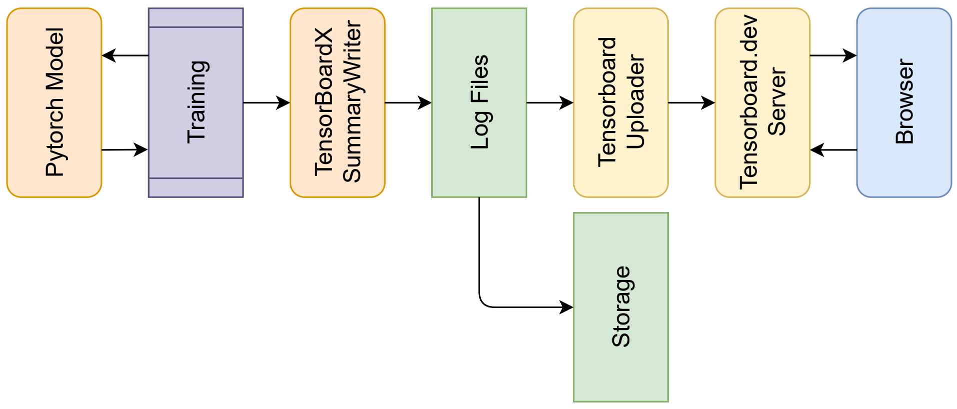


Figure 7.2: Visualizing metrics of a PyTorch model logged using the TensorBoardX package during the training.

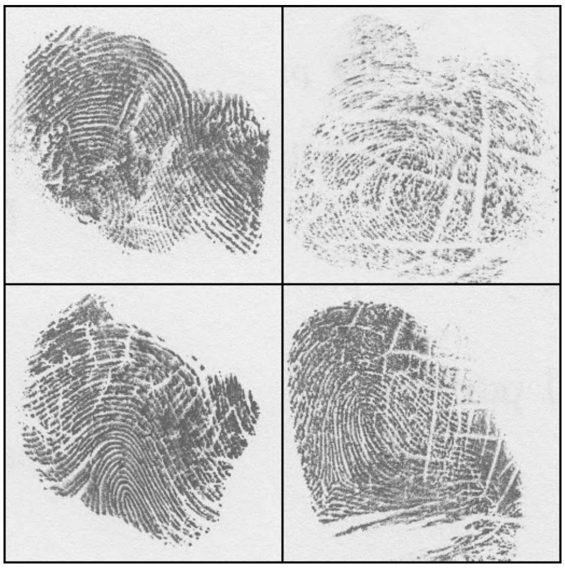
7.2 Data Preprocessing and Preparation

The preparation of the training dataset, its preprocessing and augmentation are done using the `torch.utils.data` package. To get the images of the training dataset into the Google Colab runtime, they can be either uploaded to the directly from a local machine, copied from an existing Google Drive location or cloned from a Git repository.

Once the training dataset is in the Google Drive of the Colab runtime, the root directory of the dataset is passed to the `torchvision.datasets.ImageFolder`, which is a generic dataset representation provided by the PyTorch library. It internally organizes the images in the dataset according to the structure of subdirectories of the root dataset directory. The `ImageFolder` provides samples from the training dataset when they are requested and applies any and all necessary transformations that are specified using the `torchvision.transforms` package. The transformation can be composed together and applied in sequence. In addition to the data augmentation transformations mentioned in Subsection 6.1, grayscale transformation, conversion to a PyTorch tensor and normalization of values to a range of $< -1, 1 >$ are applied to the images of the dataset.

The region of interest in the real fingerprint images from the training dataset varied a fair amount, due to variance in surface contact area, friction and pressure during fingerprint deposition. To facilitate easier training and faster convergence of the GAN network, the real sample images of the training dataset were cropped to only contain the central region of the fingerprint.

The `torch.utils.data.DataLoader` is then used to actually sample the images from the training dataset and organize them into batches of size 64. The `DataLoader` acts as a wrapper around the dataset object an iterable over the training dataset. It runs on the CPU with 2 workers preparing and loading the data in parallel. It should be noted that it was necessary to provide the workers with a random seed that is different for each worker, which they can used to initialize their random number generator. Their default behavior in PyTorch would be that both workers would generate identical random transformations (e.g random crop) instead. The `DataLoader` is then subsequently provided to the WGAN-GP which uses it to sample and load batches of data necessary for training iterations.


(a) Original Images

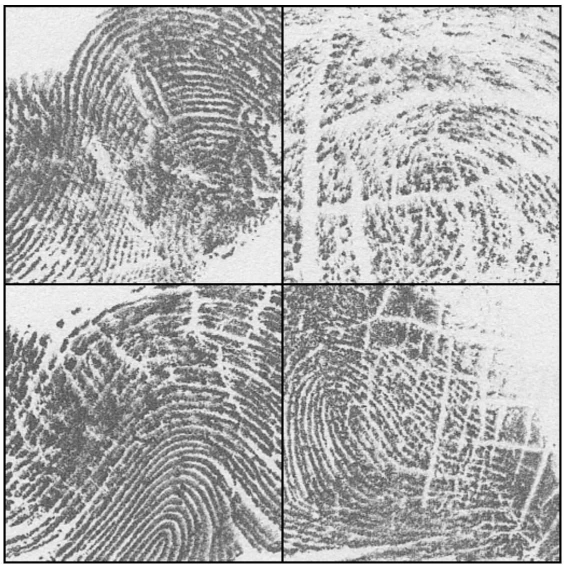

(b) Transformed Images

Figure 7.3: Comparison of a sample fingerprint image taken from the training dataset before and after application of transformations used for preprocessing and augmentation of the dataset.

7.3 Wasserstein GAN with Gradient Penalty

The Wasserstein generative adversarial network with gradient penalty which was proposed in Section 6.2 is implemented in class `WGANGP`. The `WGANGP` class facilitates the training of the GAN and logging of the necessary summary information.

The `WGANGP` constructor takes a large number of arguments. Firstly, it takes instances of `Discriminator` and `Generator` classes along with a pair of `torch.optim.Adam` optimizers, one for the parameters of each respective convolutional neural network. Secondly, size of latent dimensions z for the generator, number of training steps of the `Discriminator` for each training step of the `Generator` and the type of device on which the GAN training should run. Lastly, name of the training dataset, log file location, save file location for the checkpoints of the two network models along with the number of training epochs elapsed before their states should be saved again.

The training of the GAN model is implemented in the method `train`, which takes a PyTorch `DataLoader` and number of training epochs as arguments. The `DataLoader` prepares batches of training images and uses them to alternately train the models of `Discriminator` and `Generator` network, as detailed in Algorithm 1. A single training epoch consists of 3 training steps of the `Discriminator` and a single training step of the and `Generator`. The parameters of both these models are updated using the pair of `torch.optim.Adam` optimizers provided to the constructor. The states of both models are saved periodically using, after a sufficient number of epochs were completed. The trainable parameters of the model are serialized using the `torch.save` function into a '*.pt*' file. The expected total size of both models is approximately 110 MB. These models should be ideally saved to a Google Drive location or downloaded to a local machine to prevent the loss of the save files when the Google Colab VM session is terminated.

In addition to GAN training, the WGANGP class is also responsible for logging information about the training process of the WGAN-GP model. This is accomplished using the SummaryWriter from torch.utils.tensorboard package. The SummaryWriter allows logging of data which can be later visualized using the TensorBoard tool. The scalar values of generator's loss function, discriminator's loss function and discriminator's gradient penalty are recorded and saved to the log directory location so that they can be plotted in the form of line plots and histograms. A grid consisting of sample images created by the Generator are also saved at each checkpoint of the GAN's training.

7.3.1 Pytorch Module

The PyTorch class torch.nn.Module, not to be confused with a Python module, is the base class for all neural network models in this machine learning framework. All models implemented in PyTorch should typically inherit from the nn.Module class as it provides a number of attributes and methods essential for the machine learning model, e.g. .parameters(), .zero_grad() All instances of the Module class in PyTorch can be contained within other instances of the Module allowing them to act as submodules.

The state_dict attribute of the Module class contains all of the learnable parameters and persistent buffers of the machine learning model allowing for simple checkpointing and loading of the model. The state_dict is simply a Python dictionary that maps each of the layers in the Module to their respective parameter tensors.

7.3.2 Discriminator Network

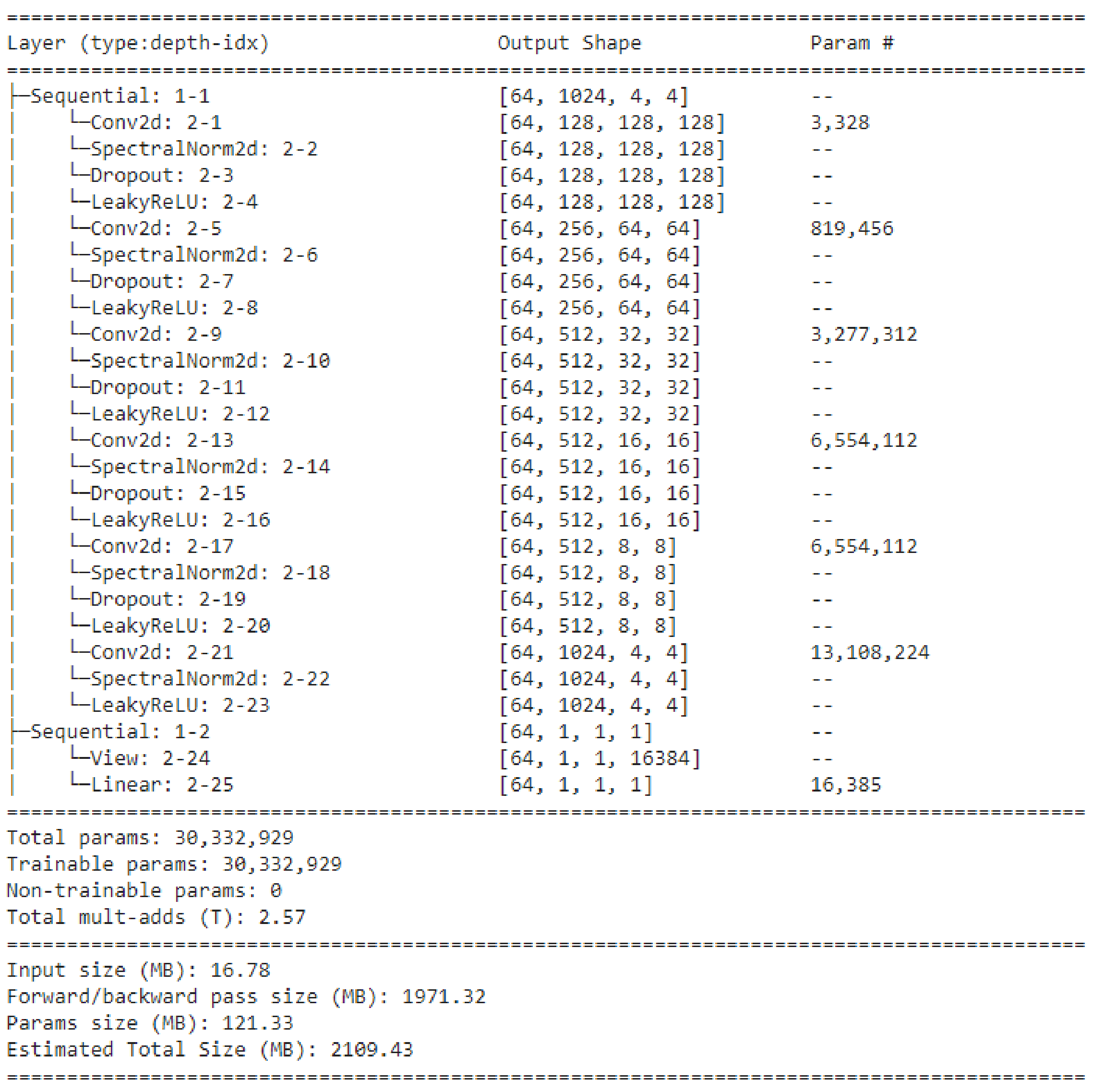
The convolutional neural network of the discriminator is implemented in the Discriminator class. It is a subclass of the torch.nn.Module and DropoutModule classes. The first class torch.nn.Module is responsible for keeping the current state of the discriminator module as well as backwards hooks necessary for the computation of gradient of the function $D(x)$ represented by the discriminator network used in backpropagation. The DropoutModule class facilitates addition of dropout layers to the module and keeping tracks of dropout layers already present within the module in order to be able to enable/disable them as necessary during the training and evaluation of the discriminator model.

The model itself is implemented as outlined in Figure 6.2. Internally, the discriminator network consists of two torch.nn.Sequential containers named main_module and output. The main_module contains a sequence of PyTorch Conv2D convolutional layers followed by TorchGAN's implementation of spectral normalization SpectralNorm2D and application of torch.nn.Dropout and torch.nn.LeakyReLU activation function.

While the output, which is appended at the end of the sequence of convolution layers, contains a View module and a fully connected torch.nn.Linear layer. The View module first flattens the multidimensional feature map data from the convolutional layers and feeds them to the fully connected layer which approximates the Wasserstein distance. The hyperparameters of the above-mentioned modules are set in accordance with the hyperparameter settings proposed in Section 6.2.

The Discriminator class also implements the forward() method, which calculates the output of the discriminator network for a batch of data samples. The method takes as an input a 4-dimensional tensor of size (*batch_size*, *channels*, *img_width*, *img_height*), where *batch_size* is 64, number of channels in the input image is 1 and both height and width of the image are 256. The output of the forward() method is the result of passing the input tensor through the sequences of nn.Modules inside the main_module and output containers. The result is a 4-dimensional tensor of size $(batch\_size, 1, 1, 1)$ containing $D(x)$ computed for each of the samples in the input batch of data.

For reference regarding the layer structure, expected feature map input/output size, total number of model parameters and size of the discriminator model see Figure 7.4.



```
==========================================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================================
├─Sequential: 1-1                        [64, 1024, 4, 4]          --
│    └─Conv2d: 2-1                       [64, 128, 128, 128]       3,328
│    └─SpectralNorm2d: 2-2               [64, 128, 128, 128]       --
│    └─Dropout: 2-3                      [64, 128, 128, 128]       --
│    └─LeakyReLU: 2-4                    [64, 128, 128, 128]       --
│    └─Conv2d: 2-5                       [64, 256, 64, 64]         819,456
│    └─SpectralNorm2d: 2-6               [64, 256, 64, 64]         --
│    └─Dropout: 2-7                      [64, 256, 64, 64]         --
│    └─LeakyReLU: 2-8                    [64, 256, 64, 64]         --
│    └─Conv2d: 2-9                       [64, 512, 32, 32]         3,277,312
│    └─SpectralNorm2d: 2-10              [64, 512, 32, 32]         --
│    └─Dropout: 2-11                     [64, 512, 32, 32]         --
│    └─LeakyReLU: 2-12                   [64, 512, 32, 32]         --
│    └─Conv2d: 2-13                      [64, 512, 16, 16]         6,554,112
│    └─SpectralNorm2d: 2-14              [64, 512, 16, 16]         --
│    └─Dropout: 2-15                     [64, 512, 16, 16]         --
│    └─LeakyReLU: 2-16                   [64, 512, 16, 16]         --
│    └─Conv2d: 2-17                      [64, 512, 8, 8]           6,554,112
│    └─SpectralNorm2d: 2-18              [64, 512, 8, 8]           --
│    └─Dropout: 2-19                     [64, 512, 8, 8]           --
│    └─LeakyReLU: 2-20                   [64, 512, 8, 8]           --
│    └─Conv2d: 2-21                      [64, 1024, 4, 4]          13,108,224
│    └─SpectralNorm2d: 2-22              [64, 1024, 4, 4]          --
│    └─LeakyReLU: 2-23                   [64, 1024, 4, 4]          --
├─Sequential: 1-2                        [64, 1, 1, 1]             --
│    └─View: 2-24                        [64, 1, 1, 16384]         --
│    └─Linear: 2-25                      [64, 1, 1, 1]             16,385
==========================================================================================
Total params: 30,332,929
Trainable params: 30,332,929
Non-trainable params: 0
Total mult-adds (T): 2.57
==========================================================================================
Input size (MB): 16.78
Forward/backward pass size (MB): 1971.32
Params size (MB): 121.33
Estimated Total Size (MB): 2109.43
==========================================================================================
```

Figure 7.4: Summary of the discriminator's model architecture detailing the expected input and output tensor shapes at each layer of the convolutional neural network alongside the total number of trainable parameters of the model.

7.3.3 Generator Network

The convolutional neural network of the generator is implemented in the `Generator` class. Similarly to the `Discriminator` class, `Generator` inherits from the `torch.nn.Module` and `DropoutModule` classes which fulfill functions identical to the ones outlined in the previous section. The internal structure of the generator model is implemented as outlined in Figure 6.4. The convolutional neural network of the generator consists of three `torch.nn.Sequential` module containers named `reshape`, `main_module` and `output`.

The `reshape` container consists of the first fully connected layer of the network which is implemented using `torch.nn.Linear` module. The output of this fully connected layer is subsequently reshaped into a 4-dimensional feature map of size $(batch\_size, 1024, 4, 4)$ so that it can be passed onto the rest of the network. This is followed by the application of batch normalization `torch.nn.BatchNorm2d` and activation function `torch.nn.ReLU`.

The `main_module` consists of a sequence of upsampling convolutional layers which use nearest neighbor `torch.nn.Upsample` by the factor of 2 followed immediately by regular `torch.nn.Conv2d`. This sequence of upsampling layers uses both `torch.nn.BatchNorm2d` and `torch.nn.Dropout` as well as application of the `torch.nn.ReLU` activation function.

The `output` sequential contains a single resize convolution upsampling layer that uses `torch.nn.BatchNorm2d`, but does not use `torch.nn.Dropout`, which reduces the high number of feature maps in its input to a single feature map of size 256×256 representing the generated grayscale image.

The `Generator` class implements the `forward()` method for generation of fake sample images from random vectors of latent variables. The `forward()` method takes as an input a single 4-dimensional tensor of size $(batch\_size,\ 1,\ 1,\ latent\_dim)$ containing all vectors of latent variables that are to be used for generating images in a given batch. The $batch\_size$ being 64 and the number of $latent\_dim$, number of dimensions in vectors of latent variables, being at least 256.

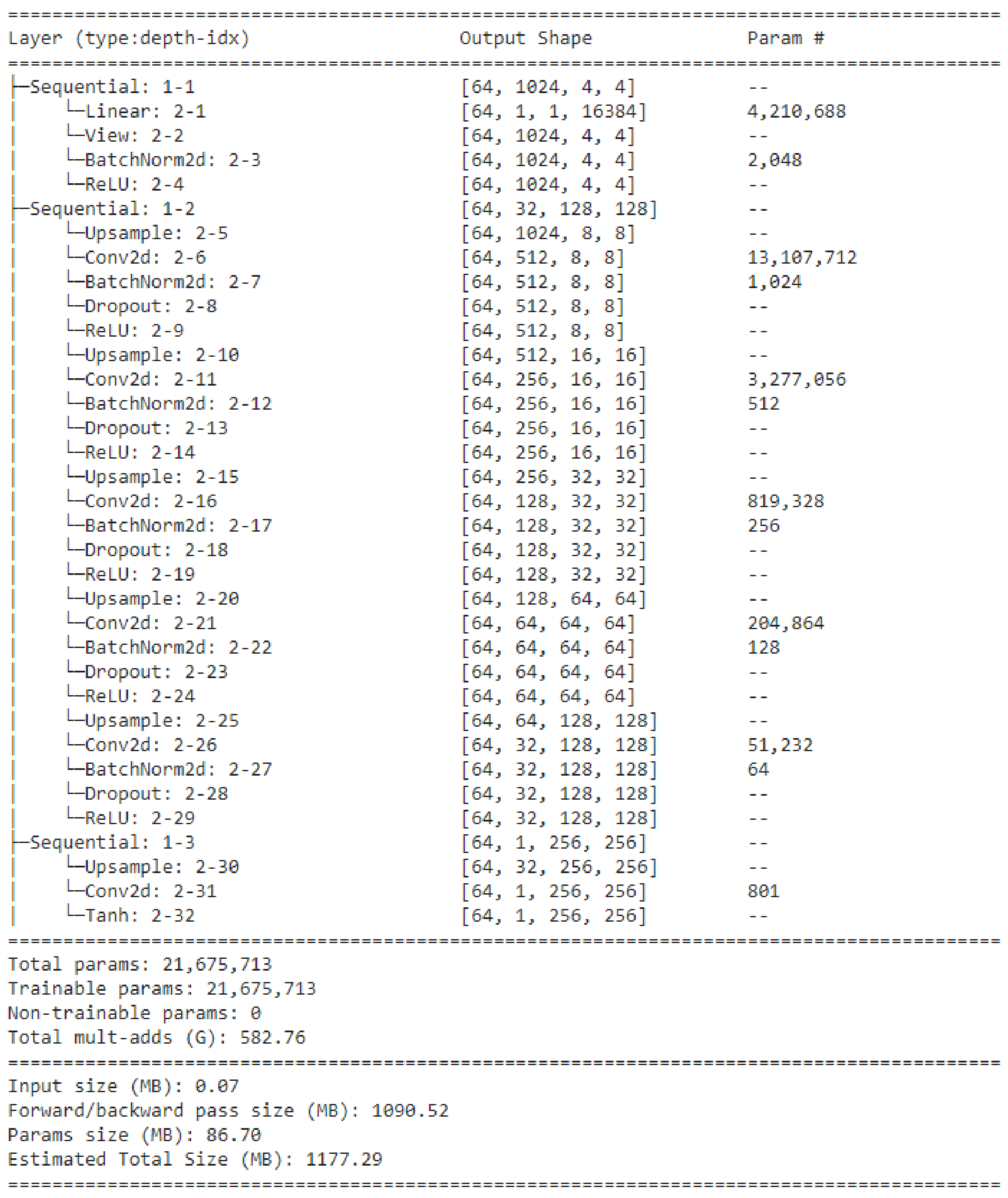
The summary of the model's internal layer structure, feature map inputs and outputs, sum total of model's parameters and total model size is shown in Figure 7.5.

7.3.4 Model Checkpointing

Training the model of the WGAN until it starts producing reasonable results usually takes a number of hours, if not days. Google Colab, and other similar cloud computing services, typically limit the maximum lifespan of VMs that they provide for computation. This means that if a run of the network was to stop unexpectedly due to these constraints, all of the model's training would be lost.

Checkpointing is a fault tolerance technique for long running processes. It allows the model to be trained over a number of sessions and prevents loss of progress in case of an abrupt stop. All that is necessary is to save the trainable parameters (weights) of the discriminator and generator networks along with running averages of their respective Adam optimizers. Every `torch.nn.Module` in PyTorch has a `state_dict` dictionary object mapping each network layer to its respective tensor of trainable parameters. Similarly, the `state_dict` of the optimizers contains their hyperparameters and current running averages.

These objects are serialized using the `torch.save()` function, which uses the pickle utility, and stores them in files with `'.pt'` or `'.pth'` file extension. These stored checkpoints can be subsequently used to restore the model to its previous state and resume training of the model by deserializing the pickled files using `torch.load()` and `load_state_dict()`.



```
==========================================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================================
├─Sequential: 1-1                        [64, 1024, 4, 4]          --
│    └─Linear: 2-1                       [64, 1, 1, 16384]         4,210,688
│    └─View: 2-2                         [64, 1024, 4, 4]          --
│    └─BatchNorm2d: 2-3                  [64, 1024, 4, 4]          2,048
│    └─ReLU: 2-4                         [64, 1024, 4, 4]          --
├─Sequential: 1-2                        [64, 32, 128, 128]        --
│    └─Upsample: 2-5                     [64, 1024, 8, 8]          --
│    └─Conv2d: 2-6                       [64, 512, 8, 8]           13,107,712
│    └─BatchNorm2d: 2-7                  [64, 512, 8, 8]           1,024
│    └─Dropout: 2-8                      [64, 512, 8, 8]           --
│    └─ReLU: 2-9                         [64, 512, 8, 8]           --
│    └─Upsample: 2-10                    [64, 512, 16, 16]         --
│    └─Conv2d: 2-11                      [64, 256, 16, 16]         3,277,056
│    └─BatchNorm2d: 2-12                 [64, 256, 16, 16]         512
│    └─Dropout: 2-13                     [64, 256, 16, 16]         --
│    └─ReLU: 2-14                        [64, 256, 16, 16]         --
│    └─Upsample: 2-15                    [64, 256, 32, 32]         --
│    └─Conv2d: 2-16                      [64, 128, 32, 32]         819,328
│    └─BatchNorm2d: 2-17                 [64, 128, 32, 32]         256
│    └─Dropout: 2-18                     [64, 128, 32, 32]         --
│    └─ReLU: 2-19                        [64, 128, 32, 32]         --
│    └─Upsample: 2-20                    [64, 128, 64, 64]         --
│    └─Conv2d: 2-21                      [64, 64, 64, 64]          204,864
│    └─BatchNorm2d: 2-22                 [64, 64, 64, 64]          128
│    └─Dropout: 2-23                     [64, 64, 64, 64]          --
│    └─ReLU: 2-24                        [64, 64, 64, 64]          --
│    └─Upsample: 2-25                    [64, 64, 128, 128]        --
│    └─Conv2d: 2-26                      [64, 32, 128, 128]        51,232
│    └─BatchNorm2d: 2-27                 [64, 32, 128, 128]        64
│    └─Dropout: 2-28                     [64, 32, 128, 128]        --
│    └─ReLU: 2-29                        [64, 32, 128, 128]        --
├─Sequential: 1-3                        [64, 1, 256, 256]         --
│    └─Upsample: 2-30                    [64, 32, 256, 256]        --
│    └─Conv2d: 2-31                      [64, 1, 256, 256]         801
│    └─Tanh: 2-32                        [64, 1, 256, 256]         --
==========================================================================================
Total params: 21,675,713
Trainable params: 21,675,713
Non-trainable params: 0
Total mult-adds (G): 582.76
==========================================================================================
Input size (MB): 0.07
Forward/backward pass size (MB): 1090.52
Params size (MB): 86.70
Estimated Total Size (MB): 1177.29
==========================================================================================
```

Figure 7.5: Summary of the generator's model architecture detailing the expected input and output tensor shapes at each layer of the convolutional neural network alongside the total number of trainable parameters of the model.

Chapter 8

Results Evaluation

This chapter details the results achieved by the implemented WGAN-GP model with regard to the quality of the synthetically generated fingerprint images. The decisions taken with regards to selection of training datasets and training samples and an overview of the training process is discussed in the following chapter. The resulting datasets of synthetically generated fingerprints are showcased and compared with real fingerprints influenced by skin diseases using a number of different fingerprint quality assessment metrics.

8.1 Model Training and Dataset Generation

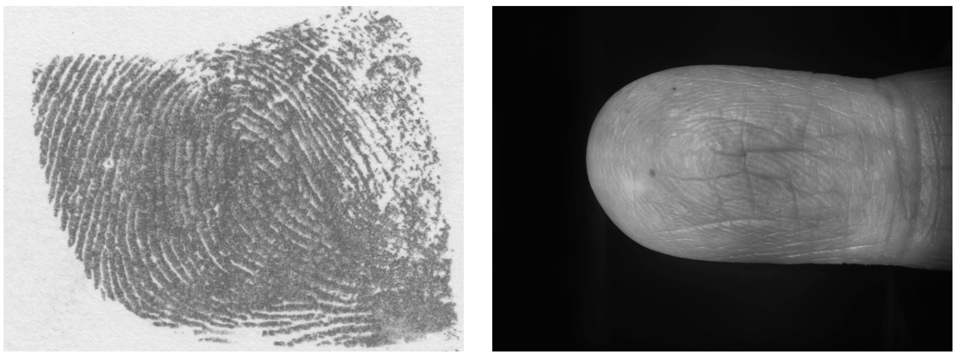
Individual run of the model's training process was conducted for each type of selected skin disease. The training was focused on three different types of skin disease with most numerous representation in the diseased fingerprint database: *atopic eczema*, *psoriasis* and *dyshidrotic eczema*. Because of the highly varied nature of fingerprint images in the database, the model was trained on manually selected subsets of diseased fingerprints. The quality of fingerprint samples fluctuates due to both quality of fingerprint acquisition and the extent of damage cause by the skin disease. The selected samples consisted of some of the less damaged images with similarly shaped regions of interest. The goal of using such a selection of training samples is to slightly reduce sample diversity and allow the model to more easily reproduce a semblance of papillary line structure in the generated images. However, this approach does come with the disadvantage of further shrinking an already small training dataset.

The WGAN-GP models were trained over the course of thousands of epochs for extended periods of time. The progress was monitored manually through the loss of the generator and discriminator networks along with periodic generation of sample images every 100 training epochs. If the quality of the generated images stopped improving or the model didn't seem to make any meaningful progress over a number of epochs, the training was stopped.

The models were trained using the Google Colaboratory cloud computing service. Majority of the training took place on VMs with two Intel Xeon 2.20 GHz processors, 25 GiB of RAM and an NVIDIA Tesla P100 PCIe GPU with 16 GiB of memory. Google Colab also offers access to TPUs, e.g. the Cloud TPUv2 with 64 GiB of high bandwidth memory, but using the TPUs did not appear to bring any benefits in terms of performance increase. The TPUs take advantage of larger batch sizes, e.g. 1024+, where their performance is comparable, if not straight up better than that of available GPUs. However, such large batch sizes are not able to be used in this case due to the total size of the training datasets.

8.1.1 Training Datasets

The models were trained on subsets of real fingerprint image samples from the diseased fingerprint database detailed in Section 3.3. The samples contained fingerprint images of varying type and quality depending on the severity of the disease and the manner in which the fingerprint was acquired. The fingerprint images were acquired with a range of sensors, but the highest number of samples were obtained from optical sensors and from inked fingerprints by optical sensors.


(a) inked and optical

(b) optical

Figure 8.1: Samples of two most prevalent types of fingerprint images obtained by two different fingerprint acquisition methods.

The training of the WGAN model was mainly focused on the fingerprint images from inked samples. I have chosen to forgo training the network on the optical images as training a higher resolution network is significantly more computationally expensive and complicated. These optical images do not lend them selves well to smaller resolutions, as the minuscule details of the papillary lines are not very easily differentiable in smaller image sizes. Additionally, these fingerprint images were acquired with a varying degree of lighting, with multiple different finger positions and rotations. All of the above-mentioned reasons pose serious difficulty in generating synthetic fingerprint images of sufficient quality when trying to train GAN on datasets of limited size.

The diseased fingerprint database also includes fingerprints gathered by a capacitative fingerprint sensor and sweep fingerprint sensor. These sample images represent a much smaller portion of the diseased fingerprint database. These types of images were not used for training of the GAN as it is difficult to achieve any sort of reasonable results on such a highly varied dataset with really small training datasets. The capacitative sweep sensor images are particularly unsuitable for model training as the captured images are of highly varied image size. The height of these images is particularly varied due to the manner in which they were acquired.

8.1.2 Model Training Summary

With the primary focus of the training being inked fingerprint images, I have first attempted to train the proposed WGAN model on these fingerprint images in their entirety. The training dataset consisted of less damaged fingerprint images influenced by atopic eczema where the papillary line structure was still visible for the most part. These images were uncropped, downscaled to size 256×256 with fingerprint image background included. While the model was able to learn to generate images that contained the general shape of a fingerprint image similar the ones in the training dataset. It did not learn to produce any semblance of a well-formed papillary line structure. The training of the model was stopped after 3500 epochs as the model was not making any meaningful progress. To see a set of images sampled during the training process refer to Figure 8.3.

Having found that using the fingerprint images in their entirety lead to difficulties when training the proposed WGAN model, I have subsequently attempted to minimize the amount of fingerprint image background in the training datasets. The model was trained on less damaged fingerprint images from the atopic eczema dataset with central crops from the fingerprint region of interest. The general idea was to help improve convergence of the WGAN model by simply having the model focus on successfully replicating the papillary line structure and any damage caused to it without needing to accurately generate the general shape of the fingerprint image. The WGAN was then trained on these 256×256 images which contained the core parts of the sample fingerprint image. It can seen in Figure 8.4 that the convergence of the model has improved and that the generated images were starting to form a coherent papillary line structure.

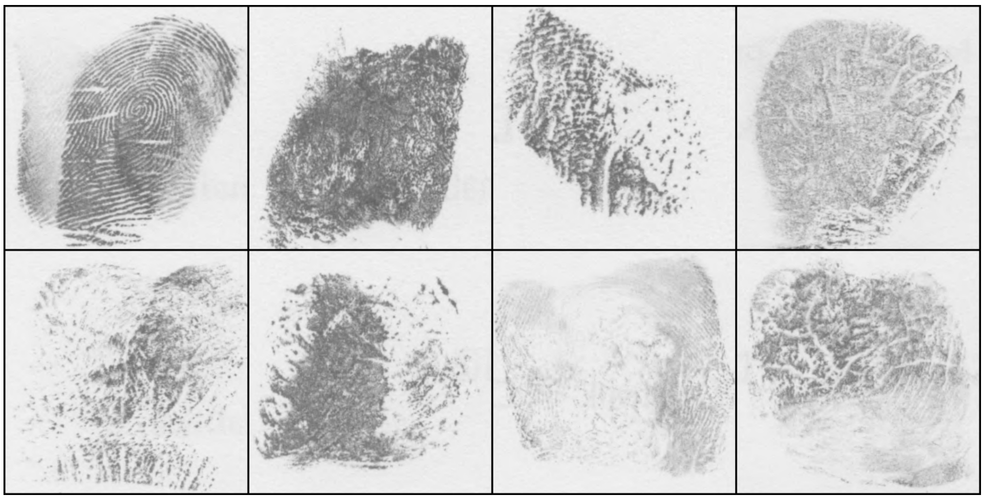


Figure 8.2: Examples of some of the samples excluded from the training datasets. Eliminating these types of highly diverse outliers promotes faster convergence of the model and helps the model form a more rigid papillary line structure.

Another issue that plagued the training process was the fact that images generated by the model often times did not have continuous ridge line directions around heavily damaged areas. If a papillary line was interrupted by a large straight line or a large irregular dark spot, there was no guarantee that it would continue in a similar direction. Some of the generated fingerprint did not even have a well formed core and the papillary lines would run in all sorts of direction from the center. Evidence of this can be seen in Figure 8.5.

What seemed to help immensely was excluding heavily damaged fingerprint samples from the training dataset during early stages of the training process, see Figure 8.2. These fingerprints that carried little to no information about papillary line structure hindered the model's ability to successfully recreate undamaged parts of the fingerprint. These heavily damaged fingerprints can be readded to the training dataset at later stages of the training process, when the model learned to generate adequately structured fingerprint, to let the model learn to replicate this extensive damage caused by the skin disease.

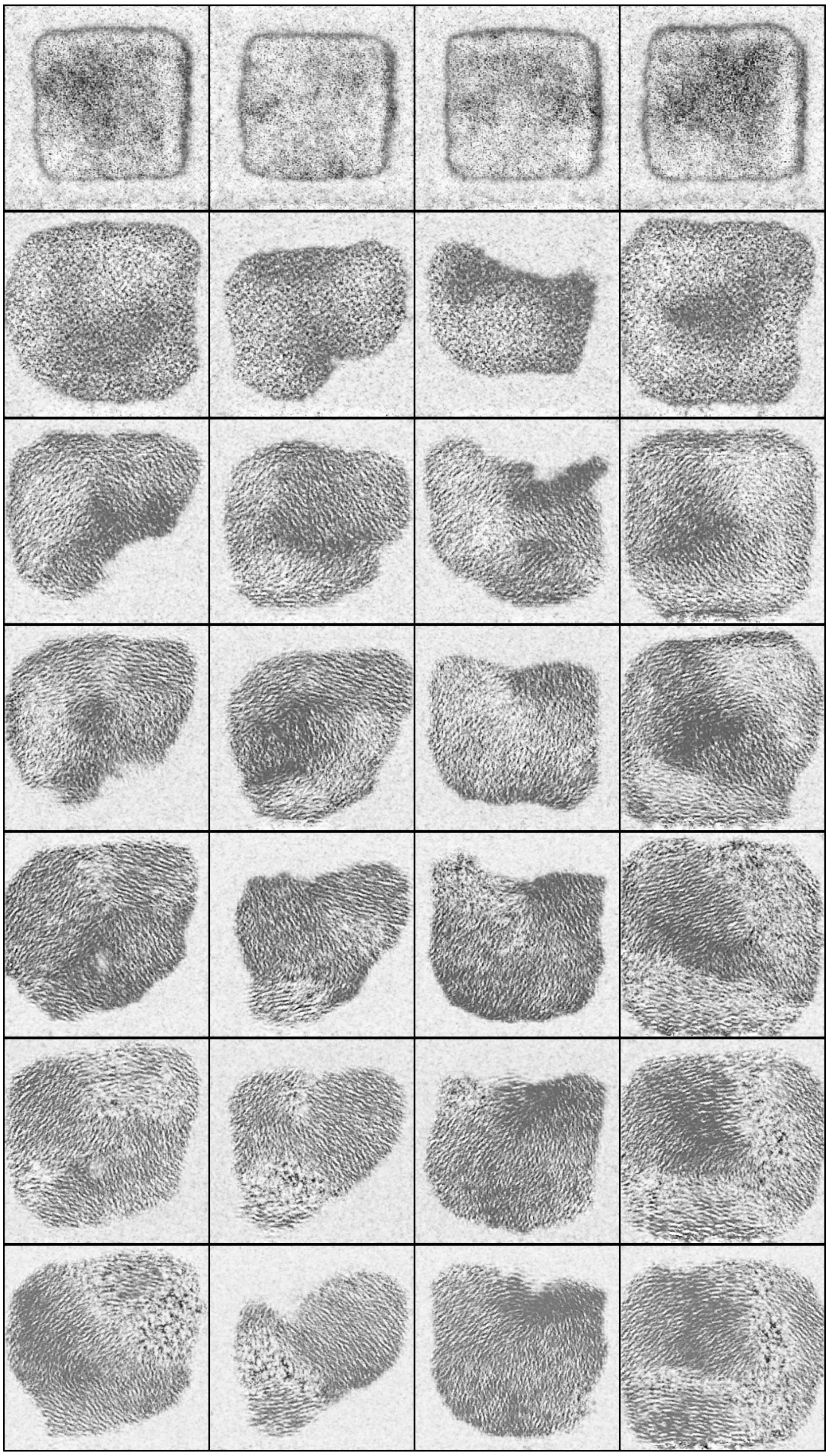


Figure 8.3: Failure to converge during training of the WGAN when using uncropped fingerprint images influenced by atopic eczema. The first samples are taken from 100-th epoch and the rest of the samples are taken at intervals of 500 epochs starting at $1000, 1500, ..., 3500$.

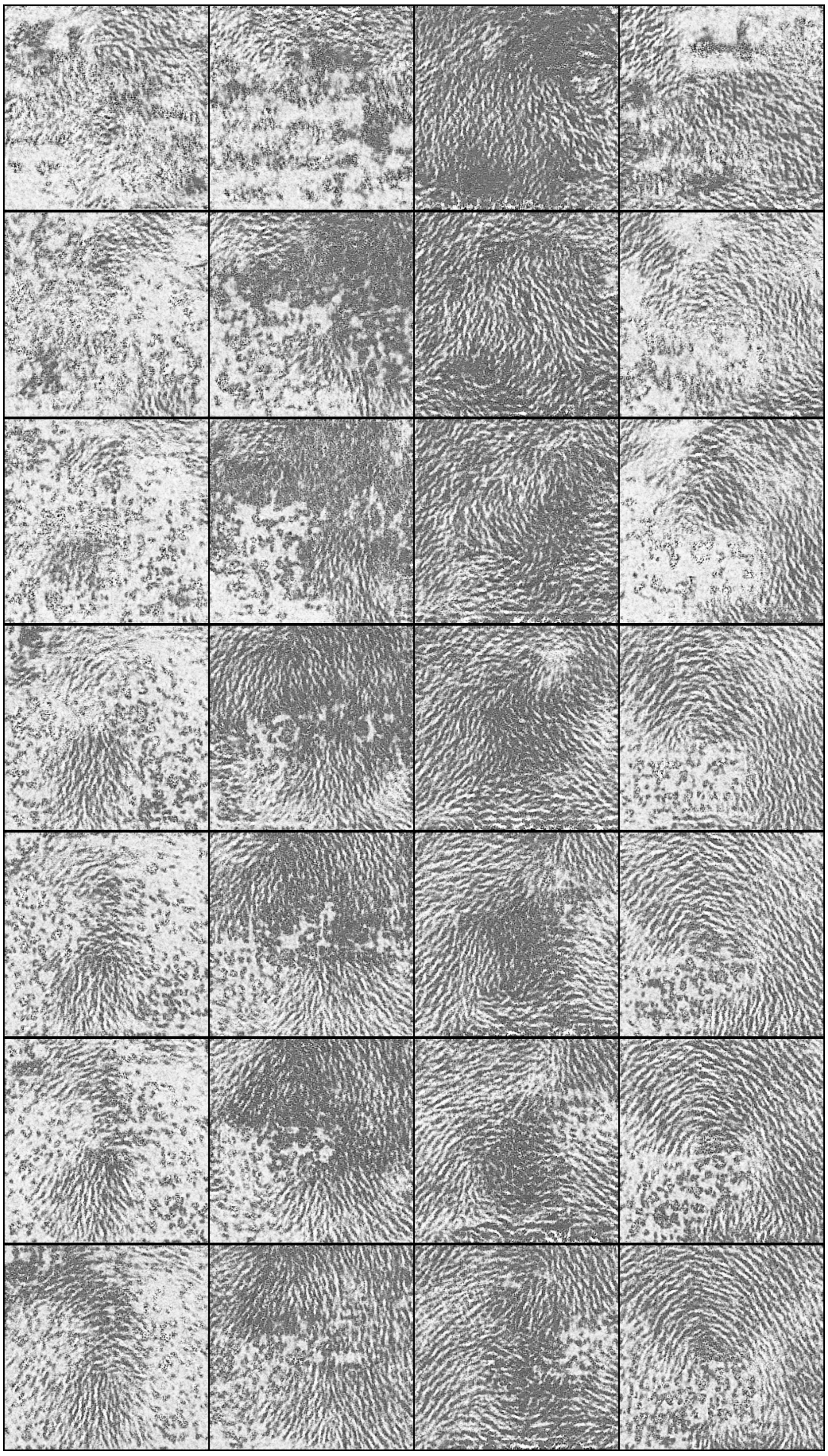


Figure 8.4: Improved convergence of the WGAN model during the training process when using training dataset of cropped fingerprint images influenced by atopic eczema. Images were sampled from training epoch 1000 up to epoch 4000 at intervals of 500.

8.1.3 Generated Diseased Fingerprint Datasets

The datasets of generated fingerprint images were sampled from the WGAN-GP models trained on their respective training datasets. The training was stopped once the model learned to generate fingerprint images of sufficient quality or the training deteriorated and the model did not seem to make any meaningful progress. The atopic eczema fingerprints were sampled after 13000 training epochs, the psoriasis fingerprints were sampled after 12500 training epochs and the dyshidrotic eczema fingerprints were sampled after 10000 training epochs.

The most successful results, in terms of creating a realistic looking fingerprint image, were achieved by the model trained on atopic eczema fingerprints. The WGAN model was also moderately successful in generating dyshidrotic eczema fingerprints. However, the model trained on psoriasis fingerprints was unable to replicate fingerprints damage caused by psoriasis and did not manage to achieve any meaningful results in this regard.

The fingerprints for the generated datasets were chosen manually from a larger sample size of created images. This was due to the fact that even after this many training iterations, the model would occasionally generate unnatural looking fingerprints. Some of the generated samples had poorly formed papillary line structure, see Figure 8.5. The direction of papillary lines around areas of heavy damage (dark places, large white spots) changes sporadically and runs in a number of different diretions from the center of the fingerprint. Other unsuitably generated images had multiple fingerprint cores in the image next to each other warping the surrounding ridge lines. These types of fingerprint images were excluded from the finalized datasets.

The resulting datasets contained 256 $\times$ 256 single channel grayscale images of each simulated disease. Total number of 300 images were selected for atopic eczema, 250 for psoriasis and 200 for dyshidrotic eczema datasets. To see large sample size of the generated images refer to Appendices B, C and D.

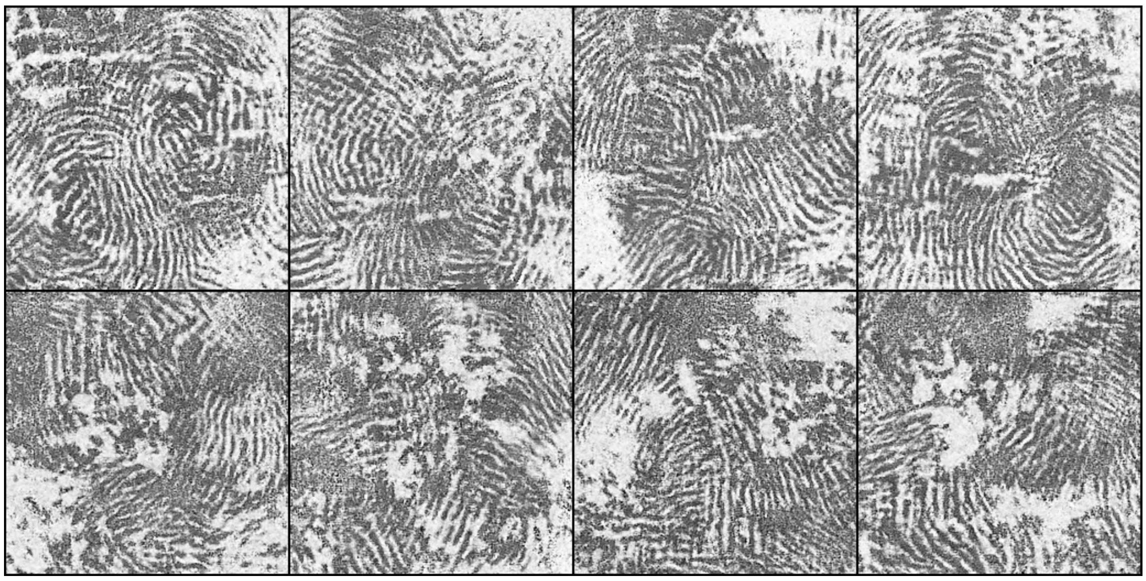


Figure 8.5: Examples of synthetic fingerprints with poorly formed papillary line structure. The direction of ridge lines in these images changes drastically and does not form a flow-like pattern. Some of the images even contain multiple cores.

Atopic Eczema

The training dataset for the atopic eczema was the most numerous from among the three chosen types of skin disease and contained some of the least damaged fingerprint images. The WGAN model trained on these images achieved the best overall results in terms of generating a consistent pattern of papillary lines in areas not damaged by the skin disease, see Figure 8.6. Majority of the generated images display at least moderate damage to the papillary line structure. Even though the model managed generate some fingerprint images that resembled atopic eczema with severe symptoms, it was not quite as successful at generating images with little to no symptoms.

The damage present in the synthetic images mirrors symptoms of the real disease in many ways. The images contain a lot of thin straight and jagged white lines crossing the papillary lines. Long straight lines were less common in the generated samples, but nevertheless still present. The images also exhibit a lot of dark places blurring the ridge lines and collections of small white spots which is consistent with atopic eczema. One thing that is not truly present in any of the images is an interlocking grid-like pattern of lines as the generated straight lines do not tend to be very long.

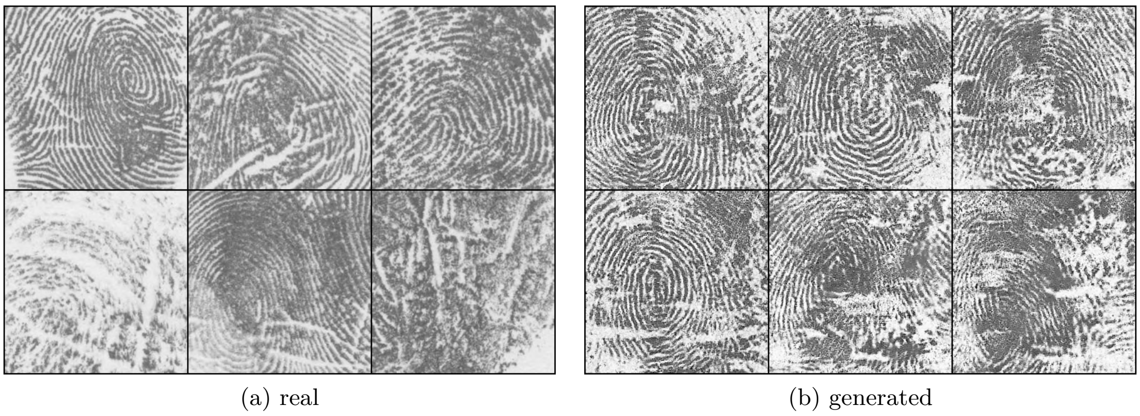


Figure 8.6: Samples of real and synthetically generated fingerprint images affected by atopic eczema.

Psoriasis

Generation of fingerprints influenced by psoriasis was the least successful. The training dataset consists of a wide range of heavily damaged fingerprints. Large percentage of these fingerprints show little signs of papillary line structure and present a wide range different types of damage. The model did not manage to create images that resemble real psoriasis fingerprints, most likely due to relatively small number of training samples and their high diversity with heavy ridge line disruptions.

None of the generated fingerprints had properly formed papillary line structure and all of them were almost completely damaged, see Figure 8.7. The most prominent observable features in the generated samples are extensive dark places and small white spots, which are symptoms of psoriasis. However, as the rest of the fingerprint image is not properly formed, the created samples are most likely not exceedingly useful. Other types of damage, e.g. irregular dark spots bounded by white border and straight white lines, are not present at all.

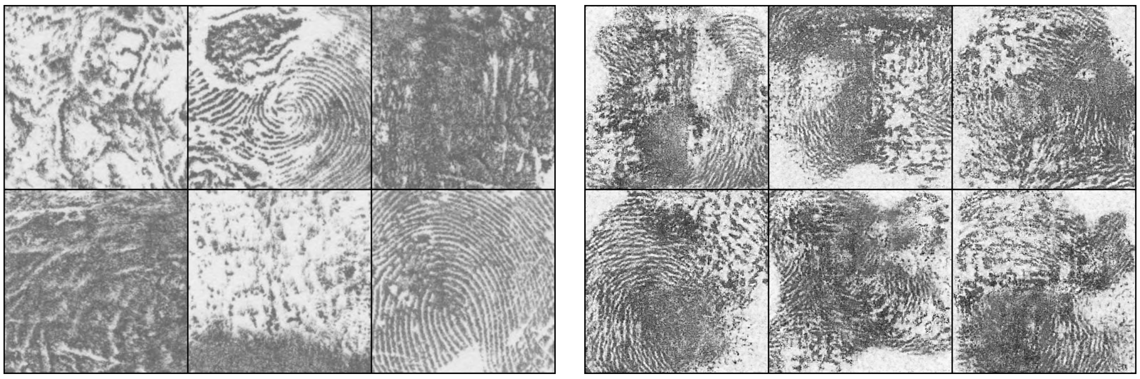

(a) real (b) generated

Figure 8.7: Samples of real and synthetically generated fingerprint images affected by psoriasis.

Dyshidrotic Eczema

Recreating images with symptoms of dyshidrotic eczema was met with moderate success. The training dataset for dyshidrosis was the smallest of the three and contained a mixture of heavily damaged and relatively undamaged fingerprints. The relatively undamaged samples had well-formed ridge lines with "missing" portions replaced by white areas and dark spots. These undamaged images helped the model learn recreate papillary lines, the model was close to forming a proper papillary line structure. As it stands, due to extensive damage in a lot of the training images, the generated images contain a lot of bifurcations, lakes, independent ridges, islands and crossovers, see Figure 8.8.

Almost all of the generated images are heavily damaged and the papillary line structure in between the damaged areas is of relatively poor quality. Nevertheless, the generated images do contain some of the symptoms of dyshidrotic eczema. The most prevalent symptoms are large irregular white spots and large dark places. Small irregular white spots are also present, but they are less clearly defined and tend to be grouped differently than in the real samples. Long white lines do not appear in the generated images. They are somewhat replaced by collections of small white areas forming jagged lines.

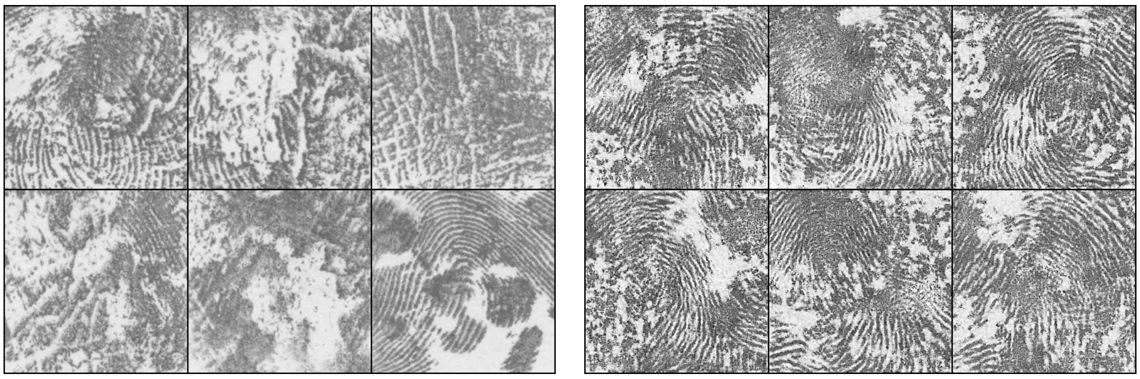

(a) real (b) generated

Figure 8.8: Samples of real and synthetically generated fingerprint images affected by dyshidrotic eczema.

8.2 NIST Finger Image Quality

NIST Finger Image Quality (NFIQ) [59] algorithm is one of the standard methods for assessing quality of fingerprint images. *National Institute of Standards and Technology* (NIST) released the algorithm in 2004 as a part of the *NIST Biometric Image Software* (NBIS). NFIQ is an open-source tool for quantifying the quality of fingerprint images, it is designed to be able to predict performance of minutiae-based fingerprint matching systems.

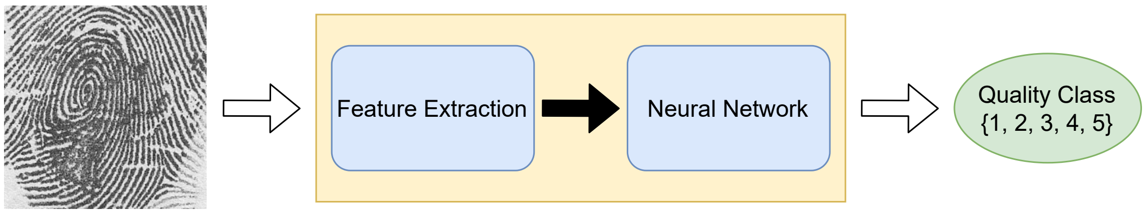


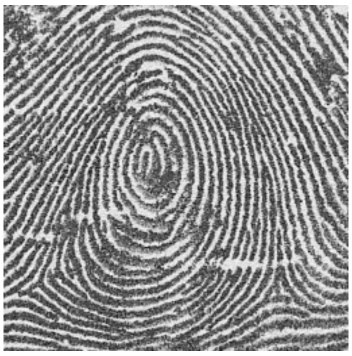
Figure 8.9: Predicting quality level of the fingerprint image using NFIQ.

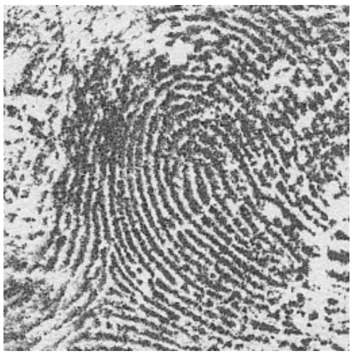
The NFIQ algorithm is based on an artificial neural network that assigns input fingerprint images to one of 5 classes (NFIQ levels of quality). Each respective class of the NFIQ algorithm represents specific quality of fingerprint images ranging from "excellent" quality images in class one to "poor" quality images in class five. For an example of fingerprint images and their respective NFIQ class see Figure 8.10.

The assignment to each of the classes is based on computing visual characteristics of the fingerprint image in the form of 11-dimensional feature vector, see Equation 21. Firstly, the minutia of the sample fingerprint are found using the MINDTCT program detailed in Section 8.3. The detection of minutia using MINDTCT can be somewhat unreliable, even though it implements procedures for mitigation of false positive detections. The NFIQ algorithm assigns a numeric value to each minutia depending on the likelihood of it being real bifurcation or line ending. Minutia that are more likely to be real are assigned a higher value. The algorithm subsequently counts the total number of detected minutia along with the number of minutia that have quality value higher than a certain threshold $q_m > t$, where $t \in \{0.5,\ 0.6,\ 0.7,\ 0.8,\ 0.9\}$. [59]

$$\vec{f} = \begin{pmatrix} foreground\ block\ count \\ total\ number\ of\ minutia \\ minutia\ quality\ 0.5\ count \\ minutia\ quality\ 0.6\ count \\ minutia\ quality\ 0.7\ count \\ minutia\ quality\ 0.8\ count \\ minutia\ quality\ 0.9\ count \\ quality\ 1\ block\ count \\ quality\ 2\ block\ count \\ quality\ 3\ block\ count \\ quality\ 4\ block\ count \end{pmatrix} \tag{21}$$

The NFIQ algorithm also splits the input image into blocks of equal size and computes their respective quality measure and assigns them one of 5 values. Every pixel that is part of a single block is assigned the same value. The first value 0 stands for background blocks of the image. These blocks are typically low-contrast parts of the image which don't contain any part of the papillary line structure. The other values 1−4 are assigned to blocks, of poor, fair, good, excellent quality respectively. Quality of these small local regions is determined using *direction map*, *low contrast map*, *low flow map* and *high curve map*. The direction map is computed using Discrete Fourier Transforms (DFT) and represents areas with sufficient ridge structure, with well formed and visible ridges and valleys. The low contrast map specifies lightly inked areas or areas containing smudges, this map is determined using pixel intensity. The low flow map marks areas that don't contain a dominant ridge flow. And the high curve map represents areas with high papillary line curvature. Detection of minutia is not exceedingly reliable in low flow and high curvature areas, meaning that these areas contribute negatively to the quality score of the local blocks. [59][59]


(a) Class 1 - excellent


(b) Class 3 - good

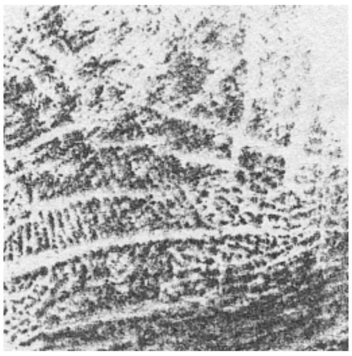
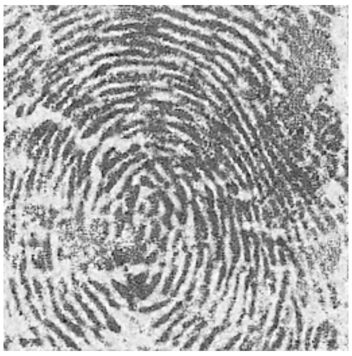
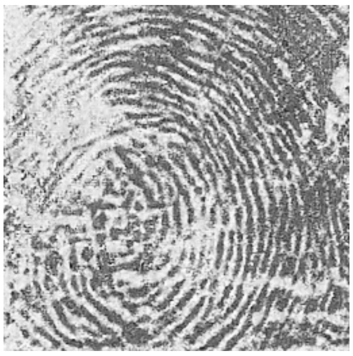

(c) Class 5 - poor

Figure 8.10: Comparison of sample fingerprint images from three different NFIQ classes from the real atopic eczema dataset.


(a) Class 1 - excellent


(b) Class 3 - good

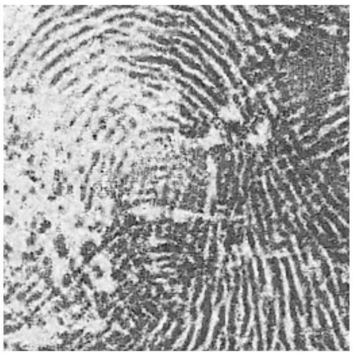
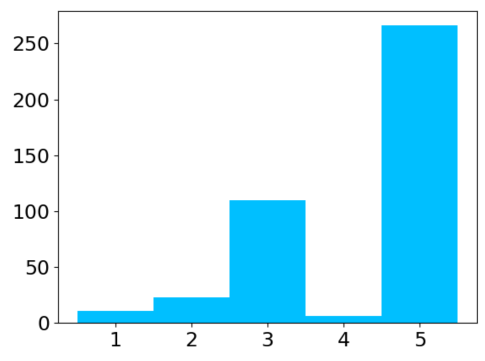
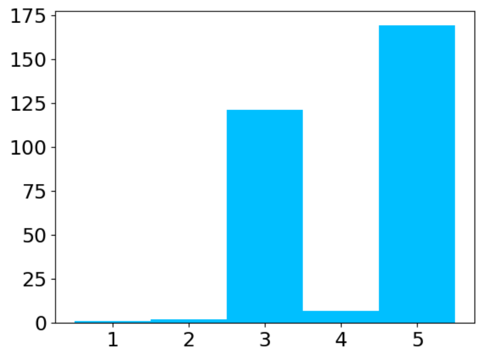

(c) Class 5 - poor

Figure 8.11: Comparison of sample fingerprint images from three different NFIQ classes from the generated atopic eczema dataset.

The aim of comparing real and generated images using the NFIQ assessment is to find out if the datasets have similar distributions of NFIQ classes. If the real dataset contains fingerprints with relatively minor damage caused by the disease as well as extensively damaged fingerprints, similarly damaged fingerprints should also be present in the generated dataset. The Figures 8.10 and 8.11 display fingerprint images from three different NFIQ classes as an example of how much damage to the papillary line structure is typically present in the respective NFIQ class.

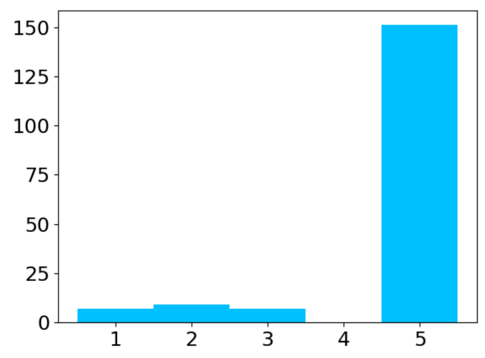


(a) real

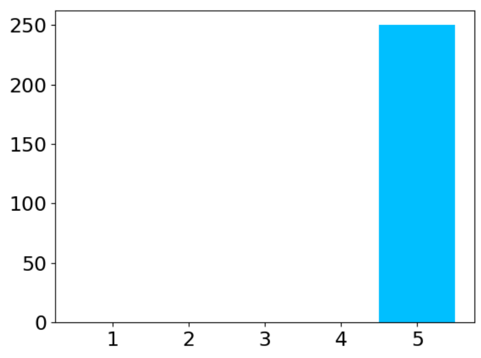


(b) generated

Figure 8.12: Histograms of NFIQ classes in the real and generated atopic eczema datasets.

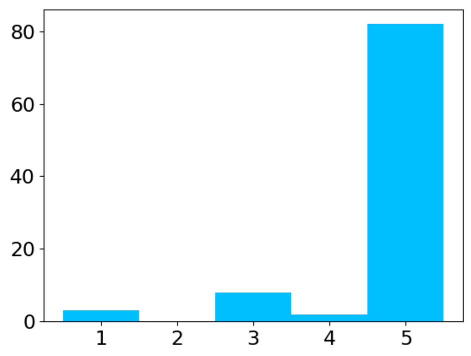


(a) real

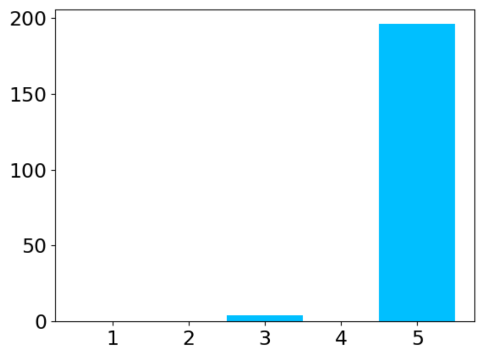


(b) generated

Figure 8.13: Histograms of NFIQ classes in the real and generated psoriasis datasets.



(a) real



(b) generated

Figure 8.14: Histograms of NFIQ classes in the real and generated dyshidrotic eczema datasets.

As previously stated in Subsection 8.1.3, the generated images typically show signs of moderate to extensive fingerprint damage. The Figures 8.12, 8.13 and 8.14 show that the majority of the images in training datasets are of “poor” quality and belong to NFIQ class 5. The models had difficulty successfully reproducing the minority of real fingerprint images that display little to no damage, as the generated images typically contained multiple different symptoms of the simulated skin disease. The only model that has learned to generate images of better quality was the one replicating atopic eczema, as this is the model that has best learned to reproduce coherent papillary line structure. However, even in this case the fingerprints of better quality were few and far between.

8.3 Minutia Count and Distribution Comparison

The synthetically generated fingerprints were compared with real diseased fingerprint images by comparing distributions of the number of minutiae and their orientations. The main idea was to find out whether these distributions closely mirror real datasets. [13]

The total counts of minutiae and their location and orientations were extracted from the sample fingerprint images using the minutia detector MINDTCT [60]. MINDTCT is an open-source software developed as a part of the NBIS. It is used to automatically detect minutiae of the fingerprint image along with their relative location, orientation, closest neighboring minutia and ridge count between detected minutia and it’s closest neighbor. The minutiae orientation is represented in degrees, with zero degrees pointing horizontally to the right. The orientation is assigned a value from $< 0, 31 >$, where each value represents a $11.25°$ increment from 0. Each of the detected minutia is also assigned a reliability score depending on the quality assessment of its surrounding region. The quality of the surrounding region is determined in the same way as in Section 8.2.

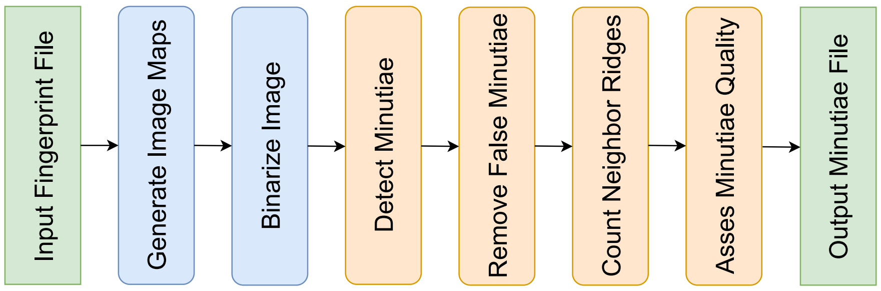


Figure 8.15: Process of minutiae detection in the MINDTCT software.

These metrics were only used to compare the atopic eczema and dyshidrotic eczema image, as these were the only generated samples that had relatively well-formed papillary line structure. Since the ridge lines in generated psoriaris images are just a collection of blurred, tightly packed crossovers, lakes etc., they were not considered in this instance.

The distribution of minutiae count in generated atopic eczema images is on average higher than in real samples, see Figure 8.16. This is to be somewhat expected as the proportion of NFIQ class 3 images is greater in the synthetic samples, see Figure 8.12. Nonetheless, this minutiae count distribution does not quite mirror the real distribution. It is more narrow and skewed towards higher counts, which most likely due to the fact that the model struggled to replicate the more extreme cases of fingerprint damage in the generated images.

The minutiae orientation distribution also differs from the real fingerprints. The orientation directions in the synthetic images are less diverse. A likely cause of this is that the model generated "good" quality images of whorl fingerprint class more frequently than other fingerprint classes. Therefore, they most likely represent larger than normal portion of the generated dataset.

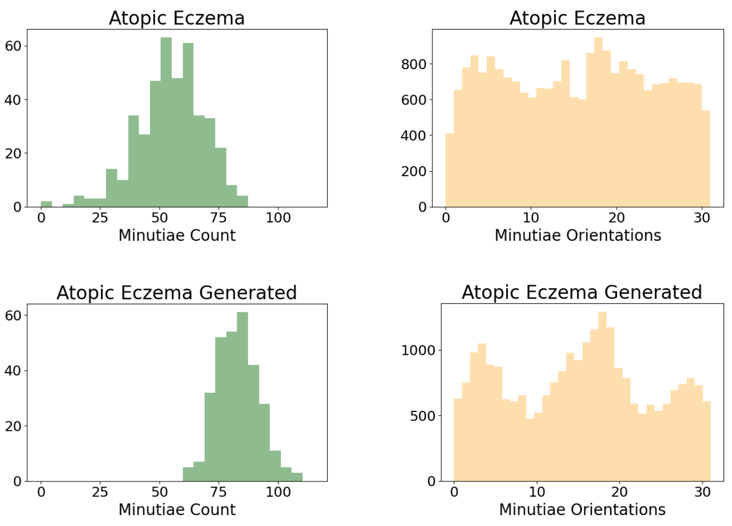


Figure 8.16: Distributions of minutiae count and orientation in atopic eczema datasets.

In the case of dyshidrotic eczema, the minutia counts are again often more numerous than in the real dataset. The papillary structure did not quite form a flow-like pattern of parallel running lines and contained a lot of ridge crossovers, which might account for why the minutia numbers are more numerous. Another reason might be that a proportion of the real dataset consists of images with total damage and indistinguishable papillary lines.

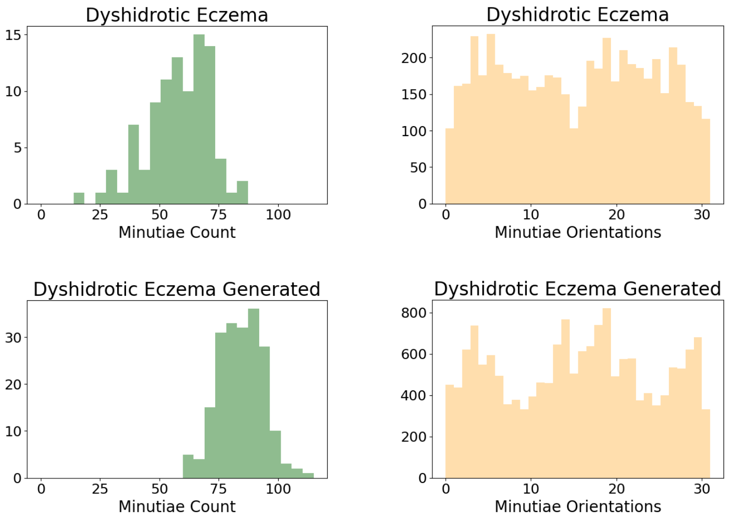


Figure 8.17: Distributions of minutiae count and orientation in dyshidrotic eczema datasets.

The generated samples did not manage to replicate this type of extensive damage, portions of the ridge lines are visible even in very heavily damaged synthetic samples. In terms of distribution of minutia orientations, the generated samples differ from the real ones. Interestingly, the distribution of minutia orientations somewhat mirrors the distribution of synthetic atopic eczema fingerprint images, even though the generated samples do not look similar to synthetic dyshidrotic eczema fingerprints.

8.4 Fingerprint Quality Visualizer

Fingerprint Quality Visualizer (FiQiVi) [11][47] is a fingerprint quality estimation software developed at FIT BUT. This software was developed in order to improve upon the NFIQ quality assessment algorithm and its weaknesses. The deficiencies of NFIQ being that the visual quality within a single quality class can be highly varied and its lack of proper techniques for mitigating influence of fingerprint damage on quality assesment. One of the aims of the FiQiVi tool was to be able to more accurately evaluate quality of diseased fingerprint images.

The FiViQi quality measurement takes into account the local and global properties of the sample image. It outputs a value in range of $< 0, 100 >$ with 0 signifying the worst possible fingerprint quality and 100 the best possible fingerprint quality. The algorithms splits the input image into smaller blocks of size 12×12 and larger blocks of size 28×28. The larger blocks are centered around the smaller blocks and 8 of its bits overlap into neighboring blocks. The algorithm computes quality measure for the large block and assigns this value to all pixels in the smaller 12×12 block. [47]

The blocks from the region of interest of the fingerprint image are first separated from the background depending on the contrast of grayscale values within the block. Low contrast regions are much more likely to be background. The quality of a particular block Q_B is computed as the minimum of six quality characteristics

$$Q_B = min(Q_o, Q_r, Q_{cb}, Q_{cn}, Q_c, \overline{Q_a}). \tag{22}$$

where Q_o is orientation certainty, Q_r is ridge to valley ratio, Q_{cb} is structural continuity within a block, Q_{cn} is orientation continuity with regard to neighboring blocks, Q_c is level of contrast and $\overline{Q_a}$ is the average of the aforementioned quality values. The overall quality of the fingerprint image Q_F is the average quality value of the foreground blocks of the fingerprint.

The FiQiVi quality was computed for all of the real and synthetically generated fingerprints, see Figure 8.18. It is noticeable that quality values for the generated images are a lot less diverse than their real counterparts. The interquartile range is a lot smaller for each of the generated datasets. This further proves that the trained WGAN models lacked the ability to generate very diverse fingerprints in terms of fingerprint damage variety. A lot of the successfully created synthetic samples could be considered as “moderately” damaged. They typically present with a lot of the diseas’ symptoms all at once, obscuring a lot of the fingerprint but leaving papillary lines in other parts still readable.

The median quality value in all of the synthetic datasets is higher than in the respective real datasets. The cause of this is decidedly the fact that the training datasets consisted of some of the less damaged samples in order to help models recreate a well-formed papillary line structure in undamaged parts of the generated images. What is very surprising is that even the unsuccessfully generated psoriasis fingerprints ranked higher than extensively damaged real psoriasis images. It could be that the quality assessment algorithm assigned higher quality value to the malformed ridge lines that appear in parts of the synthetic images.

The generated atopic eczema samples were not only the most successful in recreating proper ridge line structure, but they also had the highest diversity in terms of extent of fingerprint damage. The reason for this was very likely that the atopic eczema model was trained on the largest dataset with a lot of fingerprint images of relatively good quality.

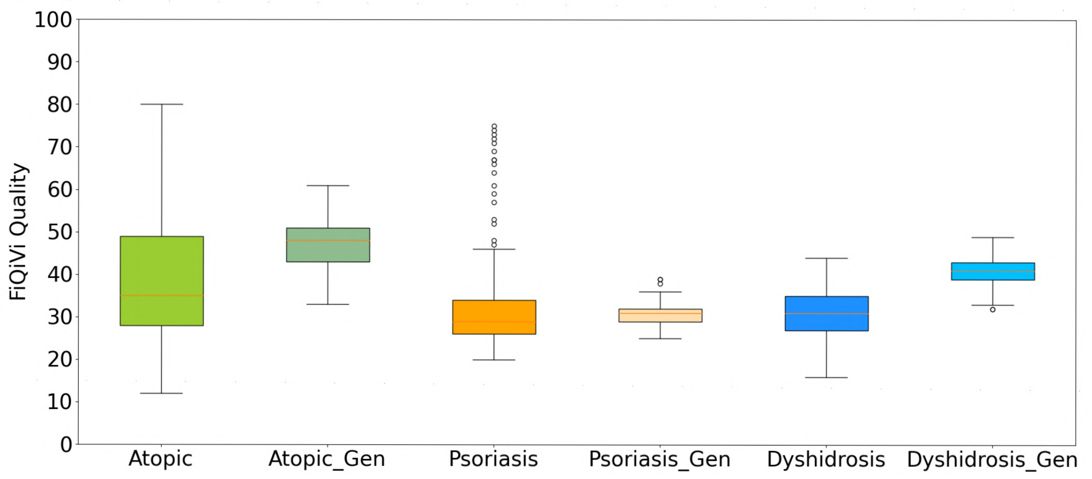


Figure 8.18: Fingerprint Quality Visualizer quality values for real and generated diseased fingerprint datasets.

Chapter 9

Conclusion

The main goal of this master's thesis was to generate datasets of synthetic fingerprint images influenced by skin disease, where the generated fingerprints would display symptoms analogous to the symptoms of the simulated disease. The fingerprint generation was done using a machine learning model based on Wasserstein GAN with gradient penalty. This GAN model was trained separately on three different datasets from the diseased fingerprint database: *atopic eczema*, *psoriasis* and *dyshidrotic eczema*. The generator network of the trained model was subsequently used to create three synthetic fingerprint datasets, one for each of the simulated disease.

The generated fingerprint images are usually at least moderately damaged. They tend to present with multitude of the skin disease symptoms all at once and less damaged fingerprint images are created only rarely. This is most likely caused by the fact that the real datasets contain only a small amount of relatively undamaged fingerprint samples. The generation of fingerprints influenced by atopic eczema was the most successful as the generated images had well-formed papillary line structure disrupted by damage typical to atopic eczema (e.g. dark places, straight white lines). Simulating effects of dyshidrotic eczema was met with moderate success. The papillary line structure in the generated images was of poor quality, but the images showed some symptoms of the skin disease. The WGAN model was unable to replicate psoriasis images. This failure to generate psoriasis images can most likely be attributed to the extent and highly diverse nature of the damage caused by the disease. Overall, the models had issues with generating fingerprint images at extreme ends of the spectrum – almost undamaged or completely damaged with unrecognizable ridge line structure.

Even though the trained WGAN models were partially successful in generating synthetic fingerprints influenced by skin disease, the overall diversity of the generated datasets was smaller than that of the real diseased datasets. This is evidenced by computation of NFIQ and FiQiVi quality assessment metrics as well as comparison of minutiae characteristics for each pair of diseased datasets. None of the trained WGAN models were able to recreate samples that spanned quite the same range as the real data.

Nevertheless, this thesis proved that generating a quality, realistic-looking dataset of diseased fingerprint images might be possible with further improvements. The GAN models were promising even though training them to a point of convergence proved to be a difficult task with such small, highly varied datasets. The most successful model was the one trained on the largest, most diverse dataset. Any increase in terms of the number of training samples would very likely further improve performance of the models. This increase in training data could come either from gathering further samples of diseased fingerprints or employing more sophisticated methods of data augmentation.

For future work, a GAN training technique that might warrant further exploration is transfer learning. Transfer learning was already employed to a small extent during the pretraining of the model on less damaged samples. It is a machine learning technique where the model is tuned to adapt from a source domain to a target domain. In this case, it would be possible to train a model on entirely undamaged fingerprints until it learns to recreate proper structure of the fingerprints. Then take this model and then train it on very similar-looking damaged fingerprint images acquired in the same manner until the model learns to introduce elements of skin disease. However, one downside of this approach would be that the final model would most likely not be able to produce heavily damaged samples.

Bibliography

[1] Aggarwal, C. C. et al. *Neural networks and deep learning.* Springer, 2018.

[2] Arjovsky, M., Chintala, S. and Bottou, L. Wasserstein gan. *ArXiv preprint arXiv:1701.07875.* 2017.

[3] Barotová, Š. Detector of Skin Diseases by Fingerprint Technology. *Bachelor's thesis FIT BUT.* 2017, p. 50.

[4] Bisong, E. Google colaboratory. In: *Building Machine Learning and Deep Learning Models on Google Cloud Platform.* Springer, 2019, p. 59–64.

[5] Bontrager, P., Roy, A., Togelius, J., Memon, N. and Ross, A. Deepmasterprints: Generating masterprints for dictionary attacks via latent variable evolution. In: IEEE. *2018 IEEE 9th International Conference on Biometrics Theory, Applications and Systems (BTAS).* 2018, p. 1–9.

[6] Brock, A., Donahue, J. and Simonyan, K. Large scale gan training for high fidelity natural image synthesis. *ArXiv preprint arXiv:1809.11096.* 2018.

[7] Cao, K. and Jain, A. Fingerprint synthesis: Evaluating fingerprint search at scale. In: IEEE. *2018 International Conference on Biometrics (ICB).* 2018, p. 31–38.

[8] Cappelli, R., Erol, A., Maio, D. and Maltoni, D. Synthetic fingerprint-image generation. In: IEEE. *Proceedings 15th International Conference on Pattern Recognition. ICPR-2000.* 2000, vol. 3, p. 471–474.

[9] Cappelli, R., Maio, D. and Maltoni, D. SFinGe: an approach to synthetic fingerprint generation. In: *International Workshop on Biometric Technologies (BT2004).* 2004, p. 147–154.

[10] Creswell, A., White, T., Dumoulin, V., Arulkumaran, K., Sengupta, B. et al. Generative adversarial networks: An overview. *IEEE Signal Processing Magazine.* IEEE. 2018, vol. 35, no. 1, p. 53–65.

[11] Dejmal, D. Analysis of Fingerprint Quality Assessment Systems. *Bachelors's thesis FIT BUT.* 2019, p. 47.

[12] Dolezel, M., Drahansky, M., Urbanek, J., Brezinova, E. and Kim, T.-h. Influence of skin diseases on fingerprint quality and recognition. *New Trends and Developments in Biometrics.* InTech. 2012, p. 275–298.

[13] Drahanskỳ, M. *Hand-based Biometrics: Methods and Technology.* Institution of Engineering and Technology, 2018.

[14] Drahansky, M., Brezinova, E., Hejtmankova, D. and Orsag, F. Fingerprint recognition influenced by skin diseases. *International Journal of Bio-Science and Bio-Technology*. 2010, vol. 2, no. 4.

[15] Drahanskỳ, M. and Kanich, O. Influence of skin diseases on fingerprints. In: *Biometrics under Biomedical Considerations*. Springer, 2019, p. 1–39.

[16] Dumoulin, V., Shlens, J. and Kudlur, M. A learned representation for artistic style. *ArXiv preprint arXiv:1610.07629*. 2016.

[17] Dumoulin, V. and Visin, F. A guide to convolution arithmetic for deep learning. *ArXiv preprint arXiv:1603.07285*. 2016.

[18] Durall, R., Chatzimichailidis, A., Labus, P. and Keuper, J. Combating Mode Collapse in GAN training: An Empirical Analysis using Hessian Eigenvalues. *ArXiv preprint arXiv:2012.09673*. 2020.

[19] Eaton Rosen, Z., Bragman, F., Ourselin, S. and Cardoso, M. J. Improving data augmentation for medical image segmentation. 2018.

[20] Fahim, M. A.-N. I. and Jung, H. Y. A Lightweight GAN Network for Large Scale Fingerprint Generation. *IEEE Access*. IEEE. 2020, vol. 8, p. 92918–92928.

[21] Feng, J. and Jain, A. K. Fingerprint reconstruction: from minutiae to phase. *IEEE transactions on pattern analysis and machine intelligence*. IEEE. 2010, vol. 33, no. 2, p. 209–223.

[22] Galton, F. *Finger prints*. Macmillan and Company, 1892.

[23] Golub, G. H. and Vorst, H. A. Van der. Eigenvalue computation in the 20th century. *Journal of Computational and Applied Mathematics*. Elsevier. 2000, vol. 123, 1-2, p. 35–65.

[24] Goodfellow, I. NIPS 2016 tutorial: Generative adversarial networks. *ArXiv preprint arXiv:1701.00160*. 2016.

[25] Goodfellow, I., Bengio, Y., Courville, A. and Bengio, Y. *Deep learning*. MIT press Cambridge, 2016.

[26] Goodfellow, I., Pouget Abadie, J., Mirza, M., Xu, B., Warde Farley, D. et al. Generative adversarial nets. *Advances in neural information processing systems*. 2014, vol. 27, p. 2672–2680.

[27] Gulrajani, I., Ahmed, F., Arjovsky, M., Dumoulin, V. and Courville, A. C. Improved training of wasserstein gans. In: *Advances in neural information processing systems*. 2017, p. 5767–5777.

[28] Habif, T. P. *Clinical Dermatology E-Book*. Elsevier Health Sciences, 2015.

[29] Ioffe, S. and Szegedy, C. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *ArXiv preprint arXiv:1502.03167*. 2015.

[30] Jain, A., Bolle, R. and Pankanti, S. Introduction to biometrics. In: *Biometrics*. Springer, 1996, p. 1–41.

[31] Jain, A. K., Flynn, P. and Ross, A. A. *Handbook of biometrics.* Springer Science & Business Media, 2007.

[32] James, W. D., Elston, D. and Berger, T. *Andrew's Diseases of the Skin E-Book.* Elsevier Health Sciences, 2011.

[33] Jupyter Team. *The Jupyter Notebook documentation.* 2021. Accessed: 2021-04-08. Available at: https://jupyter-notebook.readthedocs.io/en/stable/notebook.html.

[34] Khan, S., Rahmani, H., Shah, S. A. A. and Bennamoun, M. A guide to convolutional neural networks for computer vision. *Synthesis Lectures on Computer Vision.* Morgan & Claypool Publishers. 2018, vol. 8, no. 1, p. 1–207.

[35] Kingma, D. P. and Ba, J. Adam: A method for stochastic optimization. *ArXiv preprint arXiv:1412.6980.* 2014.

[36] Kodali, N., Abernethy, J., Hays, J. and Kira, Z. On convergence and stability of gans. *ArXiv preprint arXiv:1705.07215.* 2017.

[37] Kolarsick, P. A., Kolarsick, M. A. and Goodwin, C. Anatomy and physiology of the skin. *Journal of the Dermatology Nurses' Association.* LWW. 2011, vol. 3, no. 4, p. 203–213.

[38] Langr, J. and Bok, V. *GANs in Action: Deep Learning with Generative Adversarial Networks.* Manning Publications, 2019.

[39] LeCun, Y., Bottou, L., Bengio, Y. and Haffner, P. Gradient-based learning applied to document recognition. *Proceedings of the IEEE.* Ieee. 1998, vol. 86, no. 11, p. 2278–2324.

[40] Ledig, C., Theis, L., Huszár, F., Caballero, J., Cunningham, A. et al. Photo-realistic single image super-resolution using a generative adversarial network. In: *Proceedings of the IEEE conference on computer vision and pattern recognition.* 2017, p. 4681–4690.

[41] Maltoni, D., Maio, D., Jain, A. K. and Prabhakar, S. *Handbook of fingerprint recognition.* Springer Science & Business Media, 2009.

[42] Malvi. *The Ageing Skin – Structure of Skin.* 2011. Accessed: 2021-01-06. Available at: http://web.archive.org/web/20201124180936/https://pharmaxchange.info/2011/03/the-ageing-skin-part-1-structure-of-skin-and-introduction/.

[43] Minaee, S. and Abdolrashidi, A. Finger-GAN: Generating realistic fingerprint images using connectivity imposed GAN. *ArXiv preprint arXiv:1812.10482.* 2018.

[44] Miyato, T., Kataoka, T., Koyama, M. and Yoshida, Y. Spectral normalization for generative adversarial networks. *ArXiv preprint arXiv:1802.05957.* 2018.

[45] Nielsen, M. A. *Neural networks and deep learning.* Determination press San Francisco, CA, 2015.

[46] Odena, A., Dumoulin, V. and Olah, C. Deconvolution and checkerboard artifacts. *Distill.* 2016, vol. 1, no. 10, p. e3.

[47] Oravec, T. Methodology of Fingerprint Image Quality Measurement. *Master's thesis FIT BUT*. 2018, p. 45.

[48] Osgood, B. *The Fourier Transform and its Applications: Stanford University*. 2007. 428 p.

[49] Pal, A. and Das, A. TorchGAN: A Flexible Framework for GAN Training and Evaluation. *ArXiv preprint arXiv:1909.03410*. 2019.

[50] Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E. et al. Automatic differentiation in pytorch. 2017.

[51] Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J. et al. Pytorch: An imperative style, high-performance deep learning library. *ArXiv preprint arXiv:1912.01703*. 2019.

[52] Perez, F. and Granger, B. E. Project Jupyter: Computational narratives as the engine of collaborative data science. *Retrieved September* [https://jupyter.org/]. 2015, vol. 11, no. 207, p. 108.

[53] Radford, A., Metz, L. and Chintala, S. Unsupervised representation learning with deep convolutional generative adversarial networks. *ArXiv preprint arXiv:1511.06434*. 2015.

[54] Salimans, T., Goodfellow, I., Zaremba, W., Cheung, V., Radford, A. et al. Improved techniques for training gans. *ArXiv preprint arXiv:1606.03498*. 2016.

[55] Shorten, C. and Khoshgoftaar, T. M. A survey on image data augmentation for deep learning. *Journal of Big Data*. Springer. 2019, vol. 6, no. 1, p. 60.

[56] Simonyan, K. and Zisserman, A. Very deep convolutional networks for large-scale image recognition. *ArXiv preprint arXiv:1409.1556*. 2014.

[57] Springenberg, J. T., Dosovitskiy, A., Brox, T. and Riedmiller, M. Striving for simplicity: The all convolutional net. *ArXiv preprint arXiv:1412.6806*. 2014.

[58] Strang, G. A proposal for Toeplitz matrix calculations. *Studies in Applied Mathematics*. Wiley Online Library. 1986, vol. 74, no. 2, p. 171–176.

[59] Tabassi, E., Wilson, C. and Watson, C. I. Fingerprint Image Qualitiy. 2004.

[60] Watson, C. I., Garris, M. D., Tabassi, E., Wilson, C. L., McCabe, R. M. et al. User's Guide to NBIS. *NIST Biometric Image Software*.

[61] Weng, L. From GAN to WGAN. *ArXiv preprint arXiv:1904.08994*. 2019.

[62] Zhang, H., Goodfellow, I., Metaxas, D. and Odena, A. Self-attention generative adversarial networks. In: PMLR. *International conference on machine learning*. 2019, p. 7354–7363.

[63] Zhao, Q., Jain, A. K., Paulter, N. G. and Taylor, M. Fingerprint image synthesis based on statistical feature models. In: IEEE. *2012 IEEE Fifth International Conference on Biometrics: Theory, Applications and Systems (BTAS)*. 2012, p. 23–30.

[64] *Papers With Code Trends.* Accessed: 2021-04-10. Available at: https://paperswithcode.com/trends.

[65] Hong, L. and Jain, A. Classification of fingerprint images. In: *Proceedings of the scandinavian conference on image analysis.* 1999, vol. 2, p. 665–672.

[66] Dumoulin, V. and Visin, F. A guide to convolution arithmetic for deep learning. *ArXiv.* 2016, abs/1603.07285.

[67] Belleza, M. *Integumentary System Anatomy and Physiology.* 2017. Accessed: 2021-01-06. Available at: https://nurseslabs.com/integumentary-system/.

[68] Veličković, P., Cucurull, G., Casanova, A., Romero, A., Liò, P. et al. *Graph Attention Networks.* 2018. Visited. Available at: https://openreview.net/forum?id=rJXMpikCZ.

[69] Hassan, M. U. *VGG16–Convolutional network for classification and detection.* 2018.

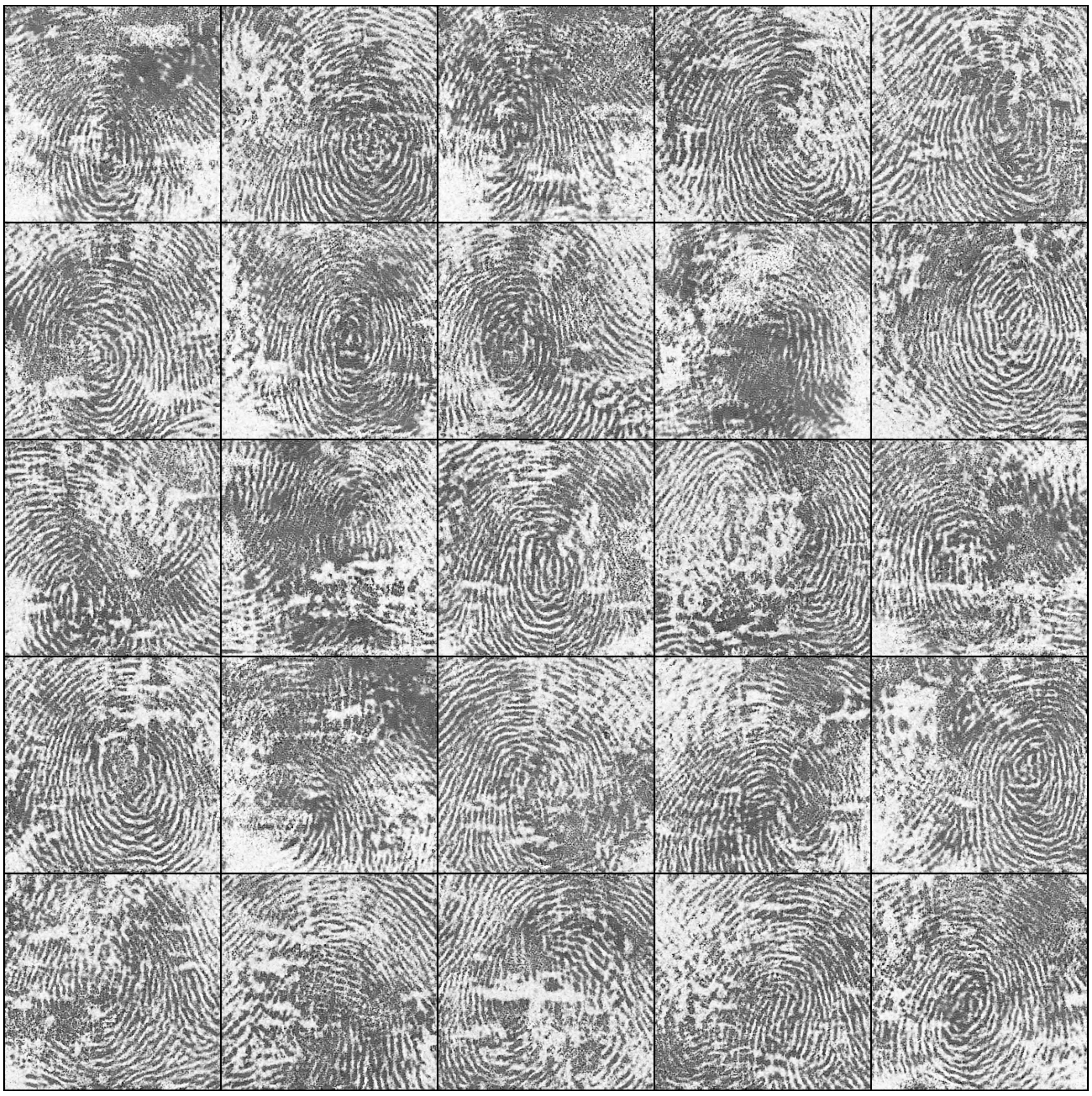
Appendix A

Contents of the Included Storage Media

- Electronic version of this master's thesis in PDF format.
- LaTeX source code for this document.
- Datasets of synthetically generated fingerprint images.
- Pretrained WGAN-GP models usable for synthetic fingerprint generation
- Source code of the WGAN-GP
- Source code and executables for fingerprint image quality assessment

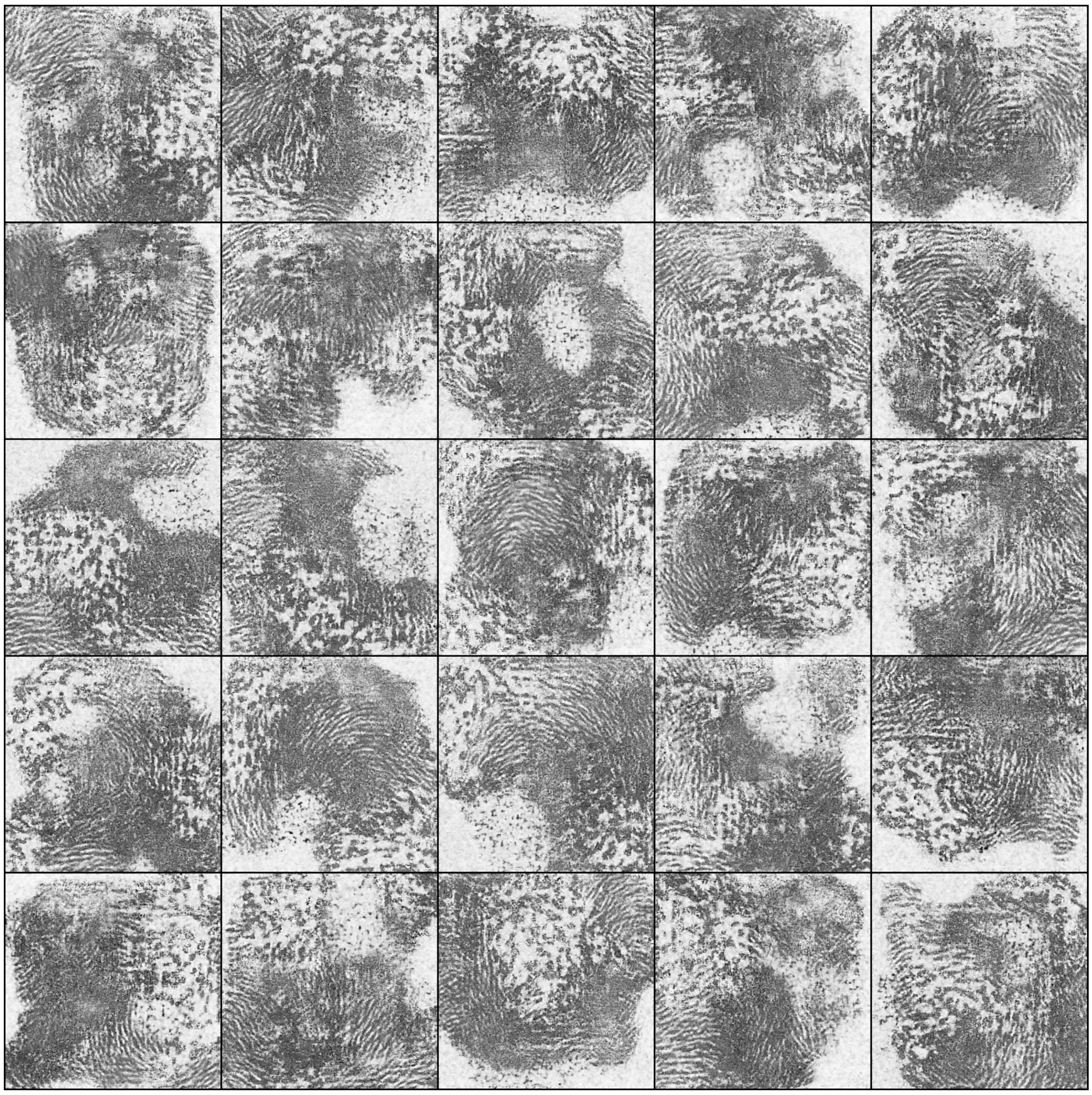
Appendix B

Sample of Generated Fingerprint Images - Atopic Eczema



Appendix C

Sample of Generated Fingerprint Images - Psoriasis



Appendix D

Sample of Generated Fingerprint Images - Dyshidrotic Eczema