BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FINGERPRINT IDENTITY PRESERVING GENERATIVE ADVERSARIAL NETWORKS

GENERATIVNÍ OPONENTNÍ NEURONOVÉ SÍTĚ ZACHOVÁVAJÍCÍ IDENTITU OTISKU PRSTU

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR **Bc. JÁN KAČUR**
AUTOR PRÁCE

SUPERVISOR **Ing. JAKUB ŠPAŇHEL**
VEDOUCÍ PRÁCE

BRNO 2023

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Master's Thesis Assignment

144751

Institut:	Department of Computer Graphics and Multimedia (UPGM)
Student:	**Kačur Ján, Bc.**
Programme:	Information Technology and Artificial Intelligence
Specialization:	Machine Learning
Title:	**Fingerprint Identity Preserving Generative Adversarial Networks**
Category:	Image Processing
Academic year:	2022/23

Assignment:

1. Study the basics of image processing. In particular, focus on neural networks.
2. Study the available materials on generative opponent neural networks and conditional generative opponent neural networks.
3. Explore current methods for generating identity-preserving training images using generative opponent neural networks.
4. Select an appropriate method and design a system for generating fingerprint images that preserve the identity of the original fingerprint.
5. Experiment with your implementation and propose your own modifications to the methods if necessary.
6. Compare your results and discuss possibilities for future developments.
7. Create a brief poster and video presenting your work, its goals, and results.

Literature:

- MIRZA, Mehdi; OSINDERO, Simon. Conditional generative adversarial nets. *arXiv preprint arXiv:1411.1784*, 2014.
- Dále dle pokynů vedoucího

Requirements for the semestral defence:

- Completion of the first three points of the assignment
- Considerable work on the fourth point of the assignment

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

Supervisor:	**Špaňhel Jakub, Ing.**
Consultant:	Sochor Jakub, Ph.D. Ing.
Head of Department:	Černocký Jan, prof. Dr. Ing.
Beginning of work:	1.11.2022
Submission deadline:	17.5.2023
Approval date:	31.10.2022

Abstract

This thesis focuses on generating latent fingerprints using Generative adversarial networks. The main objective is to generate multiple latent fingerprints from the clean fingerprint, with the same identity. The identity and the style should also be controllable separately. The chosen approach is based on AugNet model. Designed algorithm generates latent fingerprints from clean binarized fingerprint, and a random vector encoding distortions, i.e style. In the generator, AdaIN blocks are used to incorporate distortions into the input fingerprint. Various training algorithms are tested, with WGAN-GP performing the best. Individual models are compared using a combination of FID, and Rank-1 accuracy on matching generated images to original input binarized fingerprints. Best performing models are selected as a Pareto optimal combinations of these 2 metrics.

Abstrakt

Táto práca sa sústredí na generovanie latentných odtlačkov prstov za pomoci Generatívnych oponentných neurónových sietí. Hlavnou úlohou je generovanie viacerých verzií latentných odtlačkov z čistého odtlačku, s rovnakou identitou. Identitu a štýl odtlačku by malo byť možné osobitne meniť. Zvolený postup sa zakladá na modeli AugNet. Navrhnutý algoritmus generuje latentné odtlačky z čistých binarizovaných odtlačkov a náhodného vektora, reprezentujúceho skreslenie, resp. štýl. V generátore sú použité AdaIN bloky na spojenie štýlu so vstupným odtlačkom. Je testovaných viacero trénovacích algoritmov, z ktorých WGAN-GP dosahuje najlepšie výsledky. Jednotlivé modely sú porovnávané kombináciou metrík FID a Rank-1 accuracy pri porovnávaní generovaných obrázkov s originálnymi vstupnými binarizovanými odtlačkami. Najlepšie modely sú vybrané ako Pareto optimálne kombinácie týchto 2 metrík.

Keywords

fingerprint generation, latent fingerprint, GAN, conditional GAN, AugNet, MOLF, NIST SD302, WGAN-GP

Klíčová slova

generovanie odtlačkov prsta, latentný odtlačok prsta, GAN, conditional GAN, AugNet, MOLF, NIST SD302, WGAN-GP

Reference

KAČUR, Ján. *Fingerprint Identity Preserving Generative Adversarial Networks.* Brno, 2023. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Jakub Špaňhel

Rozšířený abstrakt

Identifikácia odtlačkov prstov je jednou z najdôležitejších oblastí biometrie. So stúpajúcim využívaním metód strojového učenia v rôznych vedeckých a priemyselných oblastiach nachádzajú tieto metódy veľké uplatnenie aj pri identifikácii odtlačkov prstov. Jednou z dominantných metód strojového učenia sú v súčasnosti neurónové siete. Neurónové siete často potrebujú kvalitné datasety veľkých rozmerov na trénovanie, aby dosiahli optimálne výsledky. To samozrejme platí aj v pri učení rôznych algoritmov z oblasti odtlačkov prstov. Problémom je však nízka dostupnosť verejných datasetov odtlačkov prstov, najmä kvôli ochrane súkromia.

Latentné odtlačky prstov sú typom odtlačkov, ktoré sú získané z nejakého povrchu potom, čo tam boli zanechané nejakou osobou. Narozdiel od čistých odtlačkov prstov, ktoré často pochádzajú z optických senzorov a disponujú vysokou kvalitou, sú tieto odtlačky v často v horšej kvalite, a obsahujú rôzne deformácie a poškodenia. Rozpoznávanie latentných odtlačkov patrí k najviac náročným, avšak najviac prínosným oblastiam biometrie. Na tieto účely, ako bolo už spomenuté, sa často využívajú práve neurónové siete. Verejne dostupné datasety latentných odtlačkov sú však ešte vzácnejšie, čo spôsobuje, že využitie neurónových sietí v tejto oblasti je problematické.

Táto práca je preto venovaná generovaniu syntetických latentných odtlačkov za využitia Generatívnych oponentných neurónových sietí (GAN). Cieľom je navrhnúť algoritmus, ktorý je schopný generovať viacero latentných odtlačkov z čistého binarizovaného odtlačku s tým, že identita odtlačku ostane zachovaná.

Algoritmy navrhnuté v tejto práci sú založené na modeli AugNet. Tieto algoritmy dostanú na vstup binarizovaný čistý odtlačkok a vektor náhodných čísel, z ktorého vygenerujú realistický latentný odtlačok prsta. Daný vektor reprezentuje rôzne deformácie a textúry, ktoré sa vyskytujú v latentných odtlačkoch. Jedná sa teda o kombináciu identity odtlačku so štýlom, ktorý má byť na neho aplikovaný. Veľmi dôležitou požiadavkou na takýto algoritmus je zachovanie identity binarizovaného odtlačku, aby mohli byť dáta vytvorené týmto algoritmom využité napr. pri trénovaní neurónových sietí.

Pôvodným cieľom tejto práce bolo implementovať algoritmus modelu AugNet, a pokúsiť sa zistiť, či naozaj funguje tak, ako autori uvádzajú. To sa však ukázalo ako náročná úloha, pretože opis architektúry a algoritmov v pôvodnom článku obsahoval zopár nezrovnalostí, a vynechával rôzne kľúčové detaily. Napriek tomu, slúžil ako dobrý základ pre množstvo experimentov, ktoré boli vykonané.

Keďže prvotné experimenty s modelom AugNet nedopadli podľa očakávaní, zvyšok práce bol venovaný vývoju a testovaniu modelu s architektúrou conditional GAN. Jednalo sa o zjednodušenú verziu modelu AugNet. Generátor a diskriminátor boli implementované podľa veľmi stručného popisu architektúry v danom článku. V článku nebol uvedený spôsob, akým sa má vektor náhodných čísel transformovať na štýl latentného odtlačku. Pre túto úlohu som zvolil AdaIN blok, použitý napr. v modeli StyleGAN. Eventuálne sa podarilo nájsť konfiguráciu, ktorá dokázala generovať hodnoverné odtlačky. Problémom bol tzv. mode collapse, kedy generátor generuje len zopár vizuálne odlišných štýlov latentných odtlačkov. Väčšina práce sa zaoberá riešením tohoto problému.

Počas tejto práce boli vykonané desiatky experimentov, ktorých cieľom bolo nájsť správnu architektúru a trénovací algoritmus. Väčšina z nich bola neúspešná. Eventuálne sa však podarilo nájsť vhodné riešenie. Hlavnými zlepšeniami boli trénovací algoritmus WGAN-GP a drobná, ale podstatná zmena v architektúre generátora. Tieto 2 zmeny výrazne zlepšili výsledky modelu, a pomohli odstrániť spomínaný mode collapse problém.

Porovnať výsledky modelu v tejto práci s ostatnými súčasnými metódami bolo náročné, keďže metódy ich vyhodnotenia boli ťažko replikovateľné. Preto bola navrhnutá kombinácia 2 metrík. Prvou z nich je Fréchet inception distance (FID), ktorá aproximuje vzdialenosť pravdepodobnostných rozložení skutočných a generovaných dát. Druhou je Rank-1 accuracy, ktorá určuje, aké percento datasetu sa podarilo spojiť s pôvodnými binarizovanými odtlačkami, z ktorých boli generované. Ohodnotenie spájania identít bolo vykonané tímom zo spoločnosti Innovatrics na programe IEngine.

Keďže boli na vyhodnotenie použité 2 rôzne metriky, nedá sa jednoznačne určiť, ktorá konfigurácia je najlepšia. Preto som pre 2 rôzne trénovacie datasety (NIST SD302 a MOLF DB4) vybral Pareto optimálne konfigurácie na základe kombinácie týchto 2 metrík. Pre každú z týchto konfigurácií bola potom vykreslená CMC krivka, od Rank-1 až po Rank-25 accuracy.

Fingerprint Identity Preserving Generative Adversarial Networks

Declaration

I hereby declare that this term project was prepared as an original work by the author under the supervision of Ing. Jakub Špaňhel. Supplementary information was provided by an external consultant, Ing. Jakub Sochor, Ph.D. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

.....
Ján Kačur
May 23, 2023

Acknowledgements

I would like to express my gratitude to my supervisor Ing. Jakub Špaňhel, for providing me with information necessary to accomplish this project, and helping me solving problems during consultations. I would also like to thank Ing. Jakub Sochor, Ph.D., for providing valuable data and algorithms, which were used in this thesis. Last but not least, thank you goes to all other members of Innovatrics company, who were involved in any of the processes related to this thesis.

Contents

List of Figures

Chapter 1

Introduction

Fingerprint identification has been one of the key components of biometrics for a long time. With increasing popularity of machine learning across various domains, it was inevitable, that machine learning methods would find its application in fingerprint identification. As neural networks currently dominate the machine learning industry, many fingerprint identification algorithms became centered around them. Neural networks often require large scale, high quality, curated datasets in order to learn required tasks. The same holds true for learning fingerprint related algorithms. However, as a large portion of previously publicly available fingerprint datasets has been redacted over the past years due to privacy concerns [12], it has shown to be increasingly difficult to train such models.

One of the most challenging areas of fingerprint identification is identifying latent [13] fingerprints. These are often low quality, damaged impressions of fingerprints, left at a surface by a subject. Public datasets of latent fingerprints are even more scarce, than datasets of clean fingerprints. Thus, training large scale neural networks for recognition of latent fingerprints is a very difficult task.

This thesis is focused on solving the issue of lacking public fingerprint datasets by exploring methods of latent fingerprint generation using Generative adversarial networks [17]. There are many commercial, and even some open source programs, that are capable of synthesizing fingerprint images [7, 1]. However, these are often clean fingerprint images, which are not as valuable, as latent fingerprint images. The amount of publications, that aim to generate realistic images of latent fingerprints, given the images of clean fingerprints, is very slim. One of the important criteria of such framework is preserving identity of the original clean image, so that the model trained on a dataset created by this framework learns to identify fingerprints correctly. Furthermore, it is also beneficial, if one can control identity and style of the generated image separately, and also generate multiple impressions of the same fingerprint. In this thesis, model called AugNet [39] is used as a baseline for experiments with latent fingerprint generation.

Chapter 2 talks about theoretical and practical aspects of Generative adversarial networks. Chapter 3 provides brief overview of fingerprint recognition and image processing techniques, as well as an extensive insight into different methods of fingerprint generation, including state of the art approaches. In Chapter 4, baseline model and training algorithm are described, together with a brief description of experiment framework and setup. Experiments with the different model and training algorithm variants, as well as evaluation methods, and their results, are summarized in Chapter 5. Finally, Chapter 6 concludes the achieved progress, and briefly discusses possibilities for future improvements.

Chapter 2

Generative adversarial networks

Generative adversarial networks (GANs) [17] are a type of neural network used for generating data. Nowadays there are many subtypes of these networks, and they are used in a lot of areas of machine learning. However, they share the same traits, such as generating data from a set of inputs, and training in adversarial manner.

2.1 The original GAN

In this section, I will be describing GAN model, as it was first introduced in 2014. It consists of two parts, generator G and discriminator D. G and D are differentiable functions, for example multi-layer perceptrons. The goal of the training is for generator to learn distribution over input data $\mathbf{x}$, which is a vector of unpaired samples. The generator is then used for generating data samples as $G(\mathbf{z}, \theta_G)$, where $\mathbf{z}$ is a random noise variable sampled from prior distribution $p_{\mathbf{z}}(\mathbf{z})$, and θ_G are parameters of the generator. Essentially, G learns a mapping from noise space to data space.

Discriminator D is defined as $D(\mathbf{y}, \theta_D)$, where $\mathbf{y}$ is its input, and θ_D are the parameters. It outputs a single number between 0 and 1, that expresses the probability, that given input is *real*. Input is considered real, if it came from the data $\mathbf{x}$. If it was produced by the generator, it is considered *fake*. Discriminator learns to distinguish between real and fake inputs.

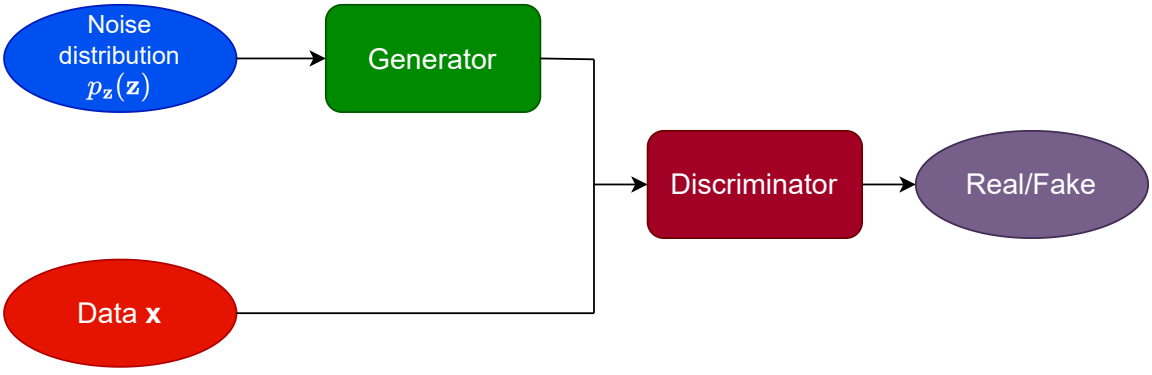


Figure 2.1: High-level architecture of the GAN model.

2.2 GAN training

For training a neural network, some criterion is needed. We can define value function $V(G, D)$ as:

$$V(G, D) = \mathbb{E}_{\mathbf{x} \sim p_{data}(\mathbf{x})}[logD(\mathbf{x})] + \mathbb{E}_{\mathbf{z} \sim p_{\mathbf{z}}(\mathbf{z})}[log(1 - D(G(\mathbf{z})))], \tag{2.1}$$

where $\mathbb{E}_{\mathbf{x} \sim p_{data}(\mathbf{x})}$ is expected value over all data $\mathbf{x}$, and $\mathbb{E}_{\mathbf{z} \sim p_{\mathbf{z}}(\mathbf{z})}$ is expected value over all sampled noise vectors. During training, generator tries to minimize this function, and discriminator tries to maximize this function. However, we cannot call this a loss function, since in general, we want to minimize loss functions. From this, we can derive loss functions for discriminator and generator as:

$$L_D(\mathbf{x}, \mathbf{z}) = log(D(\mathbf{x})) + log(1 - D(G(\mathbf{z}))) \tag{2.2}$$

$$L_G(\mathbf{z}) = log(1 - D(G(\mathbf{z}))) \tag{2.3}$$

As we can see from the equations, loss functions are different for discriminator and generator. This is because they are trained separately. As only the output of discriminator is considered in both loss functions, the generator needs to be trained via backpropagation of gradients from the discriminator. Training procedure is shown in Algorithm 1.

Algorithm 1 GAN training algorithm

Inputs: Generator G, discriminator D, data $\mathbf{X}$, prior noise distribution $p_{\mathbf{z}}(\mathbf{z})$.
Hyperparameters: Number of epochs E, number of batches n, batch size b.
Objective: Train GAN model.

for epoch = 1,...,E **do**
  for batch = x_1,...,x_n **do**
    Sample batch of vectors $\mathbf{z} = \{z_1, ..., z_b\}$ from $p_{\mathbf{z}}(\mathbf{z})$.
    Sample batch of data $\mathbf{x} = \{x_1, ..., x_b\}$ from data $\mathbf{X}$.
    Calculate loss of discriminator $l_D = \frac{1}{b} \sum_{i=1}^{b} L_D(x_i, z_i)$.
    Calculate gradients g_D of discriminator w.r.t. l_D.
    Update discriminator weights θ_D using g_D.
    Sample batch of vectors $\mathbf{z} = \{z_1, ..., z_b\}$ from $p_{\mathbf{z}}(\mathbf{z})$.
    Calculate loss of generator $l_G = \frac{1}{b} \sum_{i=1}^{b} L_G(z_i)$.
    Calculate gradients of generator g_G w.r.t. l_G.
    Update generator weights θ_G using g_G.
  end for
end for

2.3 Conditional GAN

In traditional GAN architecture, the generator takes only noise vector as an input. Conditional GAN (cGAN) [28] is a subtype of GAN, where the generator (and optionally also a discriminator) are fed some additional data $\mathbf{y}$. In general, $\mathbf{y}$ can be any type of data, such as class label, or some vector of real numbers, etc. When a model is fed this additional data $\mathbf{y}$, we say it is *conditioned* on $\mathbf{y}$.

In the original paper, the discriminator is fed the conditional input $\mathbf{y}$ as well as the generator. However, that is not the case for all the architectures. Some architectures

just provide the discriminator with the original data (either from the dataset or from the generator). Because of that, the value function can be described in multiple ways. In the case of feeding conditional input into the discriminator, the value function is defined by equation 2.4. In other case, it is defined by equation 2.5. Subsequently, we can derive loss functions. Loss function for the generator is defined in equation 2.6. Loss function for the discriminator depends on the value function, in the case of $V_1(G, D)$ it takes form of the equation 2.7, in other case it takes form of equation 2.8.

$$V_1(G, D) = \mathbb{E}_{\mathbf{x} \sim p_{data}(\mathbf{x})}[logD(\mathbf{x}|\mathbf{y})] + \mathbb{E}_{\mathbf{z} \sim p_{\mathbf{z}}(\mathbf{z})}[log(1 - D(G(\mathbf{z}|\mathbf{y})))] \tag{2.4}$$

$$V_2(G, D) = \mathbb{E}_{\mathbf{x} \sim p_{data}(\mathbf{x})}[logD(\mathbf{x})] + \mathbb{E}_{\mathbf{z} \sim p_{\mathbf{z}}(\mathbf{z})}[log(1 - D(G(\mathbf{z}|\mathbf{y})))] \tag{2.5}$$

$$L_G(\mathbf{y}, \mathbf{z}) = log(1 - D(G(\mathbf{z}|\mathbf{y}))) \tag{2.6}$$

$$L_{D1}(\mathbf{x}, \mathbf{y}, \mathbf{z}) = log(D(\mathbf{x}|\mathbf{y})) + log(1 - D(G(\mathbf{z}|\mathbf{y}))) \tag{2.7}$$

$$L_{D2}(\mathbf{x}, \mathbf{y}, \mathbf{z}) = log(D(\mathbf{x}) + log(1 - D(G(\mathbf{z}|\mathbf{y}))) \tag{2.8}$$

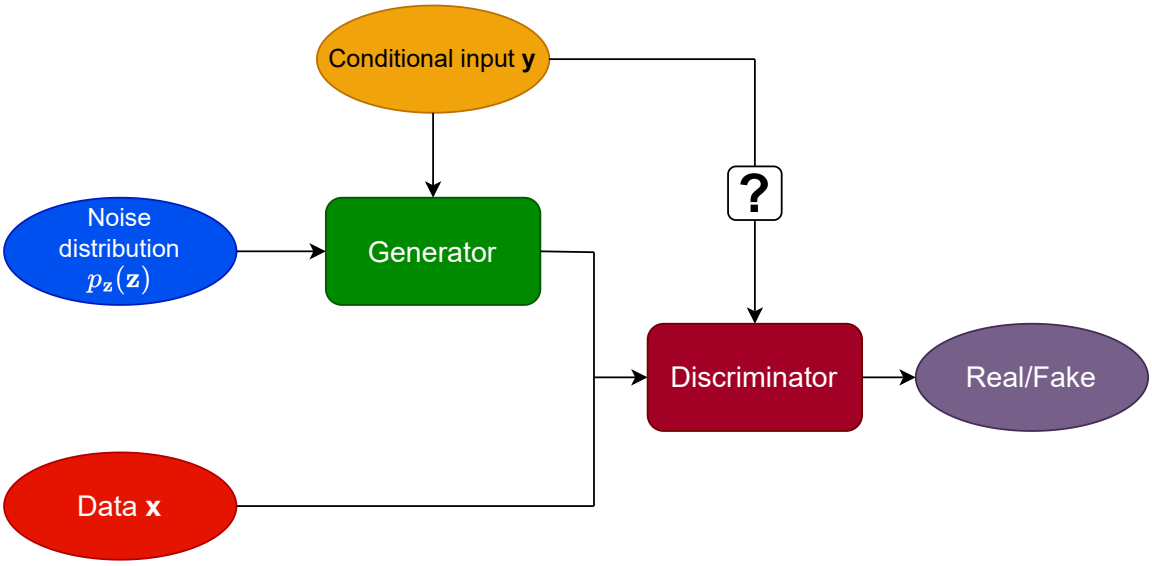


Figure 2.2: High-level architecture of generic cGAN model. The question mark means that discriminator may or may not be given the conditional input.

2.4 Deep convolutional GAN

So far, only the high-level architectures of GAN networks were discussed, providing little to no description of generator and discriminator internals. In the papers mentioned up to this point, generator and discriminator were defined as multi-layer perceptrons. However, the only theoretical constraint for their architectures is that they have to be differentiable functions with some hidden learnable parameters.

Deep convolutional GAN (DCGAN) [30] is a subtype of GAN, that uses multiple convolutional layers in the generator and the discriminator. Usually, generator utilizes an upsampling architecture, producing image from noise vector. Analogically, discriminator uses downsampling architecture, producing a probability scalar from an input image. Example architecture of generator can be seen in Figure 2.3.

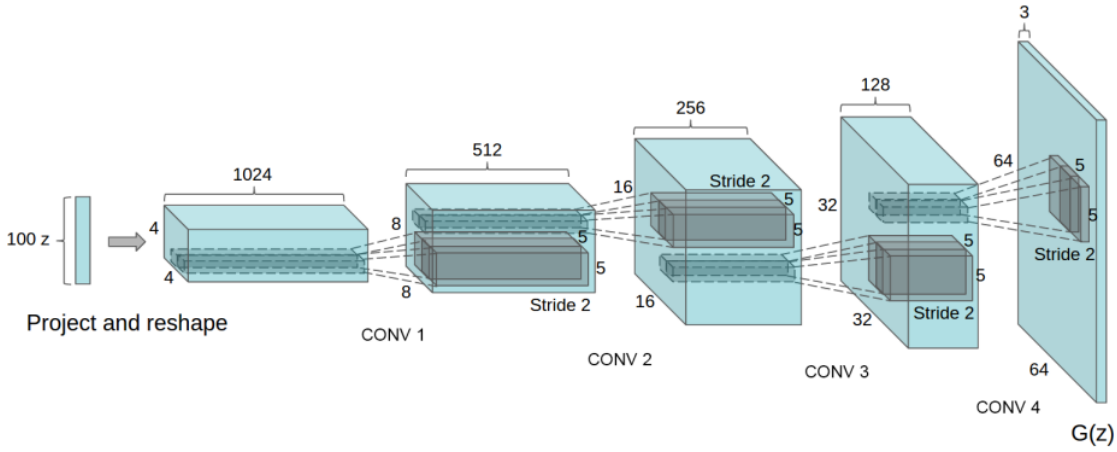


Figure 2.3: Example architecture of generator in a typical DCGAN model [30]. The input is a noise vector, which is passed through fractional convolutions. Finally, output image is generated. Discriminator works the opposite way, using convolutional layers to decrease image size and produce final output.

2.5 StyleGAN

StyleGAN [23] is a GAN image architecture that utilizes style transfer. It separates style from main characteristics, and disentangles latent space vector. Difference between traditional generator and StyleGAN generator is outlined in Figure 2.4. Unlike traditional generator, StyleGAN generator starts upsampling from learned constant matrix. Upsampling is done by synthesis network g, which is a stack of convolutional layers, gradually increasing the image resolution. StyleGAN maps input latent vector $\mathbf{z}$ into intermediate latent vector w via mapping network f, which is a multi-layer perceptron. Vector w is transformed via learned affine transformations A into vector of scale and shift weights. These weights are applied to each layer via AdaIN [20] block, which adds style to the image. There is also random noise input, that is added to each layer, after being scaled by learned weights in B.

The interesting property of this architecture is that due to different learned transformations in each layer, different parts of the style are applied in each layer, despite coming from the same latent vector $\mathbf{w}$. In lower resolution layers, basic features, such as image identity are created. On the other hand, in higher resolution layers, finer details, such as small distortions are synthesized.

Adaptive instance normalization

In StyleGAN, adaptive instance normalization (AdaIN) [20] is used to apply style to feature maps. It is based on traditional instance normalization. Instance normalization computes statistics μ and σ for each sample and channel separately. Considering that x is a single feature map (one channel output) from a convolutional layer, we can calculate these statistics as:

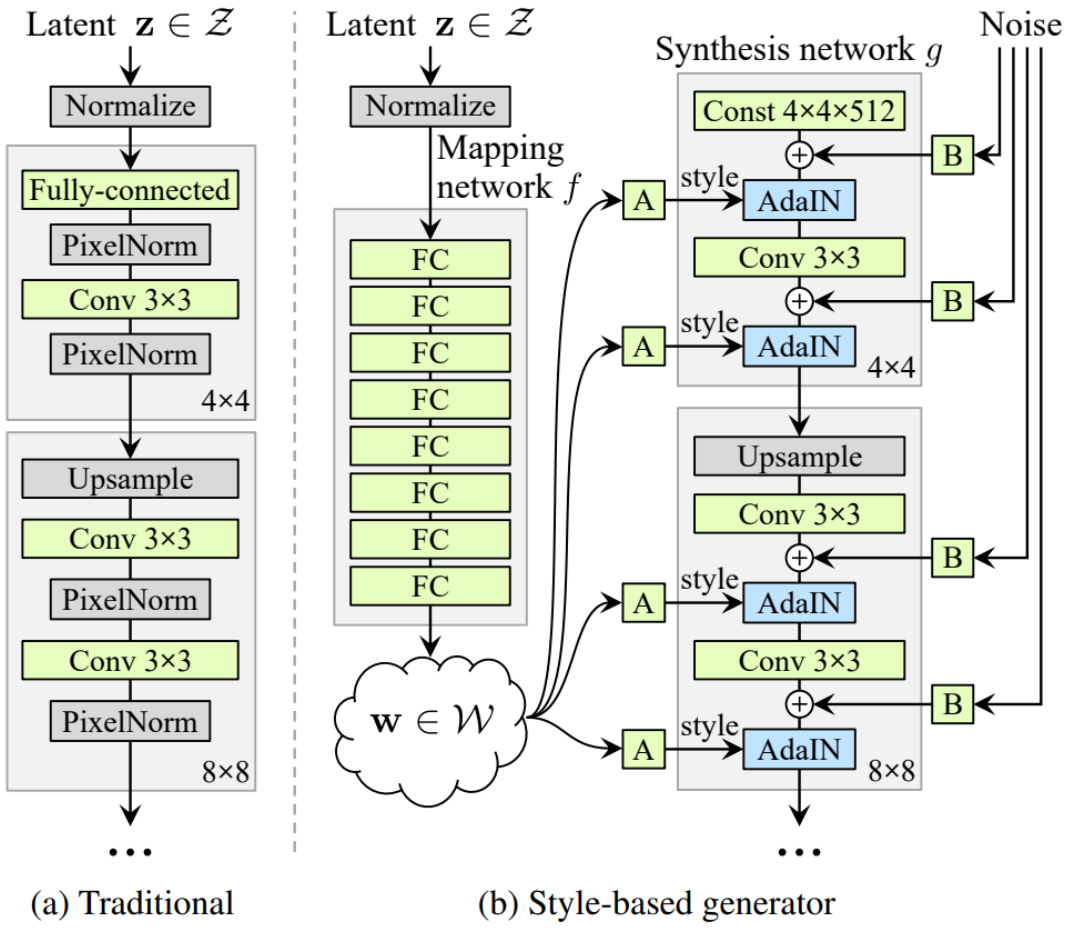


Figure 2.4: Differences between traditional generator and StyleGAN generator [23]. In traditional upsampling architecture, latent noise vector is upsampled by multiple convolutional layers to produce output image. In StyleGAN, latent vector $\mathbf{z}$ is first transformed into intermediate latent space as vector $\mathbf{w}$ by multi-layer perceptron network f. Synthesis network g also uses upsampling, but it starts from learned constant matrix instead of latent vector. After each convolution, style vector is joined with the convolution output using AdaIN [20] block. The style vector is created from intermediate latent vector $\mathbf{w}$ using learned affine transformation A. Random noise vectors are also added in between convolutional layers after being scaled by learned weights in B.

$$\mu(\mathbf{x}) = \frac{1}{HW} \sum_{h=1}^{H} \sum_{w=1}^{W} \mathbf{x}_{hw} \tag{2.9}$$

$$\sigma(\mathbf{x}) = \sqrt{\frac{1}{HW} \sum_{h=1}^{H} \sum_{w=1}^{W} (\mathbf{x}_{hw} - \mu(\mathbf{x}))^2 + \epsilon} \tag{2.10}$$

where H and W are height and width of the feature map, $\mathbf{x}_{hw}$ is value in $\mathbf{x}$ at column h and row w, and ϵ is a small positive real number to prevent division by zero in equation 2.11.

These statistics are then utilized in AdaIN operation, which is evaluated for each feature map as:

$$AdaIN(\mathbf{x}_i, \mathbf{y}) = \mathbf{y}_{s,i} \frac{\mathbf{x}_i - \mu(\mathbf{x}_i)}{\sigma(\mathbf{x}_i)} + \mathbf{y}_{b,i} \tag{2.11}$$

where $\mathbf{x}_i$ is the i-th feature map, $\mathbf{y}$ is transformed style vector, and $\mathbf{y}_{s,i}$, $\mathbf{y}_{b,i}$ are its scale and bias for i-th feature map. Since the style vector comes from affine transformation A, its output size has to be twice the number of the feature maps.

2.6 PatchGAN

In traditional GAN, discriminator classifies whole input image as real or fake. PatchGAN [21] architecture utilizes a different approach. There are multiple ways to implement this network. First of the implementations reshapes and transposes the input, converting it into a matrix of non-overlapping $N \times N$ sized image patches. Then, for each of these patches, stack of convolutional layers is applied. The output is one scalar for each image patch. However, in official PatchGAN[1] implementation, different method is used. This method simply applies all convolutional layers, which in turn yields output of the same shape, as in the first approach. The difference is that in the second approach, individual scalars in the output correspond to overlapping receptive fields in the input, unlike in the first one, where the image is explicitly split into patches first (Figure 2.5).

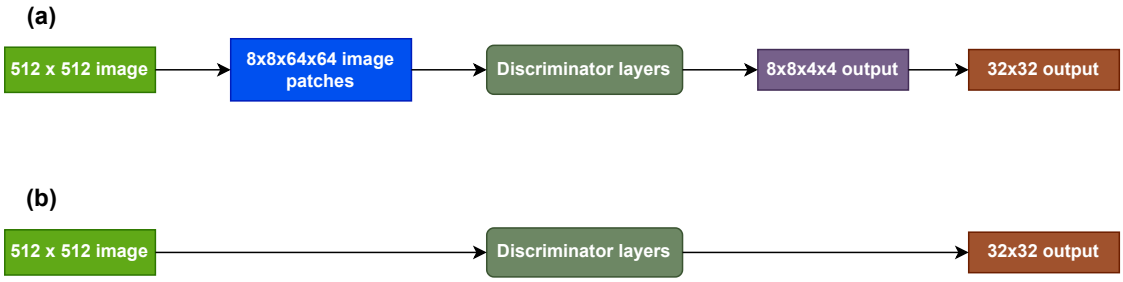


Figure 2.5: Comparison of PatchGAN variants. (a) First ad-hoc implementation, where image is split into smaller patches, then convolutional layers are applied as if the patches are separate images. (b) Widely used PatchGAN implementation, layers are simply applied on input image, the output of last layer has the same shape as (a) after reshaping. The main difference is that in (b) receptive fields are overlapping.

2.7 Least Squares GAN

As is shown in section 2.2, GAN networks utilize binary cross-entropy loss function. This function is preceded by sigmoid, which is part of the discriminator in the formulas. Least squares GAN (LSGAN) [26] on the other hand utilizes least squares error, also omitting the sigmoid function. Authors claim, that it can alleviate some of the problems occurring during GAN training, such as instability, or vanishing gradients. This is because the combination of sigmoid and binary cross-entropy does not penalize samples, that are on correct side of

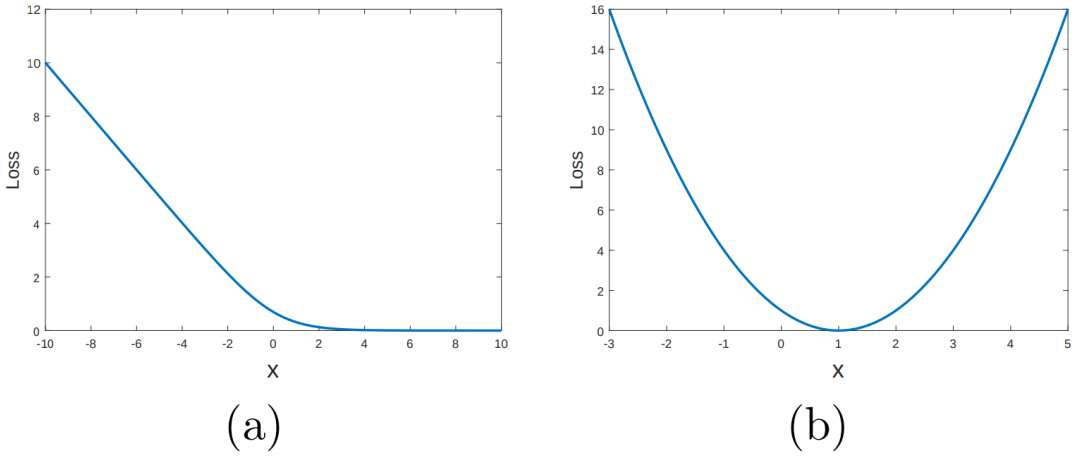
[1] https://github.com/phillipi/pix2pix

decision boundary, but far away. This happens during generator training, and the generator is not forced to generate samples closer to decision boundary, in order to fool discriminator. The LSGAN loss penalizes this, making gradients larger and thus moderating vanishing gradients problem. In other words, it makes the interval, where the loss function has relatively low value, smaller (as can be seen in Figure 2.6), thus increasing the pressure on the generator via gradients to learn more. The loss functions for discriminator and generator can have a few small variations, but they show similar results. One of configurations of these loss functions is defined as:

$$L_D(\mathbf{x}, \mathbf{z}) = \frac{1}{2}(D(\mathbf{x}) - 1)^2 + \frac{1}{2}(D(G(\mathbf{z})))^2 \tag{2.12}$$

$$L_G(\mathbf{z}) = \frac{1}{2}(D(G(\mathbf{z})) - 1)^2 \tag{2.13}$$

where G and D are considered without the last sigmoid layer.



(a) (b)

Figure 2.6: Comparison of sigmoid followed by cross-entropy (a) and least squares error (b) [26]. The interval of relatively low loss function value is smaller in LSGAN, which puts more pressure on the generator to learn via gradients.

2.8 WGAN-GP

WGAN

Wasserstein GAN or WGAN [2] is one of the GAN training algorithms. Unlike traditional GAN algorithm, it is based on minimizing so-called *Earth Mover* distance between 2 probability distributions. Let´s say that the probability distribution of real data is $\mathbb{P}_r$, and of generated data is $\mathbb{P}_q$, then Earth Mover distance is defined as:

$$W(\mathbb{P}_r, \mathbb{P}_g) = \inf_{\gamma \in \Pi(\mathbb{P}_r, \mathbb{P}_g)} \mathbb{E}_{(x,y)\sim\gamma}[\|x - y\|] \tag{2.14}$$

Minimizing this metric directly is intractable, so it has to be approximated in some way. One of the ways is to put so-called *Lipschitz* constraint on discriminator (also called *critic*

in this case). The objectives for the discriminator and the generator then become:

$$\mathcal{L}_D = \mathbb{E}_{\tilde{\mathbf{x}}\sim\mathbb{P}_g}[D(\tilde{\mathbf{x}})] - \mathbb{E}_{\mathbf{x}\sim\mathbb{P}_r}[D(\mathbf{x})] \tag{2.15}$$

$$\mathcal{L}_G = -\mathbb{E}_{\mathbf{z}\sim p_\mathbf{z}}[D(G(\mathbf{z})] \tag{2.16}$$

where $p_\mathbf{z}$ is prior noise distribution. The question then becomes how to put a Lipschitz constraint on the discriminator. In the original paper, they use weight clipping, where after each discriminator update its weights are clipped into an interval of $[-0.01, 0.01]$. However, as the authors themselves claim, weight clipping is not a good way to enforce Lipschitz constraint. The training is very sensitive to clipping threshold, leading either to slow or no convergence, or vanishing gradients.

WGAN-GP

WGAN-GP [18] is a modification of WGAN algorithm that proposes a novel way of enforcing Lipschitz constraint on the discriminator. It does so by introducing gradient penalty, so that the discriminator has gradients with norm 1 almost everywhere. The discriminator loss objective is then changed to:

$$\mathcal{L}_D = \mathbb{E}_{\tilde{\mathbf{x}}\sim\mathbb{P}_g}[D(\tilde{\mathbf{x}})] - \mathbb{E}_{\mathbf{x}\sim\mathbb{P}_r}[D(\mathbf{x})] + \lambda\mathbb{E}_{\hat{\mathbf{x}}\sim\mathbb{P}_{\hat{\mathbf{x}}}}[(\|\nabla_{\hat{\mathbf{x}}}D(\hat{\mathbf{x}})\|_2 - 1)^2] \tag{2.17}$$

where the second term is newly added gradient penalty term, and λ represents the gradient penalty weight. Full training procedure is defined in Algorithm 2. Note that in this case, the discriminator is trained multiple times per generator update, denoted by constant n_{critic}.

Algorithm 2 WGAN-GP training algorithm [18]

Hyperparameters: Gradient penalty weight λ, number of critic iterations per generator iteration n_{critic}, batch size m, Adam [24] hyperparameters α, β_1, β_2.
Inputs: Initial critic parameters w_0, initial generator parameters θ_0.
Objective: Train model using WGAN-GP algorithm.

while θ has not converged **do**
  for $t = 1, ..., n_{critic}$ **do**
    for $i = 1, ..., m$ **do**
      Sample real data $\mathbf{x} \sim \mathbb{P}_r$
      Sample latent variable $\mathbf{z} \sim p(\mathbf{z})$
      Sample random number $\epsilon \sim U[0, 1]$
      $\tilde{\mathbf{x}} \leftarrow G_\theta(\mathbf{z})$
      $\hat{\mathbf{x}} \leftarrow \epsilon\mathbf{x} + (1 - \epsilon)\tilde{\mathbf{x}}$
      $L^{(i)} \leftarrow D_w(\tilde{\mathbf{x}}) - D_w(\mathbf{x}) + \lambda(\|\nabla_{\hat{\mathbf{x}}}D_w(\hat{\mathbf{x}})\|_2 - 1)^2$
    end for
    $w \leftarrow \text{Adam}(\nabla_w \frac{1}{m}\sum_{i=1}^m L^{(i)}, w, \alpha, \beta_1, \beta_2)$
  end for
  Sample a batch of latent variables $\{\mathbf{z}^{(i)}\}_{i=1}^m \sim p(\mathbf{z})$
  $\theta \leftarrow \text{Adam}(\nabla_\theta \frac{1}{m}\sum_{i=1}^m -D_w(G_\theta(\mathbf{z})), \theta, \alpha, \beta_1, \beta_2)$
end while

2.9 Fréchet inception distance

Fréchet inception distance (FID) [4] is a metric for evaluating image quality produced by generative models. It is widely used with Generative Adversarial Networks. It is defined

as a *Wasserstein-2* distance between two multivariate Gaussian distributions. Given that μ_r, Σ_r are mean and covariance of real data distribution, and μ_g, Σ_g are mean and covariance of generated data distribution, the FID is then calculated as:

$$FID(r, g) = \|\mu_r - \mu_g\| + Tr\left(\Sigma_r + \Sigma_g - 2\sqrt{\Sigma_r \Sigma_g}\right) \tag{2.18}$$

where Tr is a trace operation. The parameters of distributions are calculated on latent space embeddings of data created by a neural network, usually pretrained InceptionV3[2]. Last classification layer is omitted, and output of the network is a 2048-dimensional feature vector for each input image.

[2]https://www.tensorflow.org/api_docs/python/tf/keras/applications/inception_v3/InceptionV3

Chapter 3

Fingerprint generation

Fingerprint recognition is a large field in biometrics research. Many machine learning methods are used to tackle this issue, mainly neural networks. Neural networks usually require large amounts of high-quality data for training in order to provide desired results. However, as many fingerprint datasets, such as NIST SD4 [38], NIST SD14 [37], and NIST SD27 [16] have been retracted from public use due to privacy regulations [12], it is becoming increasingly difficult to develop and train large neural networks for fingerprint recognition. One of the possible solutions for this problem is to generate new fingerprint images, and create new large-scale datasets for training models. In this chapter, various methods of generating fingerprints will be described, focusing on utilizing GANs. Special emphasis will be given on generating latent fingerprints, as well as identity preservation, as these represent the real challenge in the field of fingerprint recognition.

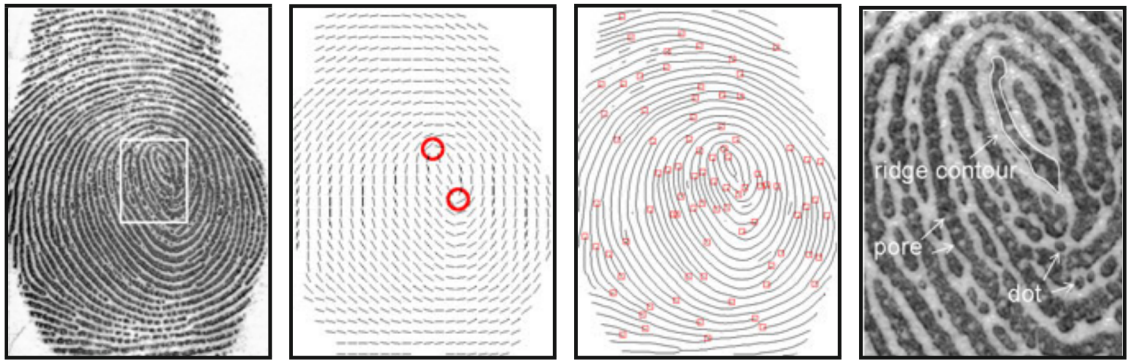
3.1 Fingerprint basics

Fingerprint identification

Fingerprint details can be divided into 3 classes [36], as is shown in Figure 3.1:

- **Level 1 (L1)** - Global patterns, such as ridge orientation maps
- **Level 2 (L2)** - Local patterns, such as minutiae
- **Level 3 (L3)** - Fine details, such as sweat pores or scratches

Fingerprint identification is carried out based on these characteristics. There are different identification methods, however, the most used techniques rely on minutiae (L2) detection. This is for a multitude of reasons, first being that minutiae contain majority of identity information. Second reason is that minutiae information is memory-efficient. Besides that, extraction of minutiae is also robust to many sources of degradation [22].



(a) (b) (c) (d)

Figure 3.1: Different levels of fingerprint details [22]. (a) Original fingerprint image (b) Level 1 - ridge orientation maps (c) Level 2 - minutiae markers (d) Level 3 - ridge contour, pores and dots

Fingerprint types

Fingerprint images can be divided into 5 different types based on capture method and count of fingerprints: [13] [34] [35]

1. **Plain** - fingerprint is pressed down on a flat surface
2. **Rolled** - fingerprint is rolled from one side to the other (nail-to-nail) in order to capture all the details
3. **Latent** - fingerprint is lifted from a surface that was touched by a person
4. **Slap** - multiple fingerprints on one or both hands are captured simultaneously, also called simultaneous plain
5. **Simultaneous latent** - multiple latent fingerprints are lifted from a surface

Each fingerprint type, that is not latent (plain, rolled, slap), will be called *clean*. Fingerprint image, that contains only 2 levels of activation (0 and 1), is called *binarized*. Fingerprint, that is 3 of these types are shown in Figure 3.2.

Fingerprint alignment

During fingerprint acquisition, different placements of finger on the scanner may result in different impressions of the finger. Alignment process is a transformation of one of the images, such that it is geometrically aligned with the other one. For that, transformation model has to be specified. In general, rigid transformation is sufficient for this task. In the case of more intense non-linear deformations introduced during fingerprint scanning, more complex methods, such as Generalized Hough transformation can be used [22]. Another issue is selecting appropriate method for estimating transformation parameters, given existing information about the fingerprint. One of such challenges will be discussed in section 4.2.

(a) (b) (c)

Figure 3.2: Different types of fingerprint images for the same fingerprint [22]. (a) Rolled fingerprint (b) Plain fingerprint (c) Latent fingerprint.

3.2 Image processing

Projective transformation

Projective transformation [25] is a type of geometric transformation. It is the most general type of homography transformation. Regarding fingerprints, 2D transformations are relevant. The projective transformation can be defined for a point $[x, y]$ in 2D space as:

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & v \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} \tag{3.1}$$

where $[x', y']$ would be a transformed point in that space, a_{ij} are transformation parameters, and v is a constant, either 0 or 1. This transformation has 8 degrees of freedom, corresponding to the parameters. These parameters can also be estimated, given a set of pairs of points before and after transformation.

Otsu´s binarization

There are multiple algorithms for fingerprint binarization. Some of them require commercial software. Given that the fingerprint image is not substantially degraded, simple thresholding methods can be used. One of them is called Otsu´s thresholding[1] [29] (also Otsu´s binarization), which works by finding the image threshold, that minimizes intra-class variance in the image. Any threshold will divide the image into 2 parts, black and white pixels. These 2 parts are understood as classes in this context, meaning the algorithm tries to minimize variance within them. The variance σ^2 at threshold t is computed as:

$$\sigma^2(t) = \omega_{bg}(t)\sigma^2_{bg}(t) + \omega_{fg}(t)\sigma^2_{fg}(t) \tag{3.2}$$

[1] https://muthu.co/otsus-method-for-image-thresholding-explained-and-implemented/

where $\sigma_{bg}(t)$ ($\sigma_{fg}(t)$) is the variance in values of pixels, that would be under (over) threshold t, and $\omega_{bg}(t)$, $\omega_{fg}(t)$ are probabilities of these classes (calculated as number of pixels of given class divided by a total number of pixels). The algorithm finds such threshold t for which σ^2 has the minimal value.

3.3 Related work

In this section, existing methods of generating fingerprints using GANs will be described. Only 2 of these methods are focused on latent fingerprint generation. All of these methods aim to generate gray-scale fingerprint images.

Finger-GAN

Finger-GAN [27] is a GAN-based approach designed to generate realistic fingerprint images. The model used is a simple convolutional GAN, where both generator and discriminator have 5 layers. The model was trained separately on 2 datasets, FVC 2006 [8] and PolyU [41], and the results were also evaluated separately. The key difference of this approach is the use of custom loss function. Sometimes, images generated by GAN models can contain tiling patterns or a lot of unwanted noise. Since fingerprints are typically composed of many fine lines, these lines need to be connected correctly between individual image rows. To achieve this line connectivity, anisotropic version of 2D total variation [9] was used. Total variation measures the amount of variation between neighbouring pixels. It is computed as:

$$TV(Y) = \sum_{i,j} |y_{i+1,j} - y_{i,j}| + |y_{i,j+1} - y_{i,j}| \tag{3.3}$$

where Y is the input image, and i, j are row, resp. column indices. It is a sum of variations between all neighbouring pixels, both horizontally and vertically. Minimizing this quantity should produce smoother and more connected images. The loss function is then defined as:

$$\mathcal{L}_{GAN-TV} = \mathbb{E}_{\mathbf{x}\sim p_{data}(\mathbf{x})}[logD(\mathbf{x})] + \mathbb{E}_{\mathbf{z}\sim p_{\mathbf{z}}(\mathbf{z})}[log(1 - D(G(\mathbf{z})))] + \lambda TV(G(\mathbf{z})) \tag{3.4}$$

where λ is a hyperparameter determining the weight of total variation in the loss function.

The images produced by this network look considerably realistic for both datasets. However, the network was not trained to generate latent fingerprints. Furthermore, it is not possible to add conditional identity information into the network, which makes it practically impossible to generate multiple instances of one fingerprint.

Level 3 fingerprint generation

As mentioned in section 3.1, fingerprint details can be classified into 3 levels. The approach described in the paper [36] is focused on generating L3 fingerprint images. First, they generate new clean fingerprint using open-source software called Anguli [1]. After slightly dynamically changing ridge thicknesses, this image is called master fingerprint. After that, pores and scratches are added to this image using distribution learned from real data. Then, fingerprint acquisition is simulated, using cropping and random affine transformations. Resulting images are then passed through a series of augmentation steps, producing a set of so-called seed images. The final step consists of adding realistic texture to the seed image using CycleGAN [42]. CycleGAN can learn a mapping between 2 image domains without prior knowledge of inter-domain image pairs. Real fingerprint data are taken from PolyU

[41] database. Similar to seed images, they are also augmented. Then, the CycleGAN is used to learn the mapping between seed images and real images. Using the trained CycleGAN, they generated synthetic fingerprint dataset called L3-SF, totalling 7400 fingerprint samples. The process is illustrated in Figure 3.3.

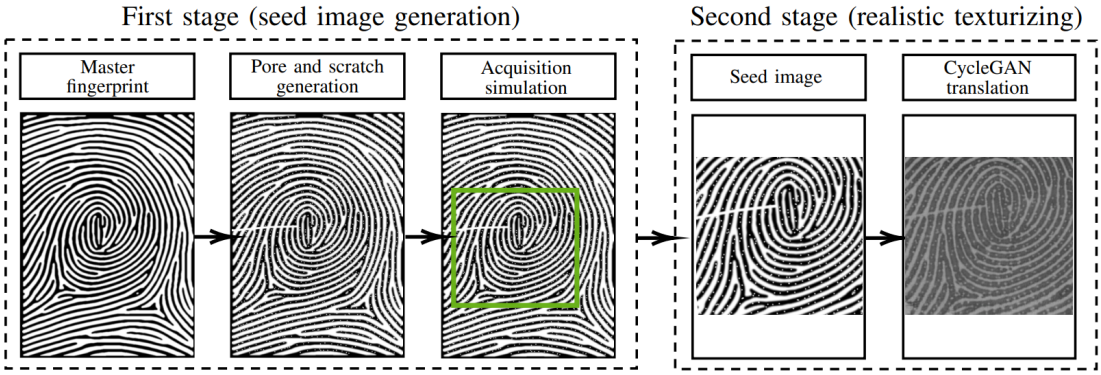


Figure 3.3: Level 3 fingerprint generation approach [36]. First, master fingerprint is generating using Anguli [1]. Then, pores and scratches are added, according to real distribution. Fingerprint acquisition is simulated using cropping (again learned from real data) and random affine transformations. Resulting images are augmented to create seed image set. CycleGAN [42] is utilized to learn a mapping between the seed images, and augmented subset of PolyU [41] dataset.

It was concluded, that the generated fingerprints look realistic, based on human perception test using 60 volunteers. The learned mapping also seems to preserve fingerprint identity. Although realistic L3 details are generated, there is no mention of generating latent fingerprints using this method. As CycleGAN learns 1:1 mapping between 2 image domains, it is also not viable to generate multiple impressions of the same fingerprint. Authors claim, that this step is done when simulating multiple different fingerprint acquisitions. However, the diversity of these images does not match the diversity present in latent fingerprints. In addition, it is not able to simulate different environments (backgrounds) present in latent fingerprints.

PrintsGAN

PrintsGAN [12] is an approach at generating fingerprints using multiple GAN models. It consists of three stages. In the first stage, binary fingerprint, which is called Master-Print is synthesized. In this process, BigGAN [6] architecture is utilized to generate 256 $\times$ 256 binary fingerprints from 512-dimensional noise vector. For training this GAN G_I, large binary fingerprint dataset is needed. For that, MSP longitudinal database [40] was used. This dataset contains 282K gray-scale images, so the binarized versions of them were created. First, commercial software (Verifinger v12 SDK) was used to extract binarized versions for the subset of 10K images from the dataset. Subsequently, auto-encoder R was trained on this extracted dataset to translate from gray-scale to binary images using L2 loss. Through this auto-encoder, binary dataset for training the the first stage GAN was created.

In the second stage, warping and cropping are added to master Master-Print. For this, another GAN G_w is used. This network consists of encoder E_w, decoder D_w, and warping

encoder L_w. First, the encoder transforms input image I_{ID} into feature maps. Simultaneously, warping encoder transforms its input 16-dimensional noise vector into parameters Θ. From the extracted feature maps, decoder computes a segmentation mask S, which simulates cropping the image. Warping is carried out by Thin Plate Spline transformation $\mathcal{F}(I, \Theta)$, which aims to simulate different finger placements on the scanner. The final output is computed as $I_w = \mathcal{F}(I_{ID}, \Theta) * S$, using the parameters from the warping encoder.

In the last stage, texture is added to warped and cropped fingerprint. Here, BigGAN [6] architecture is utilized again. It is however modified to integrate texture noise into the image via instance normalization. The texture is represented as 128-dimensional noise vector, which is encoded into parameters of instance normalization, similar to StyleGAN in section 2.5. During training, discriminator compares real fingerprint images to the generated ones. Finally, to ensure that identity of the generated fingerprint is preserved, trained auto-encoder R is used to transform textured image I_r to its binary form, and it is compared to image before texturing I_w via L2 loss. Whole process of generating fingerprints is outlined in Figure 3.4.

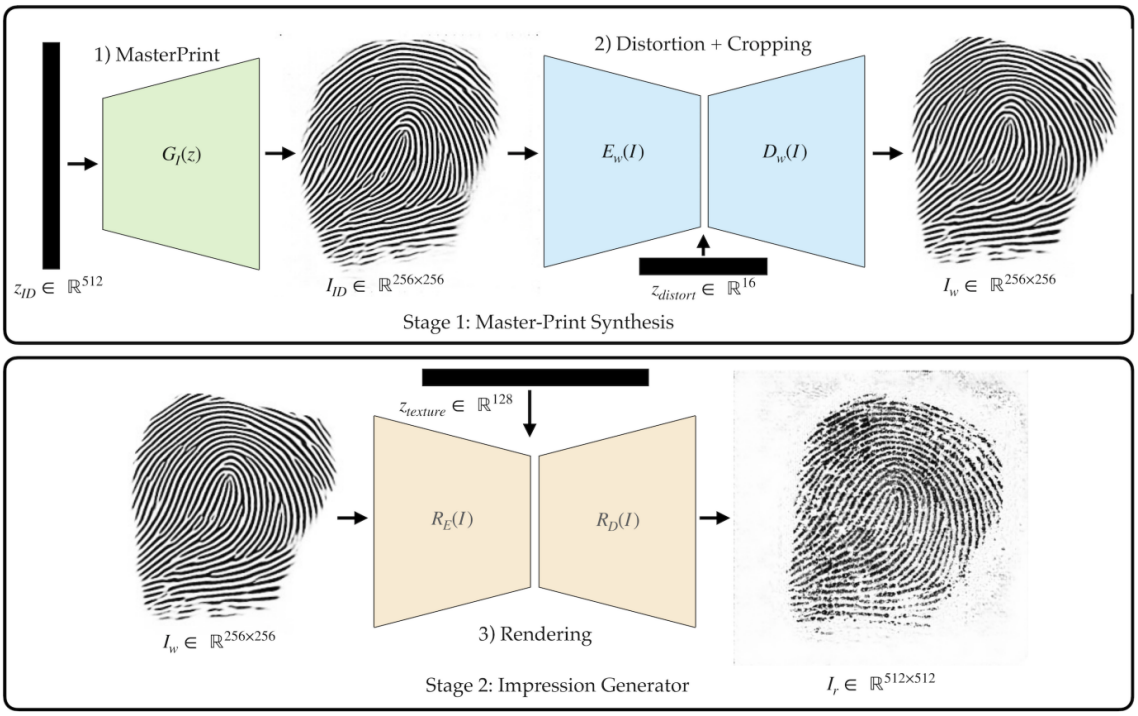


Figure 3.4: PrintsGAN architecture [12]. Whole model consists of 3 GANs. Master print is synthesized from noise vector z_{ID} using GAN G_I, utilizing MSP longitudinal database [40] and trained auto-encoder R in the training stage. Warping and cropping are added to the master print to simulate pressing finger on a scanner. This is carried out by GAN G_w, using Thin Plate Spline transformation. Vector $z_{distort}$ controls the transformation parameters. Transformed image is then passed to the third GAN G_r, where realistic texture is added, dictated by $z_{texture}$ vector. The final image is then converted to binary form via auto-encoder R, and compared to output of the warping network I_w using L2 loss, to ensure identity preservation.

PrintsGAN showed great results, with capability to generate realistic fingerprints, tested by expert human subjects. It can generate multiple impressions per fingerprint (with variations in both warping and texture), and it preserves the identity of master print during texturing stage. Furthermore, all of the components of the fingerprint image (identity, warping, texture) can be controlled by individual vectors, making it a great tool for generating large amounts of high-quality fingerprints. The only disadvantage of this approach is, that it can not generate latent fingerprints, because it was not trained to do so.

Synthetic latent fingerprint generator

The approach proposed in the paper [5] aims to generate latent fingerprints from rolled fingerprints. The training is split into 2 stages. In the first stage, CycleGAN [42] model is trained to learn a mapping between rolled and latent fingerprints. Rolled fingerprints for training are taken from NIST SD4 [38] database, whereas latent ones are from MSP latent database [40]. CycleGAN architecture is modified to use both global and patch discriminators, to improve stability. After the CycleGAN is trained, the latent fingerprint database is run through pre-trained ResNet152V2[2] [19] network, to extract image embeddings. These embeddings are taken as the output of the last fully connected layer, with 2048 features. Then, k-means clustering is applied on the embeddings to split these images into $k = 3$ categories, namely Good, Bad, and Ugly. This division comes from NIST SD27 [16] dataset, where the fingerprints are split this way, based on their quality. For each of these clusters, separate CycleGAN model is created, as a fine-tuned version of the model trained in the first stage. The images from the given cluster are used as a training data for finetuning. Finally, using these models, synthetic latent fingerprints belonging to any of the 3 categories can be created, by passing rolled fingerprints via respective fine-tuned model. The process is outlined in Figure 3.5.

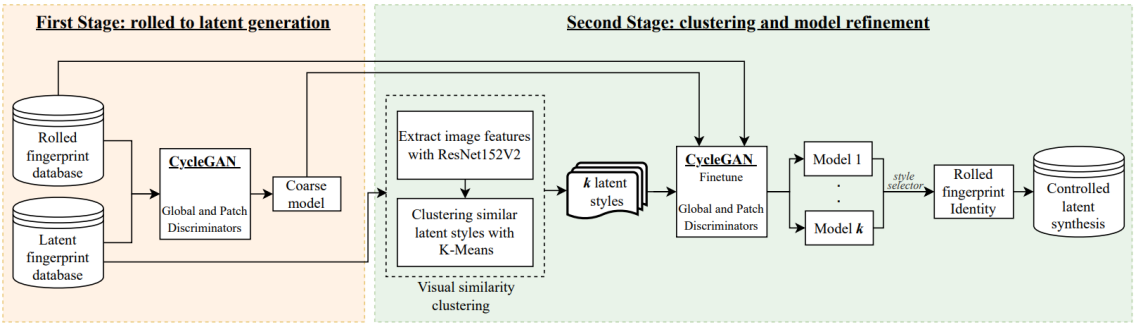


Figure 3.5: Generating latent fingerprints [5]. In the first stage, coarse CycleGAN model is trained to translate between rolled and latent fingerprints, utilizing NIST SD27 [16] dataset for rolled fingerprints, and MSP latent database [40] for latents. In the second stage, pre-trained ResNet152V2 [19] network extracts embeddings from the latent fingerprints. The latent fingerprints are then clustered into k categories using k-means clustering, according to their embeddings. For each of these categories, coarse model is fine-tuned on the fingerprints from the given category, creating k separate models. In the end, these models can be used to synthesize k different styles of latent fingerprints from rolled fingerprints.

[2]https://www.tensorflow.org/api_docs/python/tf/keras/applications/resnet_v2/ResNet152V2

The generated fingerprints are evaluated using NFIQ 2 [3] score, showing that generated fingerprints are realistic. Besides that, DeepPrint [11] model finetuned using these images showed a performance boost. There was no analysis of identity preservation, however the fact that DeepPrint model performance improved after fine-tuning, as well as sample images, seem to confirm that identity is preserved. The disadvantage of this approach is that despite claiming that it can generate multiple impressions of the same finger, the authors only chose to generate 3 of them, and it seems like it would be difficult to generate much more. This is because for each impression, there needs to be separate model fine-tuned, which is not practical in terms of computational power and memory capacity. Furthermore, increasing the number of impressions would decrease the number of training images per impression, which could degrade the quality of images, and variance between impressions.

AugNet

The aim of the work in the paper [39] is to improve latent fingerprint reconstruction. To tackle this issue, they first propose GAN-based data augmentation framework AugNet, which is our object of concern. This framework is trained to generate latent fingerprints from their binarized clean counterparts. Given large scale dataset of clean binarized fingerprints, and small scale dataset of pairs of latent fingerprints and their binarized versions, it can generate latent fingerprints from the clean binarized ones. It also learns to disentangle between fingerprint identity and degradation patterns, and to map a Gaussian distribution into the distribution of these patterns.

The AugNet consists of generator G, encoder E, and discriminators D_1 and D_2. It is trained in 2 stages. In the first stage, the unpaired dataset of clean binarized fingerprints is utilized. The clean binarized fingerprint is passed into the generator, together with a vector z sampled from Gaussian distribution. The generator produces an image, which is then fed into discriminator D_1, together with real latent images. G and D_1 are trained in adversarial manner, with their loss marked as $\mathcal{L}_{GAN1}$. Simultaneously, encoder E is trained to learn the original noise vector z from produced latent image. Output of E is then compared with z using L1 loss, marked as $\mathcal{L}_z$

In the second stage of training, paired dataset of latent fingerprints is used. The latent fingerprint is passed into encoder to extract degradation vector z. This vector is fed together with corresponding binarized latent into the generator, which produces fake latent fingerprint. The generator is again trained together with discriminator D_2 in adversarial manner, marking the loss as $\mathcal{L}_{GAN2}$. Furthermore, there is a L1 loss constraint marked as $\mathcal{L}_{L_1}$ between the produced latent and the original latent. This is because the degradation vector was extracted from the original latent, so the generator should produce the same result. The last loss component is $\mathcal{L}_{KL}$, which is defined as KL divergence between the reconstructed vector z and the Gaussian distribution. It ensures that vector representing the degradation in real data follows Gaussian distribution. The final loss function is then defined as:

$$\mathcal{L}_{AugNet} = \lambda_{GAN1}\mathcal{L}_{GAN1} + \lambda_{GAN2}\mathcal{L}_{GAN2} + \lambda_{KL}\mathcal{L}_{KL} + \lambda_{L_1}\mathcal{L}_{L_1} + \lambda_z\mathcal{L}_z \tag{3.5}$$

where λ_{GAN1}, λ_{GAN2}, λ_{KL}, λ_{L_1}, and λ_z are arbitrarily chosen hyperparameters. The training process is illustrated in Figure 3.6.

This method is able to generate latent fingerprints from the binarized clean ones. It was shown, that using dataset augmented by this method, together with proposed reconstruction framework ReconNet outperformed multiple state-of-the-art approaches. There

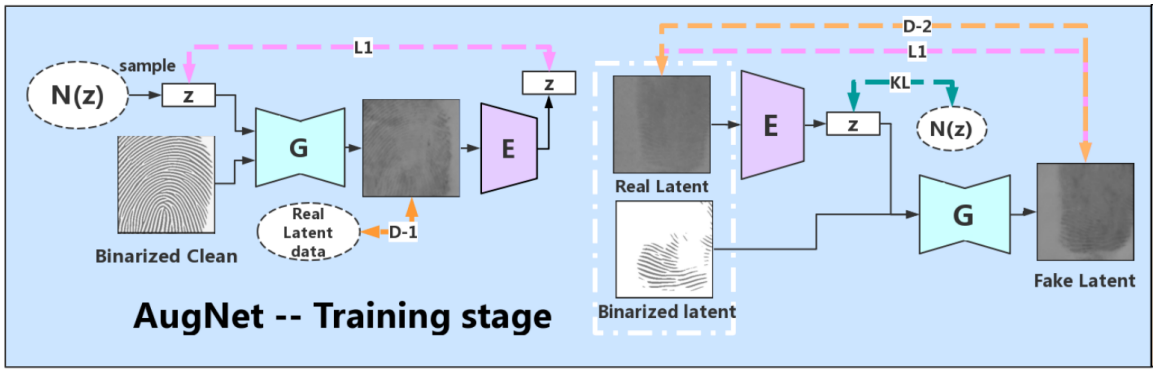


Figure 3.6: AugNet training process [39]. In the first stage, generator G produces fake latent fingerprint from sampled vector z and clean binarized fingerprint. It is trained together with discriminator D_1 in adversarial manner. Encoder E learns to reconstruct degradation vector from the latent image, using L1 loss to compare it to the original z vector. In the second stage, E extracts degradation vector z from real latent image, which is passed to the generator with corresponding binarized latent. Analogically, G generates fake latent, and D_2 are trained in adversarial manner. The generated image should be the same as the input latent (because of the reconstructed vector z), hence the L1 loss. There is also KL divergence loss, so that vector z follows Gaussian distribution for real latent data.

was not any quantitative evaluation of quality of the generated fingerprints, nor any specific claims about identity preservation were made (other than the disentanglement between identity and degradation). Any other existing model was also not trained using the augmented dataset. However, regarding the identity preservation, given that the combination of AugNet and ReconNet showed promising results, as well as visual example from the paper, it is possible to conclude, that identity is preserved. It is also possible to generate a lot of multiple instances of the same fingerprint, just by sampling random vectors. Furthermore, it is possible to control the identity (binarized clean fingerprint) and degradation (random vector) separately. Despite lacking formal evaluation of the model, it is the only one of the mentioned papers, that seems to at least partially fulfill all the criteria (latent fingerprints, identity preservation, multiple impressions per fingerprint, controllable identity and style).

Chapter 4

Implementation

4.1 Proposed approach

For my approach on generating fingerprints, I chose to implement framework inspired by AugNet [39]. This is because, as it was mentioned in section 3.3 in AugNet subsection, it is the only one of studied approaches, that at least partially fulfills all the criteria. The following description is based on the paper and the supplementary file[1] for the paper. However, there are a few key points missing, so those will be experimented with in chapter 5.

Discriminator

Discriminator in this approach follows classic downsampling convolutional architecture. More precisely, PatchGAN [21] architecture is utilized (described in section 2.6). Architecture is outlined in Figure 4.1. Instead of pooling layers, strided convolution is used. Padding is applied to keep height and width of outputs always at powers of 2. This also applies to encoder and generator.

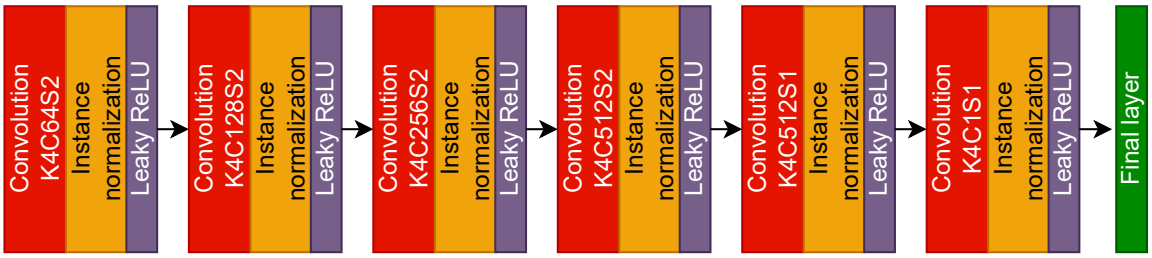


Figure 4.1: Architecture of the discriminator. KxCySz describes a convolutional layer with kernel size x, y channels, and with a stride of z. Final layers (or layers) are subject to experimentation in chapter 5.

Encoder

Encoder, similar to discriminator, is using convolutional downsampling architecture. Its task is to extract fixed length vector from latent fingerprint image. Unlike the discriminator,

[1] https://drive.google.com/open?id=1K3e7NuHTPoAWW2HGSYJDGfbA1086nmhJ

it does not use any special architecture type. Convolutional layers are followed by Global Average Pooling to collapse each feature map into a single scalar. At the end, fully connected layer transforms this vector into a new vector, that represents final output. According to the supplementary file, it is a 16-dimensional vector. Full architecture is shown in Figure 4.2.

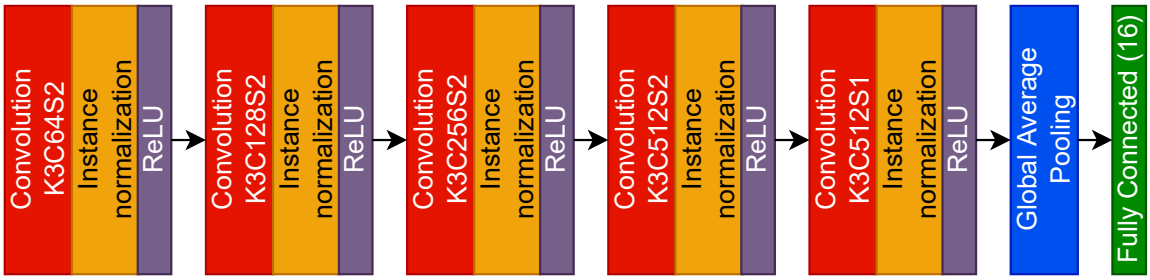


Figure 4.2: Architecture of the encoder. KxCySz describes a convolutional layer with kernel size x, y channels, and with a stride of z. The output size of fully connected layer is 16.

Generator

The generator is the most complex, and also most vaguely defined component of the AugNet. It employs U-Net architecture, which means it is a sequence of downsampling layers, followed by upsampling layers, with a few layers in between, and symmetric connections between downsampling and upsampling layers. The most crucial missing information in the paper is, how is the latent degradation vector incorporated into the image. Based on StyleGAN [23] architecture described in section 2.5, which adapts the style vector into the image using AdaIN [20] block, as well as PrintsGAN [12] authors encoding noise vector into parameters for instance normalization, I decided to use this approach as well. Mapping network from StyleGAN was also adopted, to create intermediate latent vector space. As StyleGAN generator works in upsampling manner, the style vector is only added in the upsampling part of our generator, at each layer. Figure 4.3 shows the full generator architecture.

Training

As mentioned before, 2 datasets are utilized for training. First of them is larger, unpaired dataset of clean fingerprints. Second of them is smaller, paired dataset of latent fingerprints, and their binarized counterparts. As can be seen from Figure 3.6, the architectures for these datasets are different. That effectively divides the training into 2 separate stages. In the paper, there is no mention of how these 2 stages are alternated. From practical point of view, it makes the most sense to alternate between them after each epoch, as the datasets have different sizes. For the exact training procedure, see Algorithm 3.

Implementation details

For implementation of AugNet model, TensorFlow[2] Keras API was utilized. Individual modules were implemented by subclassing *tf.keras.Model* and *tf.keras.Layer*. The only

[2]https://www.tensorflow.org/

methods that required to be implemented for forward pass were *__init__*, *call*, optionally *build*. Due to complexity of training algorithm, no high level Keras training API was used, instead, custom training loop was implemented, using custom *train_step* method. The code was also written to work on multiple GPUs, using *tf.distribute.MirroredStrategy*.

Algorithm 3 AugNet training algorithm. $\mathcal{L}_{AugNet}$ refers to loss function defined by equation 3.5. Only the parameters passed in parentheses are relevant, others are considered to be zero. Sometimes, the same computation is carried out twice, because we do not want to accumulate gradients between both of these computations.

Inputs: Generator G, discriminators D_1, D_2, encoder E, prior noise distribution $p_{\mathbf{z}}(\mathbf{z})$, clean dataset $\mathbf{X}_{clean}$, latent dataset $(\mathbf{X}_{latent}, \mathbf{X}_{binary})$.
Hyperparameters: Number of epochs E.
Objective: Train AugNet model.

for epoch = 1,...,E **do**
  for $\mathbf{x}$ in $\mathbf{X}_{clean}$ **do**
    Sample batch of vectors $\mathbf{z}$ from $p_{\mathbf{z}}(\mathbf{z})$.
    Generate batch of fake data $\mathbf{y} = G(\mathbf{x}, \mathbf{z})$
    Sample batch of real data $\mathbf{r}$ from $\mathbf{X}_{latent}$
    Get discriminator predictions $\mathbf{p} = D_1(concat(\mathbf{r}, \mathbf{y}))$
    Calculate total loss $\mathcal{L}_{AugNet}(\mathbf{p})$ and update weights in D_1.
    Sample batch of vectors $\mathbf{z}$ from $p_{\mathbf{z}}(\mathbf{z})$.
    Generate batch of fake data $\mathbf{y} = G(\mathbf{x}, \mathbf{z})$
    Get discriminator predictions $\mathbf{p} = D_1(\mathbf{y})$
    Calculate reconstructed vectors $\mathbf{z}' = E(\mathbf{y})$
    Calculate total loss $\mathcal{L}_{AugNet}(\mathbf{p}, \mathbf{z}, \mathbf{z}')$ and update weights in G and E.
  end for
  for $\mathbf{x}_{latent}, \mathbf{x}_{binary}$ in $(\mathbf{X}_{latent}, \mathbf{X}_{binary})$ **do**
    Calculate reconstructed vectors $\mathbf{z} = E(\mathbf{x}_{latent})$
    Generate batch of fake data $\mathbf{y} = G(\mathbf{x}_{binary}, \mathbf{z})$
    Get discriminator predictions $\mathbf{p} = D_2(concat(\mathbf{x}_{latent}, \mathbf{y}))$
    Calculate total loss $\mathcal{L}_{AugNet}(\mathbf{p})$ and update weights in D_2.
    Calculate reconstructed vectors $\mathbf{z} = E(\mathbf{x}_{latent})$
    Generate batch of fake data $\mathbf{y} = G(\mathbf{x}_{binary}, \mathbf{z})$
    Get discriminator predictions $\mathbf{p} = D_2(\mathbf{y})$
    Calculate reconstructed vectors $\mathbf{z}' = E(\mathbf{y})$
    Sample batch of vectors $\mathbf{z}$ from $p_{\mathbf{z}}(\mathbf{z})$.
    Calculate total loss $\mathcal{L}_{AugNet}(\mathbf{p}, \mathbf{z}, \mathbf{z}', \mathbf{y}, \mathbf{x}_{latent})$ and update weights in G and E.
  end for
end for

4.2 Data processing

Clean fingerprints need to be binarized for training the model. In this work, Otsu´s thresholding [29] (section 3.2) was used to binarize clean fingerprint images.

For optimal results, all fingerprint images should have the same size. Also, the width and height of the image should be the same. In the AugNet [39] paper, there was no mention

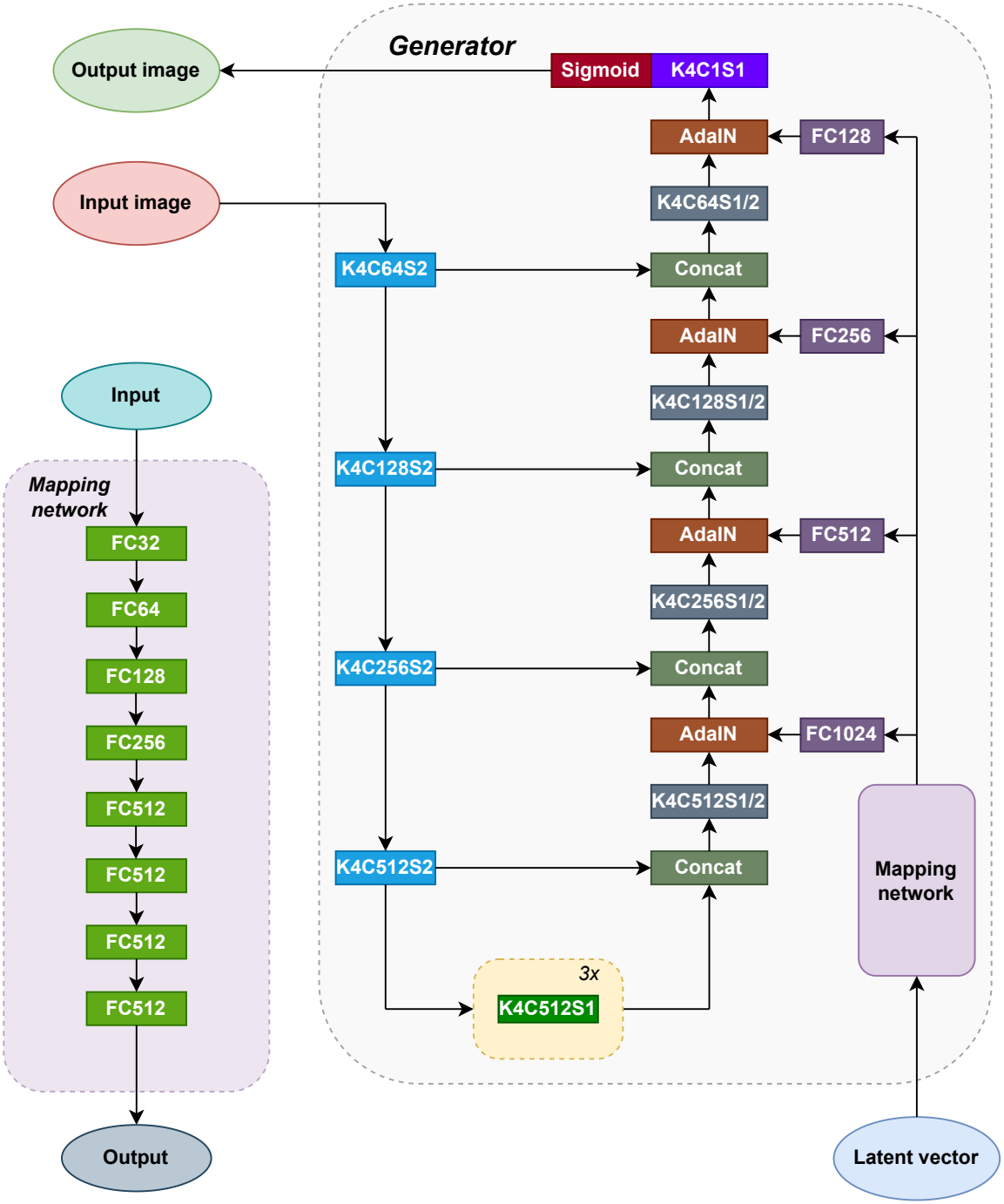


Figure 4.3: Architecture of the generator, depicted on the right side of the image. On the left side, there is an architecture of mapping network component. KxCySz describes a convolutional layer with kernel size x, y channels, and with a stride of z. Upsampling part contains fractional strides (1/2). All of the convolutional layers except the last one are followed by instance normalization and Leaky ReLU. FCx represents fully connected layer with output size of x.

of image dimensions. In the supplementary file[3] they mention „*cropping the central 512 × 512 regions for reconstruction*“, which could imply 512 × 512 image dimensions. Also, in the PrintsGAN [12] paper, they explicitly state generating 512 × 512 grayscale images. Given these circumstances, the same dimensions will be used in this approach. All images were first padded with white pixels, so that the dimensions are the same. If image had shape $H \times W$, after padding it became $max(H, W) \times max(H, W)$. Then, images were resized to 512 × 512 size.

Another common issue when using fingerprints with convolutional networks is image alignment. In the available dataset of latent fingerprints, the images were not aligned. As mentioned in section 5.1, there are some additional information about NIST SD302 [15] dataset provided. This data contains mapping of minutiae points from clean fingerprint to latent, so that latent can be aligned according to clean. These points were used by module *transform*[4] from *scikit-image* library to estimate parameters of projective transformation [25] (section 3.2). Estimation was carried out by *skimage.transform.estimate_transform* method, and for subsequent transformation, *skimage.transform.ProjectiveTransform* class was used. Binarized version of latent was also downsampled, so the minutiae points for the latent image had to be scaled. Figure 4.4 shows example transformation. Complete data processing pipeline is depicted in Figure 4.5. All images were normalized to range $[0; 1]$.

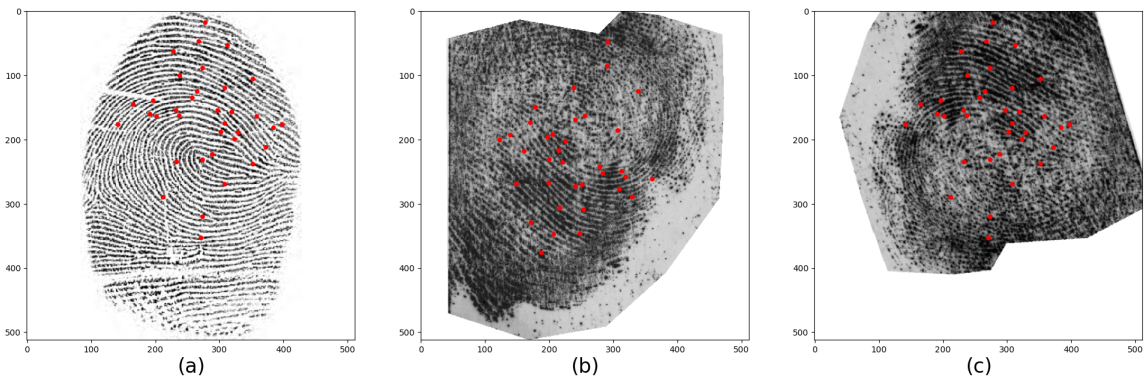


Figure 4.4: Example of latent fingerprint alignment. (a) Clean fingerprint (b) Same latent fingerprint (c) Same latent fingerprint after transformation. Red dots are representing minutiae.

4.3 Experiment setup

As a part of this thesis, large number of model training experiments was conducted. Each training run could use different model architecture, data source, processing pipeline, training algorithm, hyperparameters, etc. There was a need for a robust experiment tracking system. For that, git tags in combination with *DVC*[5] were used. Each git tag corresponded to one experiment, and it contained:

[3] https://drive.google.com/open?id=1K3e7NuHTPoAWW2HGSYJDGfbA1086nmhJ
[4] https://scikit-image.org/docs/stable/api/skimage.transform.html
[5] https://dvc.org/

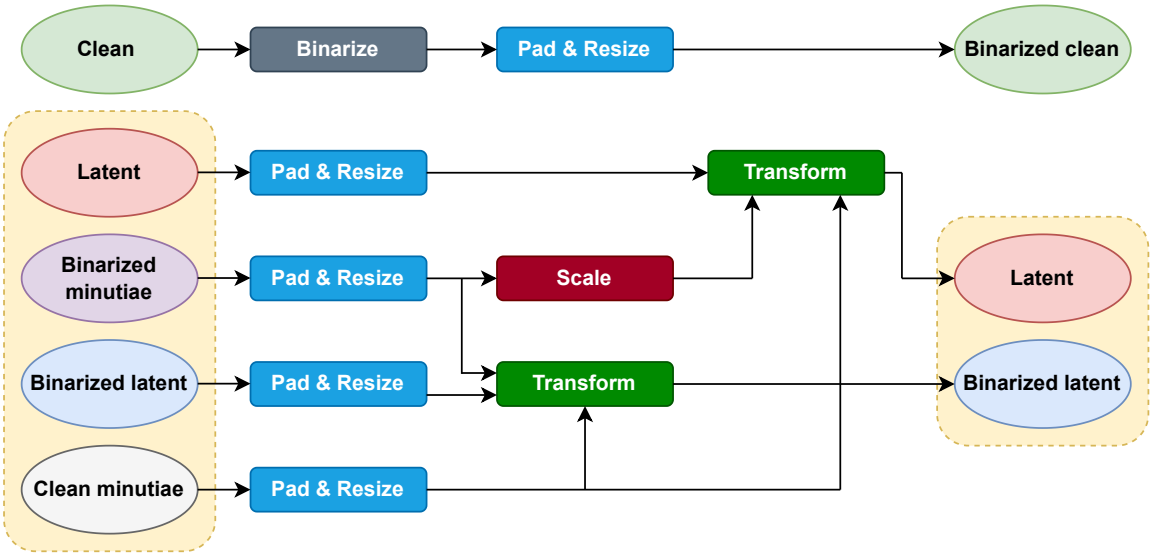


Figure 4.5: Data processing pipeline. All images are padded to square, and then resized to 512 × 512. Clean fingerprints are binarized using Otsu´s thresholding [29] (section 3.2). Binarized latent is transformed using projective transformation [25] (section 3.2), its parameters are estimated from mapping to clean fingerprint minutiae. Latent is transformed the same way, minutiae are just scaled.

1. Python source files (model, training algorithm, data pipeline, evaluation scripts,...)
2. Python configuration file - mainly model/training hyperparameters, data directories,...
3. DVC files

DVC files referred to directories, which stored model weights at different checkpoints, *TensorBoard*[6] logs, and samples of generated images.

For scheduling multiple experiments, as well as running them in parallel, I implemented a custom Python script. The script took *YAML*[7] configuration file as an input. In this file, it was possible to specify different parameters, which were parsed into Python configuration file. It was also possible to set different git commits for each experiment, referring to specific model versions. The file also supported option for specifying, which of the groups of parameters should be taken as a cartesian product, generating separate run configuration for each one of their combinations. Scheduling parameters were also included, such as how many experiments should be run in parallel, how many GPUs should be used, and after what interval should the training be restored after failure.

After parsing the configuration file, the experiment running script generated separate folder for each experiment in subfolder ignored by git and DVC. That way, it was not interfering with debugging and development, as it only checked out specific files to given commits, copied them, and restored them back. Each of the generated folders contained copy of all Python source files, including configuration file. After generating the folders, the running script launched training and evaluation scripts (in correct order). After all of them

[6] https://www.tensorflow.org/tensorboard
[7] https://yaml.org/

finished, I reviewed the results, and decided, which experiments are worth keeping. There was another script for copying selected experiment folder back to main folder, including all Python files, logs, weights, and output image samples. After copy, I decided which checkpoints (weights) are worth keeping, and discarded the others. Finally, there was a script for saving the experiment. This script committed all larger files (weights, logs, output samples) to DVC, and subsequently created a git commit and a tag, with given name and description. It also created cleanup commit after the first one, removing DVC files, so that they are only present in the experiment commits.

Typical experiment workflow is depicted in Figure 4.6. For all of the experiments, 2 to 4 NVIDIA RTX A5000 24GB[8] GPUs were used, depending on availability.

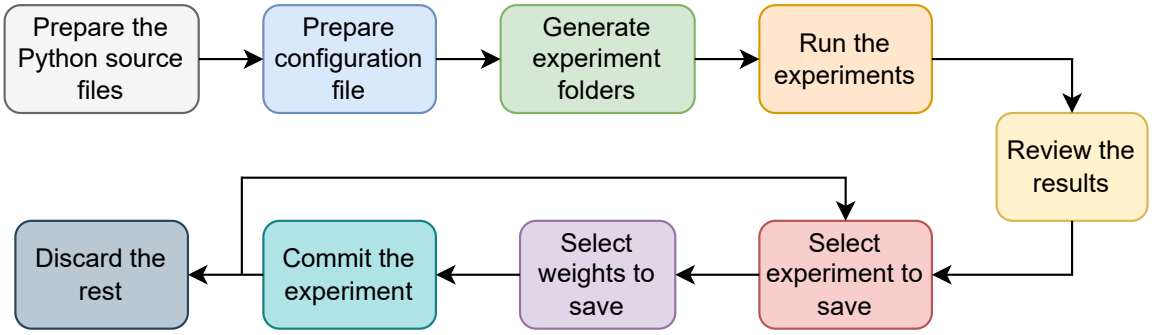


Figure 4.6: Typical experiment workflow.

[8] https://www.nvidia.com/en-us/design-visualization/rtx-a5000/

Chapter 5

Experiments

5.1 Available data

In this section we will be discussing available datasets for training fingerprint generation models.

NIST SD302

NIST SD302 [15] is a large dataset of multiple types of fingerprints. It is split into 9 sets of fingerprints marked a to i, each of them having common type, device, or scenario. For this work, only sets from a to e are available, so from this point on, all the information is relevant to this subset. Counts of fingerprints for each different fingerprint type are shown in Table 5.1. They were taken from 200 unique subjects.

Table 5.1: Summary of NIST SD 302 dataset. Table shows count of fingerprints for each type.

Type	**Count**
Rolled	13629
Plain	3188
Slap	3472
Palm	9732
Latent	9990
Total	**40011**

Besides the fingerprints themselves, there was also 158153 image pairs, that matched fingerprints together to provide identity information. However, there was no matching between latent and non-latent fingerprints. Latent fingerprints in this dataset are taken in different environments. Their types are listed in Table 5.2.

Additional information for NIST SD302

In order to train AugNet model as is stated in section 3.3, 2 datasets are needed. One is unpaired dataset of binarized clean fingerprints, which is not that difficult to obtain. However, acquiring a paired dataset of latent images and their binarized versions is a more challenging task. Thanks to Innovatrics company, I was able to get binarized versions of a subset of latent fingerprints from NIST SD302 dataset. In total, there are 3720 binarized

Table 5.2: Listing of different environments of latent fingerprints and their codes [14].

1A	Peering Into Window	**4E**	Low-quality White Envelope
1B	Fist Banging on Glass	**4F**	Greeting Card and Envelope
1C	Fingertip Window Slide	**4G**	Manila Envelope
1D	Get-away Palm on Glass	**5A**	Photo Paper
1E	"OK" Sign on Glass	**5B**	Glossy Magazine
1F	Counter Vault on Glass	**5C**	U.S. Currency
1G	Cylinder Grab	**6A**	Stamp
1H	Impatient Tapping on Glass	**6B**	Address Label
2A	Samsung Galaxy S5	**6C**	Clear Packing Tape
2B	Apple iPhone 5s	**6D**	Black Electrical Tape
3	Check	**6E**	Duct Tape
4A	Lined Paper	**7A**	Circuit Board
4B	Low-quality Copy Paper	**7B**	CD/DVD
4C	High-quality Copy Paper	**7C**	Clear Plastic Bag
4D	Yellow Lined Paper	**7D**	Black Plastic Bag

fingerprints. For some of them, there is also an information about matched clean fingerprint from NIST SD302 dataset. This matching was done using commercial software, and it includes name of the matched clean image, matching score, as well as mapping of minutiae points between clean and binarized fingerprint. This mapping is very useful, as it is possible to estimate parameters of projective transformation of latent fingerprints, in order to align them according to clean fingerprints. In total, this information was provided for 1967 of the binarized latent fingerprints. Distribution of the matching scores is shown in Figure 5.1. I decided to only use the matches with score 100 or above, leaving 1455 fingerprints in total.

MOLF DB

MOLF DB (Multisensor Optical And Latent Fingerprint Database) [32] is a database containing multiple fingerprint types. It provides clean fingerprints from 100 subjects scanned by different devices, and also their latent fingerprints. Clean fingerprints are mapped to latent ones, which provides good training data for models of our interest. Table 5.3 shows contents of the dataset. Additional binarized versions of majority of these images were also provided by Innovatrics company.

Table 5.3: Contents of MOLF DB dataset [32].

Subset	**Fingerprint type**	**Number of images**
DB1	Plain	4000
DB2	Plain	4000
DB3	Slap	1200
DB3_A	Plain	4000
DB4	Latent	4400
DB5	Simultaneous latent	1600

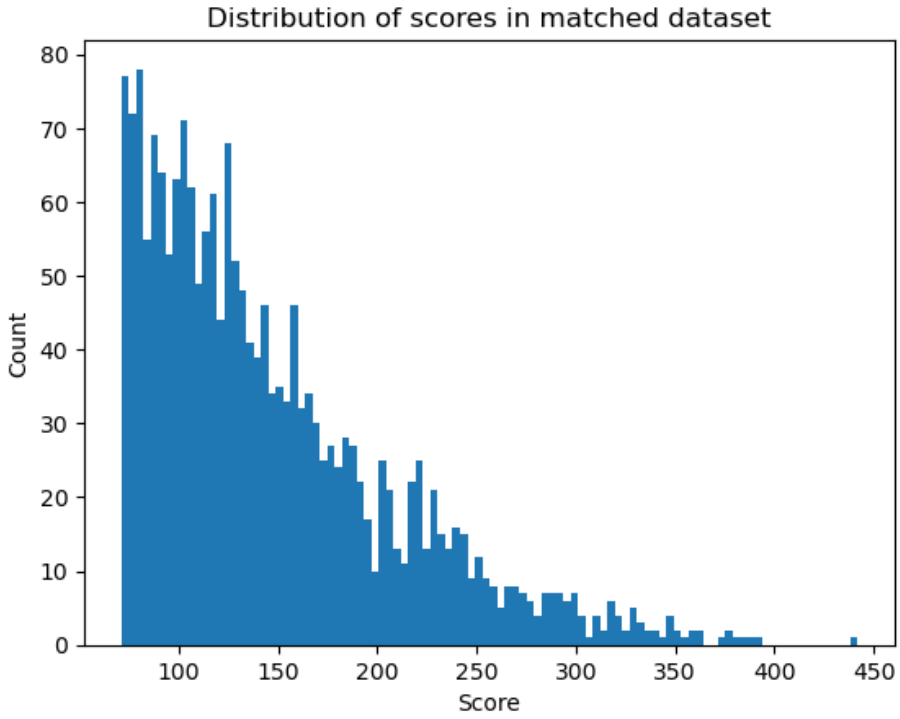


Figure 5.1: Distribution of matching scores in data provided by Innovatrics company.

MSLFD

IIITD MSLFD (IIITD Multi-Surface Latent Fingerprint Database) [31] is a database of latent fingerprints from different surfaces. It contains 551 fingerprints from 51 subjects acquired from 8 individual surfaces. The list of surfaces is as follows:

1. Ceramic plate
2. Ceramic mug
3. Transparent glass
4. Steel glass
5. Compact disc
6. Compact disc mailer
7. Paperback book cover
8. Hardbound book cover

This dataset could provide valuable diversity for latent fingerprint generation. Frankly, there are no corresponding clean fingerprints, and the latents are also not aligned, which complicates its use in generative model training.

Datasets used in the experiments

For the dataset of binarized clean fingerprints, only MOLF DB1 [32] binarized using methods in section 4.1 was used in all experiments. For the latent dataset, 3 separate datasets were used across experiments. They are summarized in Table 5.4. From this point on, I will be referring to these datasets as they are described in this table.

Table 5.4: Latent datasets used in experiments.

Dataset	Image count
NIST SD302 latent	9990
NIST SD302 latent subset	1455
MOLF DB4	4400

5.2 First experiment

Description

The model described in section 4.1 was trained using the subset of available data. For the unpaired dataset of clean fingerprints, MOLF DB1 [32] (section 5.1) was transformed to binary fingerprint set using methods described in section 4.2. For the paired dataset, aligned subset of NIST SD302 [15] (section 5.1) was utilized, together with additional information. This information contains binarized versions of latent fingerprints, as well as minutiae mappings to align latent fingerprints, as was described on previously mentioned sections. For a loss function, LSGAN [26] (section 2.7) was utilized, however, final sigmoid layer was kept in the model. Loss function weights were kept the same, as in the paper [39]. No additional regularization techniques were applied. Adam [24] was used as an optimizer with recommended values from the paper. Summary of dataset sizes, and various model and training hyperparameters is provided in Table 5.5.

Table 5.5: Summary of data, model, and training hyperparameters. Patch size refers to PatchGAN [21] (section 2.6) architecture. Latent vector size is the size of the sampled noise vector. Loss constants represent weights in the original AugNet loss function (equation 3.5).

Clean fingerprint dataset size	4000
Latent fingerprint dataset size	1455
Image dimensions	512 × 512
Discriminator patch size	64 × 64
Latent vector size	16
Leaky ReLU α	0.3
λ_{GAN1}	1
λ_{GAN2}	1
λ_{KL}	0.01
λ_{L_1}	10
λ_z	0.1
Adam learning rate	1e-4
Adam β_1	0.5
Adam β_2	0.999
Batch size	16
Epochs	100

Results

Figure 5.2 summarizes the output of the generator after different stages of training. It is evident that trained model did not produce desired outputs. Trained generator produces

repetitive tiling pattern in the image. The pattern seems to get more noisy further down the training. Furthermore, it seems that the model gradually learns to generate noise-invariant output, which is undesired, since we want to generate different fingerprint impressions based on the noise (latent) vector. However, the untrained generator seems to combine fingerprint identity with style from latent vector surprisingly well. This might indicate, that the generator architecture suits this problem nicely. Also, the fingerprint identity preservation, although not evaluated rigorously, looks to be working substantially well, even in the noisy outputs in the last row. Main fingerprint ridge structure, or at least parts of it, can be seen in all of the outputs. In some cases, the ridge structure is visible more clearly after epoch 10, than with the untrained model.

5.3 Conditional GAN training

As is described in section 4.1, training of AugNet [39] model consisted of 2 alternating parts. In the first one, conditional GAN model was trained to generate realistic latent fingerprints, given clean binarized template. It was trained together with the encoder, which would learn to extract latent vector that was used to generate the image. In the second part, the trained encoder was used to extract the latent vector from the real latent fingerprint, and the generator was trained to reconstruct this latent fingerprint from the binarized version and extracted latent vector. The other mentioned fact was that it is unclear, how were these 2 training stages alternated. Few of the first experiments were dedicated to training the model in this way, alternating these 2 stages after each epoch (Algorithm 3). They were unsuccessful, and failed to generate any plausible fingerprint images, as can be seen in section 5.2. Training was also highly unstable, and the model failed to converge. Because of these reasons, I decided to spend most of the time experimenting with the first training stage (conditional GAN training without the encoder). The first reason for that is because it would be necessary for it to generate plausible fingerprint images for the second stage to work, and secondly, there was a lot more room for improvement. The following sections will thus be dedicated to training conditional GAN model, unless stated otherwise. The generic procedure for this training is depicted in Algorithm 4.

5.4 Discriminator improvements

PatchGAN architecture implementation

As was mentioned in section 4.1, discriminator network follows PatchGAN [21] architecture. In section 2.6, 2 different implementations of this architecture are compared. For the first few experiments, I used the first implementation, the one that splits input image into non-overlapping image patches. After that, I decided to switch to the second more simple variant, mainly because it was used effectively in many scenarios.

Activation and loss function

The second important aspect of discriminator architecture is what to do with the output after convolutional layers. As is shown in Figure 2.5, for 512×512 image input, the output is a matrix of shape 32×32. The first option appearing in some of the implementations is to treat each element of that matrix as a response to respective image patch. Thus for each of the elements, sigmoid function is applied to convert it into probability, and then

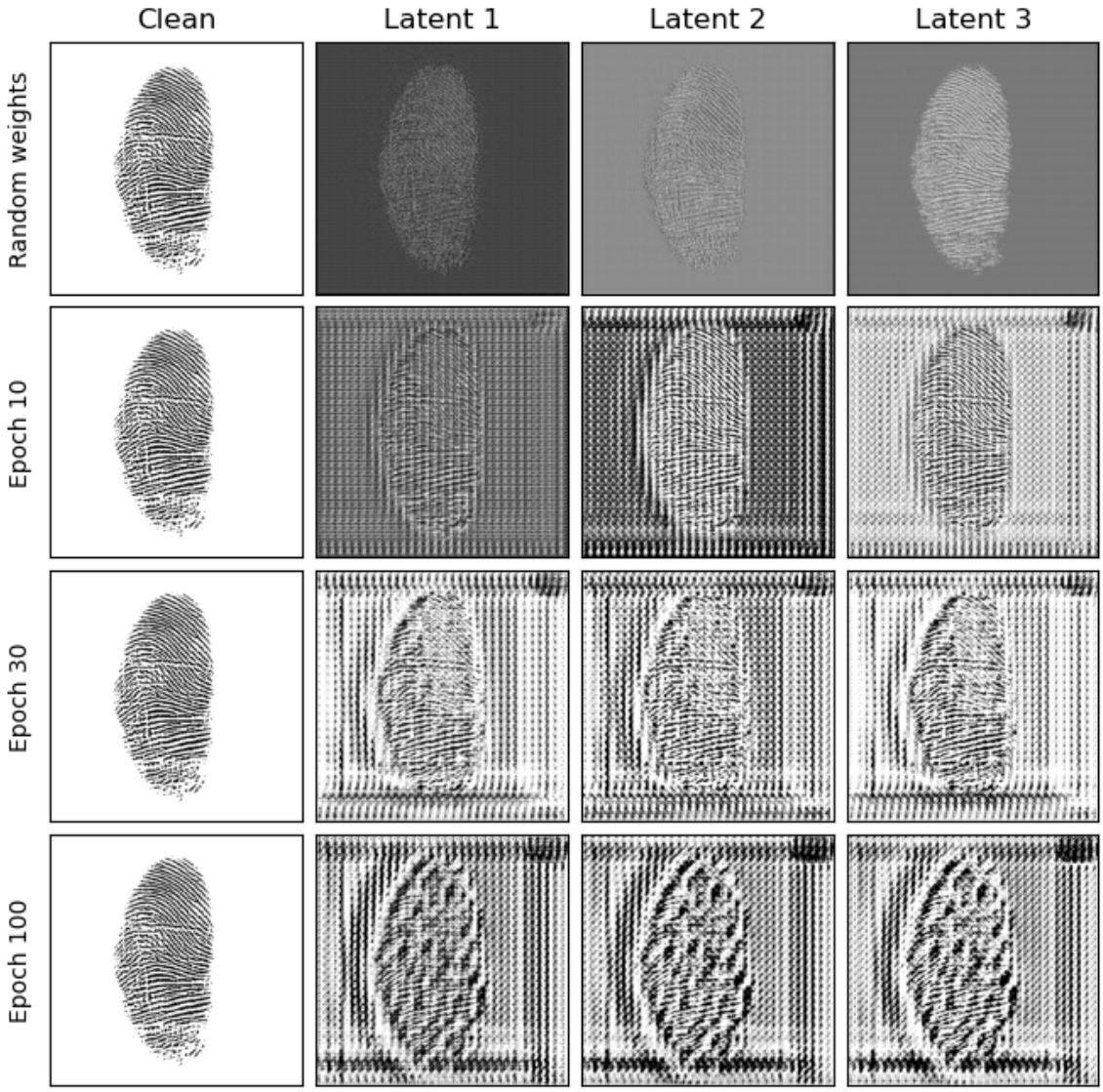


Figure 5.2: Generator output after different stages of training. First column shows the clean fingerprint passed into the generator. Next 3 columns show generator outputs for 3 random latent vectors sampled from normal distribution. First row corresponds to untrained model with randomly initialized weights. Each of the next rows represent generator output after being trained for 10, 30, and 100 epochs, respectively.

binary cross-entropy is applied element-wise, giving one loss value for each matrix element. In other words, the ground truth is not a single scalar label, rather 32×32 matrix of same labels (meaning that each of the image patches should be real/fake). This method, however, did not succeed in generating realistic fingerprints. The second method is to reduce the matrix to a single scalar by calculating mean, and use that as a final discriminator output, dropping the sigmoid and using LSGAN loss function. This method yielded first larger improvement in resulting image quality, together with significantly increasing the number of epochs.

Algorithm 4 Conditional GAN training algorithm

Inputs: Generator G, discriminator D, prior noise distribution $p_{\mathbf{z}}(\mathbf{z})$, clean dataset $\mathbf{X}_{clean}$, latent dataset $\mathbf{X}_{latent}$.
Hyperparameters: Number of epochs E.
Objective: Train conditional GAN model.

for epoch = 1,...,E **do**
  for $\mathbf{x}$ in $\mathbf{X}_{clean}$ **do**
    Sample batch of vectors $\mathbf{z}$ from $p_{\mathbf{z}}(\mathbf{z})$.
    Generate batch of fake data $\mathbf{y} = G(\mathbf{x}, \mathbf{z})$
    Sample batch of real data $\mathbf{r}$ from $\mathbf{X}_{latent}$
    Get discriminator predictions $\mathbf{p} = D(concat(\mathbf{r}, \mathbf{y}))$
    Calculate loss $\mathcal{L}_D(\mathbf{p})$ and update weights in D.
    Sample batch of vectors $\mathbf{z}$ from $p_{\mathbf{z}}(\mathbf{z})$.
    Generate batch of fake data $\mathbf{y} = G(\mathbf{x}, \mathbf{z})$
    Get discriminator predictions $\mathbf{p} = D(\mathbf{y})$
    Calculate loss $\mathcal{L}_G(\mathbf{p})$ and update weights in G.
  end for
end for

The difference can be seen in Figure 5.3. First row is output of the generator after being trained for 10 epochs with learning rate of $2e-6$, where there was element-wise sigmoid applied to discriminator outputs, and then LSGAN loss function was calculated, also element-wise. The second row is generator output after 40 epochs of training with learning rate of $1e-4$, with no activation function at the end of discriminator. The responses were simply averaged together, and then LSGAN loss function was calculated. The first model was trained with very low learning rate for a low amount of iterations, because after more epochs, it quickly collapsed to generating purely black or white images.

Despite these improvements, the training was still quite unstable, and very sensitive to learning rate changes. For example, training was successful for learning rate of $1e-4$, but diverged very quickly for learning rate $2e-4$.

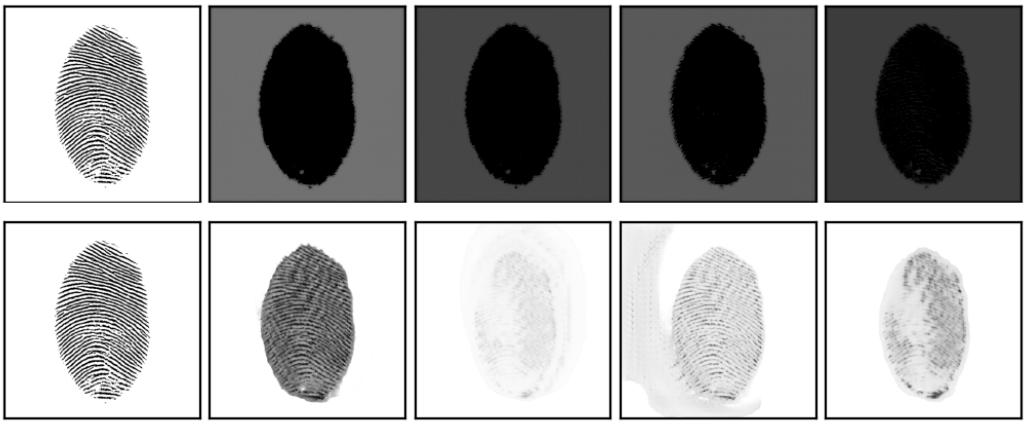


Figure 5.3: Comparison of generator outputs. First row - LSGAN loss function with element-wise sigmoid, 10 epochs, learning rate $2e-6$. Second row - LSGAN loss without sigmoid, averaged at the end, 40 epochs, learning rate $1e-4$.

5.5 Mode collapse problem

Attempt at training encoder

After reaching reasonable image quality in conditional GAN training, I decided move on to second stage of AugNet training, described in section 4.1. Although the original plan was to train first and second stage at the same time, I decided to try this approach:

1. Train the conditional GAN.
2. Generate dataset from the trained generator, containing generated images and random vectors, from which they were generated.
3. Train the encoder on this dataset.
4. Train the generator together with the encoder.
5. Run the second training stage.

The generated dataset contained 10000 fingerprint images, synthesized from 500 binarized fingerprints (different identities), and 20 different style vectors for each of these identities. The dataset was split to 8000 training and 2000 validation samples. The encoder was trained to predict the original style vector from the image, using $\mathcal{L}_1$ loss. However, the encoder failed to converge, as it is shown in Figure 5.4. The $\mathcal{L}_1$ training loss was too high even after 40 epochs, considering that the vectors were sampled from standard normal distribution. The loss was calculated as an average over all vector elements, so it can be understood as an average absolute difference between each predicted vector element and corresponding ground truth. What is even more important is that the loss almost did not change on validation dataset, which indicates overfitting. This failure can be attributed to mode collapse in the images from the generator, which is depicted in Figure 5.5. The generator can only generate few visually distinct latent styles from the entire input latent space, forcing the encoder to randomly guess the original style vector. This is because many different style vectors lead to one almost identical image, so the encoder would just overfit to generated dataset, and would be of no use for the next training stage. This is why I decided to focus on mode collapse problem in the conditional GAN training.

Mode collapse reduction attempts

There were multiple attempts to reduce mode collapse problem. They are summarized in this subsection.

1. Entire NIST SD302 latent dataset: Up to this point, the latent training dataset used was the aligned subset containing 1455 images (see Table 5.4). I tried to run the same network configuration, but with the entire NIST SD302 latent dataset, containing 9990 images. This is because the images in the latent subset were less challenging, given that they were able to be extracted using already working methods. My hopes were that if discriminator sees more diverse images, the generator would be forced to generate more diverse image styles. Unfortunately, that was not the case. In addition, it produced images with even lower quality than before.

2. Activations in mapping network: At this point I realized, that generator architecture shown in Figure 4.3 has one major flaw: there were no activation layers in mapping network. This meant that the mapping network collapses to a single linear layer and is not

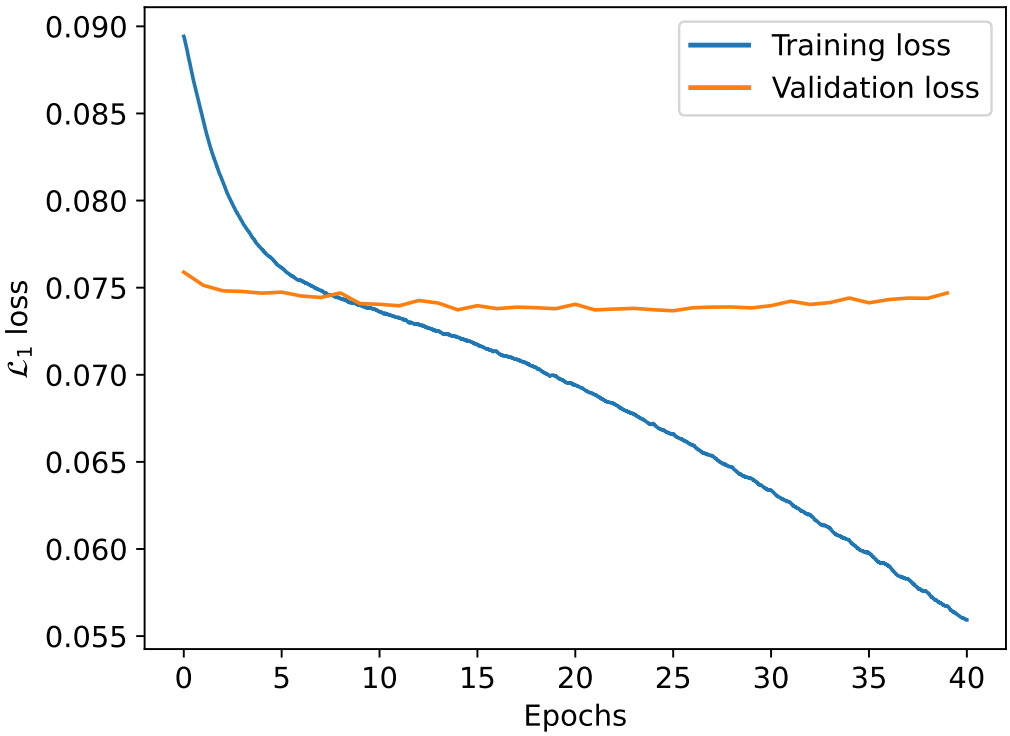


Figure 5.4: Encoder training and validation loss. The $\mathcal{L}_1$ loss was calculated as an average over all vector elements, so it can be understood as an average absolute difference between each predicted vector element and corresponding ground truth.

able to learn any useful transformations. Therefore I decided to add LeakyReLU ($\alpha = 0.3$) activation after each layer. There was, however, no significant change in the mode collapse, nor in the image quality.

3. StyleGAN inspirations: Besides the first change, I tried to use the mapping network more similar to the StyleGAN [23] paper. I changed LeakyReLU α to 0.2, as well as applied so-called *truncation trick* mentioned in the paper. The truncation trick is used to provide higher quality images in tradeoff for lower image variation by restricting latent space in some way. Although decreasing image variation is the opposite of reducing mode collapse, I hoped it could stabilize the training in some way, and help this problem. After transforming input vector to intermediate latent space, the truncation is applied as follows:

$$\mathbf{w}_{new} = \mathbf{w}_{avg} + \psi(\mathbf{w} - \mathbf{w}_{avg}) \tag{5.1}$$

where $\mathbf{w}$ is the original vector, $\mathbf{w}_{new}$ is the resulting truncated vector, and $\mathbf{w}_{avg}$ is the moving average, and ψ is a constant hyperparameter. The moving average is updated after each network iteration in this way:

$$\mathbf{w}_{avg\_new} = \mathbf{w}_{batch} + \beta(\mathbf{w}_{avg\_old} - \mathbf{w}_{batch}) \tag{5.2}$$

where $\mathbf{w}_{batch}$ is the average of latent vectors over the batch, $\mathbf{w}_{avg\_old}$ is the old average, $\mathbf{w}_{avg\_new}$ is the new average, and β is again a constant hyperparameter. Note that all of these calculations are done element-wise, i.e. for each of the 512 latent vector elements. The moving average is also a 512 dimensional vector. Details of this calculations are taken from

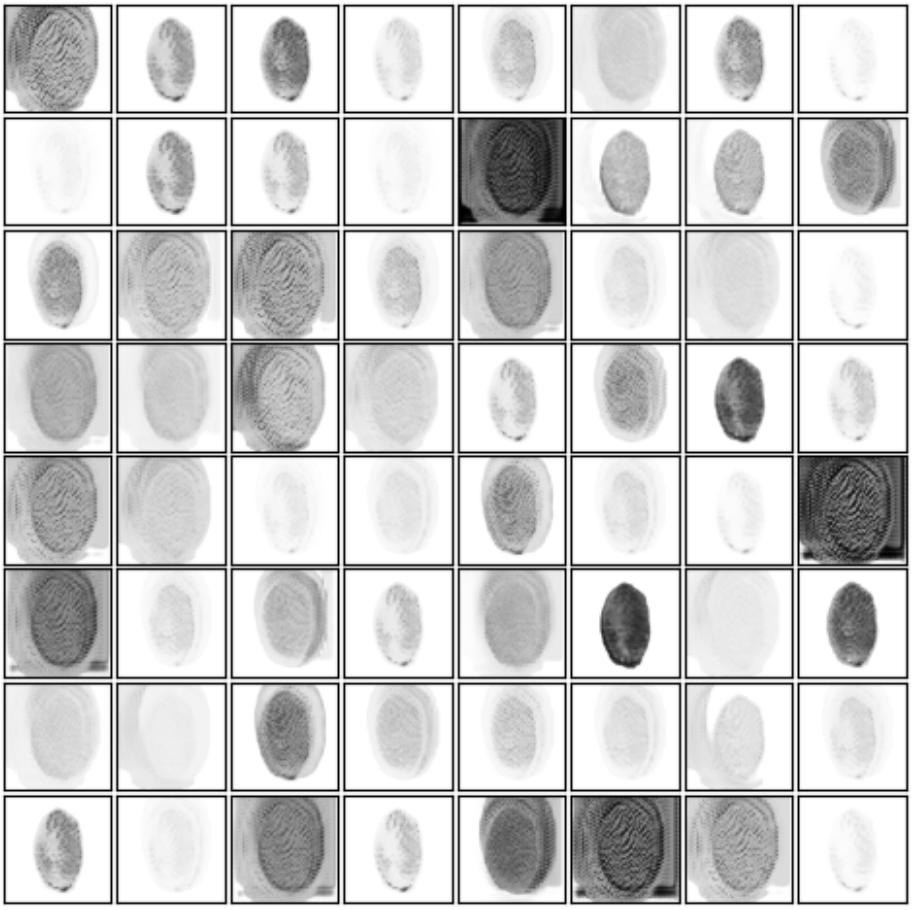


Figure 5.5: Visualization of mode collapse problem. Although no two images are exactly identical, only a few distinct styles can be seen in the image, and variations within a certain style are very small.

official TensorFlow implementation[1], as well as values for the hyperparameters ($\psi = 0.7$, $\beta = 0.995$). The last improvement taken from StyleGAN was reducing learning rate for the mapping network. According to the paper, it increased training stability. The parameter λ for the learning rate was set to 0.01.

I experimented with all 6 combinations of $\lambda = 0.1, 0.01$ and learning rate $= 5e-5, 1e-4, 2e-4$, training each combination for 40 epochs. Despite that, I was not able to get any significant reduction of mode collapse, nor any increase in image quality.

4. Multiple discriminators: The next attempt was use multiple discriminators for the training. This was partially inspired by paper [10]. The point of this approach is to let each discriminator be an *expert* on some subset of the dataset, meaning that it will classify only some subset of the input dataset. In the paper, they specify the number of discriminators, and let the network itself learn the best possible division of the dataset to individual discriminators. I chose a different method. First, I extracted the feature vectors for the entire dataset using pre-trained ResNet152V2[2] [19] network, as it was also done in paper [5]. Then, I used K-Means clustering to divide the features into $k = 3$ clusters.

[1] https://github.com/NVlabs/stylegan

[2] https://www.tensorflow.org/api_docs/python/tf/keras/applications/resnet_v2/ResNet152V2

Training images were split into these clusters based on features. For each cluster, there was a separate discriminator. However, as was also mentioned in the paper [10], all layers of the discriminator were the same, except for the last one. For each of the k clusters, there was a separate convolutional layer, each with its own weights. Then, for the classification of generator outputs, the discriminator with the lowest prediction value is selected. Given that fake images are assigned label *0* and real images label *1*, the discriminator with the lowest response can be considered and expert on that specific prediction, as it is most sure, that generated image is fake. The exact training procedure is depicted in Algorithm 5.

Algorithm 5 Multiple discriminator conditional GAN training algorithm

Inputs: Generator G, discriminators D, prior noise distribution $p_{\mathbf{z}}(\mathbf{z})$, clean dataset $\mathbf{X}_{clean}$, latent dataset $\mathbf{X}_{latent}$.
Hyperparameters: Number of epochs E.
Objective: Train conditional GAN model with multiple discriminators.

for epoch = 1,...,E **do**
  for $\mathbf{x}$ in $\mathbf{X}_{clean}$ **do**
    Sample batch of vectors $\mathbf{z}$ from $p_{\mathbf{z}}(\mathbf{z})$.
    Generate batch of fake data $\mathbf{y} = G(\mathbf{x}, \mathbf{z})$
    Sample batch of real data $\mathbf{r}$ and their respective clusters $\mathbf{c}$ from $\mathbf{X}_{latent}$
    Get discriminator predictions for real data $\mathbf{p'_{real}} = D(\mathbf{r})$
    Select correct predictions $\mathbf{p_{real}}$ based on clusters $\mathbf{c}$ from $\mathbf{p'_{real}}$.
    Get discriminator predictions for fake data $\mathbf{p'_{fake}} = D(\mathbf{y})$
    Select minimum predictions $\mathbf{p_{fake}}$ from $\mathbf{p'_{fake}}$.
    Calculate loss $\mathcal{L}_D(\mathbf{p_{real}}, \mathbf{p_{fake}})$ and update weights in D.
    Sample batch of vectors $\mathbf{z}$ from $p_{\mathbf{z}}(\mathbf{z})$.
    Generate batch of fake data $\mathbf{y} = G(\mathbf{x}, \mathbf{z})$
    Get discriminator predictions $\mathbf{p'_{fake}} = D(\mathbf{y})$
    Select minimum predictions $\mathbf{p_{fake}}$ from $\mathbf{p'_{fake}}$.
    Calculate loss $\mathcal{L}_G(\mathbf{p_{fake}})$ and update weights in G.
  end for
end for

There were 4 separate training runs with different learning rates, $lr \in \{5e{-}5, 1e{-}4, 2e{-}4, 5e-4\}$, each trained for 40 epochs. However, none of these runs provided useful results, and the multiple discriminators setup had the opposite effect, making image diversity even worse.

5. MOLF DB4 dataset: After unsuccessful attempts on reducing mode collapse, I decided to change the latent dataset to MOLF DB4 [32], as it was used in the original AugNet [39] paper. Besides changing dataset itself, I also tried changing latent vector size from 16 to 32, as well as normalizing images into range $[-1; 1]$ instead of $[0; 1]$. There were 3 separate training runs for each of these configurations, each with different learning rate $lr \in \{5e-5, 1e-4, 2e-4\}$. Each training run consisted of 40 epochs. As with the previous attempts, this one was also unsuccessful, providing no improvement in mode collapse reduction, nor image quality.

Unfortunately, none of these methods were successful in reducing mode collapse. This problem was eventually solved, using changes described in following sections.

5.6 WGAN-GP

After multiple failed attempts to reduce mode collapse, I decided to change loss objective altogether. I used WGAN-GP [18] training procedure, described in section 2.8. The training procedure itself did not help to generate better images at first. There were multiple experiments with this training algorithm, each with some incremental changes. Up to this point, training loop of all models consisted of iterating over dataset of clean binarized fingerprints, and sampling random images from latent fingerprint dataset for the discriminator. After reading WGAN-GP [18] paper, I decided it would be better to iterate over latent dataset instead, and sample random images from the clean dataset. Since the objective is to generate latent fingerprints that are as close to the real distribution as possible, it makes more sense that the discriminator (or so-called critic in this case) sees all images from latent dataset at each epoch. This way it is able to approximate the Wasserstein distance better. Besides of choosing random binarized fingerprints, I also experimented with keeping the binarized fingerprint constant for the whole training. These experiments will be described in the following sections.

Despite the fact that the change of objective to WGAN-GP itself did not lead to any significant image quality improvement, the training loss was more stable than in case of LSGAN. Because of this reason, I decided to use this training objective in the majority of following experiments. The specific training procedure for fingerprint generation model is described in Algorithm 6.

Algorithm 6 Conditional WGAN-GP training algorithm

Inputs: Generator G, discriminator D, prior noise distribution $p_{\mathbf{z}}(\mathbf{z})$, clean dataset $\mathbf{X}_{clean}$, latent dataset $\mathbf{X}_{latent}$.
Hyperparameters: Number of epochs E, gradient penalty weight λ.
Objective: Train conditional WGAN-GP model.

for epoch = 1,...,E **do**
  while data in $\mathbf{X}_{latent}$ **do**
    Sample batch of clean data $\mathbf{c}$ from $\mathbf{X}_{clean}$
    for $t = 1, ..., n_{critic}$ **do**
      Get next batch of latent data $\mathbf{r}$ from $\mathbf{X}_{latent}$
      Sample batch of vectors $\mathbf{z} \sim p_{\mathbf{z}}(\mathbf{z})$
      Sample batch of random numbers $\epsilon \sim U[0, 1]$
      Generate batch of fake data $\tilde{\mathbf{x}} = G(\mathbf{c}, \mathbf{z})$
      Calculate interpolation $\hat{\mathbf{x}} = \epsilon\mathbf{x} + (1 - \epsilon)\tilde{\mathbf{x}}$
      Calculate gradient $g = \nabla_{\hat{\mathbf{x}}} D_w(\hat{\mathbf{x}})$
      Calculate gradient penalty $g_p = (\|g\| - 1)^2$
      Calculate loss $\mathcal{L}_D = D(\tilde{\mathbf{x}}) - D(\mathbf{x}) + \lambda g_p$
      Calculate gradients of $\mathcal{L}_D$ w.r.t. weights in D and perform update step.
    end for
    Sample batch of vectors $\mathbf{z} \sim p_{\mathbf{z}}(\mathbf{z})$.
    Calculate loss $\mathcal{L}_G = -D(G(\mathbf{c}, \mathbf{z}))$
    Calculate gradients of $\mathcal{L}_G$ w.r.t. weights in G and perform update step.
  end while
end for

5.7 Generator architecture changes

Despite experimenting with multiple alternatives of training algorithm, data, loss objectives, and network architectures, the mode collapse problem was still not solved. So I decided to take a closer look at the generator architecture (Figure 4.3). In this architecture, the fingerprint style is synthesized from input latent vector by AdaIN [20] blocks, which is simply an instance normalization followed by scale and shift, applied individually for each feature map. Unlike classic instance normalization, the scale and shift parameters are not learned, they are taken from the intermediate latent vector. The changes I experimented with were:

1. Different activations in mapping network (ReLU, LeakyReLU).
2. Making instance normalization in convolutional layers non-trainable.
3. Adding AdaIN blocks after each layer, not just the upsampling part.
4. Removing mapping network.
5. Changing latent vector size from 16 to 128.

Most of these changes had little to no effect on generator output. After that, I decided to modify the convolutional layers instead. As it is mentioned in description of Figure 4.3, each convolutional layer contains the convolution operation itself, followed by instance normalization, and LeakyReLU activation function. Only after that, the AdaIN operation is applied. The fact that AdaIN operation is only applied after instance normalization and activation probably hindered generator´s performance. So I decided to change this architecture altogether, removing the AdaIN from outside of convolutional layer, and replacing the former learnable instance normalization with it. The updated architecture is shown in Figure 5.6. Note that there is also no mapping network. This change was one of the most influential, as it improved image quality, and drastically reduced mode collapse problem. These improvements will be discussed in section 5.9.

5.8 Evaluation methods

Methods description

Given the task of conditional latent fingerprint generation, the objective is to generate realistic latent fingerprints, while preserving the identity of the original binarized template. The evaluation of this task can be quite complicated, since if we want to generate more challenging latent fingerprints, it will be hard to verify identity preservation. On the other hand, if the generated fingerprints can be easily matched to their source, it means that there is not much added value, since the generated distortions are not that challenging. That is why there needs to be some form of trade-off between challenging latent fingerprints, and ability to preserve their identity.

There are 3 studies [33, 5, 39] worth mentioning when it comes to evaluation of quality of generated latent fingerprints using GAN models. The metric called NFIQ2 score [3] is used in the first [33] and the second [5] to asses the quality of generated fingerprints. The metric is used mainly for determining the quality of optical sensor fingerprint images, and assigns the score from 0 (worst quality) to 100 (best quality) to each image. It is not meant to be used for latent fingerprints. Despite that, these studies use the metric to

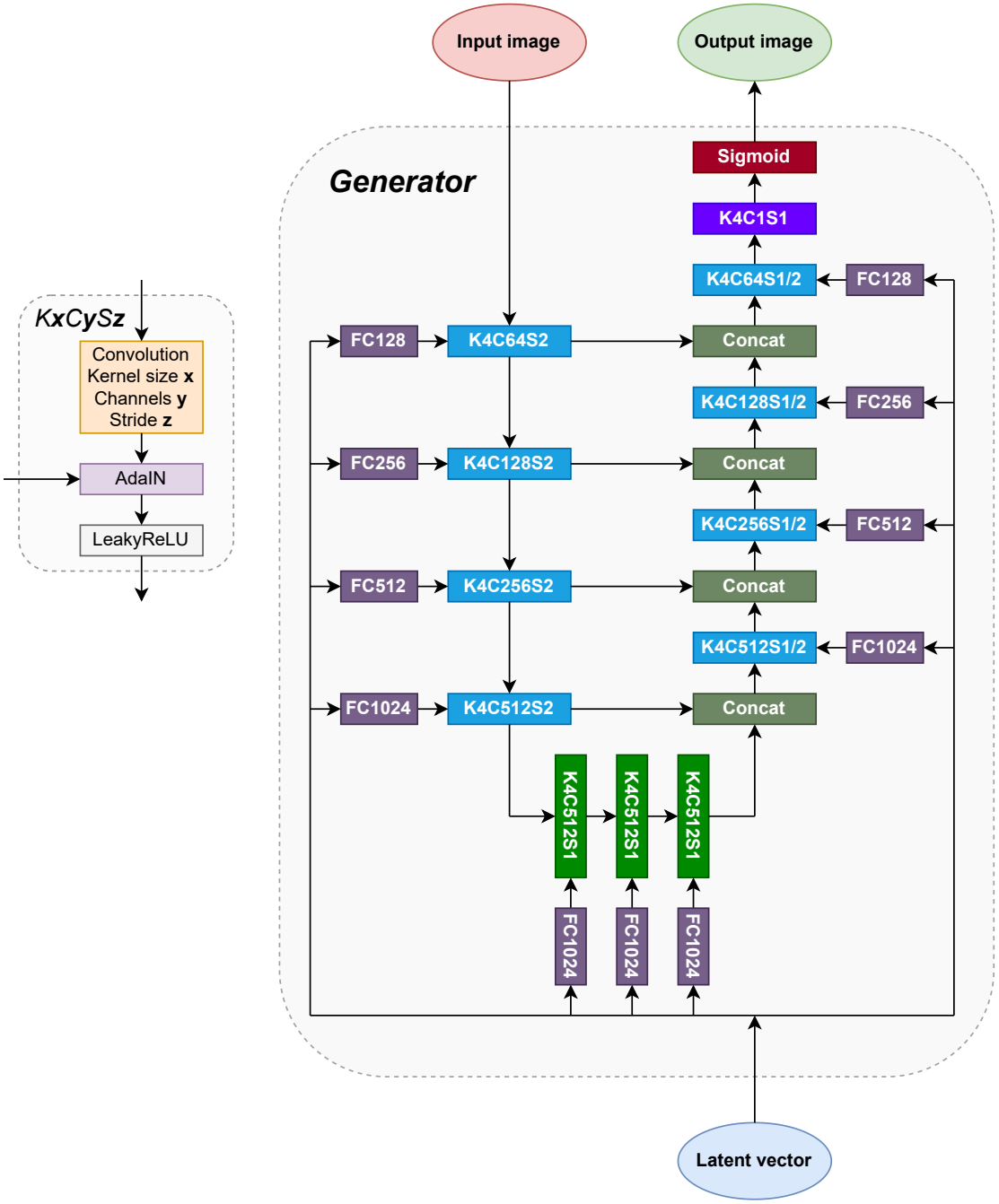


Figure 5.6: Modified generator architecture. Generator is shown on the right side of the image. Left side contains internals of convolutional layers with given parameters. The last layer is the exception, it does not contain AdaIN block. If stride is fractional, the convolution operation is transposed convolution. FCx represents fully connected layer with x output neurons.

compare the quality of generated fingerprints to the original data. In this work, for the most experiments, NIST SD302 [15] latent dataset is used, unlike in mentioned studies. I used an open source implementation of NFIQ 2[3], version *2.2.0* to calculate the score on NIST SD302 latent dataset. The average calculated score was around **2.6**, which was in my opinion too low for the baseline comparison. Both of these studies used NIST SD27 [16] latent dataset, where the average score is around 20, and the second study [5] also uses MSP longitudinal latent database [40], with the average score around 5, well above the figure for NIST SD302.

The another approach to indirectly evaluate synthetic fingerprint quality is to train some fingerprint recognition model with synthetic data, and see, if there is any performance improvement. This was done in the second study [5], where they trained DeepPrint [11] model with the augmented data, and showed, that it improved its Rank-1 retrieval performance. In the third study [39], they used custom neural network reconstruction model called ReconNet, trained on their augmented dataset generated by AugNet to prove that the Rank-1 accuracy is improved. The problem with these methods is that they are difficult to replicate, as there is not publicly available implementation of DeepPrint, nor ReconNet.

Despite these limitations and inability to accurately compare results to other state-of-the-art methods, some baseline needs to be established to at least compare the experiments between themselves. One of them is subjective image quality and identity preservation assessment, which is impractical for large number of experiments and data, and also highly inaccurate. I decided to use combination of 2 metrics. The first one of them is Fréchet inception distance (FID) [4], described in section 2.9. It is used to measure the distance between the distributions of real and generated images. It is not a 100% decisive metric, but at least it can provide rough measurements of how different are generated data from real data.

The second evaluation method was chosen to get at least some form of comparison of identity preservation between different experiments. For each of the selected setups, I generated a dataset of latent fingerprints. For each of these fingerprints, the match score was evaluated between it and the original binarized template, as well as randomly selected negative examples from the training set. The match score was calculated by *IEngine* software designed by Innovatrics company. Then, all the matching pairs were ordered in descending manner by the score. From that, Rank-1 matching accuracy was calculated, as a percentage of fingerprints in a dataset, that matched to their original template with the highest score, compared to negative samples. The pseudocode is provided in Algorithm 7. This method is based on an assumption, that if the identity is somehow compromised, the fake identity comes most likely from the other samples of the training data. The original plan was to compare each generated sample to each sample from the training data, but that was computationally infeasible, hence the limited number of negative samples.

In total, over 60 experiments (different network training runs) were conducted as a part of this thesis. Majority of them were unsuccessful, which were mostly described in the previous sections. I decided to choose only a part of the experiments, mostly the successful ones, for evaluation using formal metrics. I also included a few of the less successful experiments, to show a difference between individual training setups.

[3] https://github.com/usnistgov/NFIQ2

Algorithm 7 Rank-N accuracy evaluation

Inputs: Generated datasets $D_1, ..., D_n$, number of data in each dataset N_d, match scoring algorithm M, latent training dataset L, binarized training dataset B.

Hyperparameters: Rank N, number of negative matches from latent dataset l, number of negative matches from binarized dataset b.

Objective: Evaluate Rank-n accuracy

for dataset $= D_1, ..., D_n$ **do**
  $m = list()$
  for sample d in dataset **do**
    Sample l negative samples from L as d_l
    Sample b negative samples from B as d_b
    Evaluate match scores $s = M(concat(d, d_l, d_b))$
    Sort the list of scores s in descending order.
    if d is in the first N elements of the list s **and** $score(d) > s[N]$ **then** $m.append(1)$
    else $m.append(0)$
    end if
  end for
  Rank-N accuracy for current dataset $acc = sum(m)/N_d$
end for

Parameters of generated datasets

Here is the list of various hyperparameters that control the properties of generated dataset:

1. **Model version** - this is the most important property of the dataset. It covers all properties of the model (e.g. architecture), as well as training algorithm (e.g. loss function, data, optimizer, batch size, image normalization, etc.)

2. **Epoch number** - the checkpoints of the weights of the trained model were kept in previously specified intervals of epochs. After each run, the ones that looked most promising were saved. This was done to combat overfitting issues.

3. **Number of input fingerprint identities** - this parameter specified, how many different binarized fingerprint images were used to generate the dataset.

4. **Number of latent vectors per identity** - the vectors were sampled from Gaussian distribution, as the second input to the generator. Note that different vectors were sampled for each dataset instance.

5. **Latent vector parameters** - despite the fact that all models were trained using standard normal distribution ($\mu = 0, \sigma = 1$), I discovered that modifying these parameters in inference stage had sometimes positive effects on the dataset quality.

Calculating FID

For the first FID calculation, I decided to try as many hyperparameters as possible. The selected family of setups consisted of 25 model versions containing 37 epoch checkpoints. Each generated dataset consisted of 4000 images. All of the combinations of number of identities and number of vectors per identity were as follows:

1. 1 identity, 4000 vectors per identity
2. 16 identities, 250 vectors per identity
3. 200 identities, 20 vectors per identity
4. 4000 identities, 1 vector per identity

Each of these were also combined with latent vector gaussian distribution parameters, namely $\mu \in \{0, 1, -1\}, \sigma \in \{1.0, 0.5\}$, giving 6 combinations. Total number of datasets was 37 checkpoints $\times$ 4 identity parameters $\times$ 6 gaussian parameters = 888.

After these calculations were done, I analyzed the results. First, I looked at the best FID score for each latent dataset. The results are summarized in Table 5.6. It can be clearly seen from the table that the MOLF DB4 dataset has the best results, because the latents are much less challenging and diverse, than in the case of NIST SD302. The fingerprints in the subset of NIST SD302 are less challenging than the ones in the original dataset, and they are also aligned, which could raise a question of why they are worse in the FID metric. However, this dataset was used only in the early versions of the model, before all of the significant improvements.

Table 5.6: Summary of best FID scores for each dataset on the first run.

Dataset	**Best FID**
NIST SD302 latent	98.398
NIST SD302 latent subset	165.147
MOLF DB4	31.549

After this experiment, I also tried to verify, that FID metric in some way reflects image quality, and similarity to the training dataset. The comparison can be seen in Figure 5.7. It turns out that there are some minor discrepancies between subjective image realism rating, and FID score. For example, the first row achieved better score than the second, but the realism of images is subjectively better in the second row. There could be a few reasons for this, which are discussed in section 5.9. Nevertheless, it also shows that there is indeed a general trend, that the images get better with lower FID scores.

When it comes to input binarized image distribution, I decided to only use 200 identities with 20 latent vectors per identity in the subsequent experiments, as they seemed to produce lowest FID scores. I also explored different gaussian parameters of latent noise vector. I noticed that more out-of-center vectors ($\mu = -1, 1, \sigma = 1.0$) had usually better FID scores, but worse image quality. This might be due to a fact that they produced more diverse images, making them look closer to real latent data distribution. On the other side, the out-of-center vectors with lower standard deviation sometimes generated even better images, than standard normal distribution. I decided to use this fact in the next evaluation process.

5.9 Evaluation results

For the final evaluation, Rank-1 accuracy was evaluated together with FID scores, as is described in section 5.8. It was not computationally feasible to evaluate the same number of databases, as in previous calculation. But given the observations at the first FID calculation, I reduced the total number of datasets. I also included one more model versions, which was

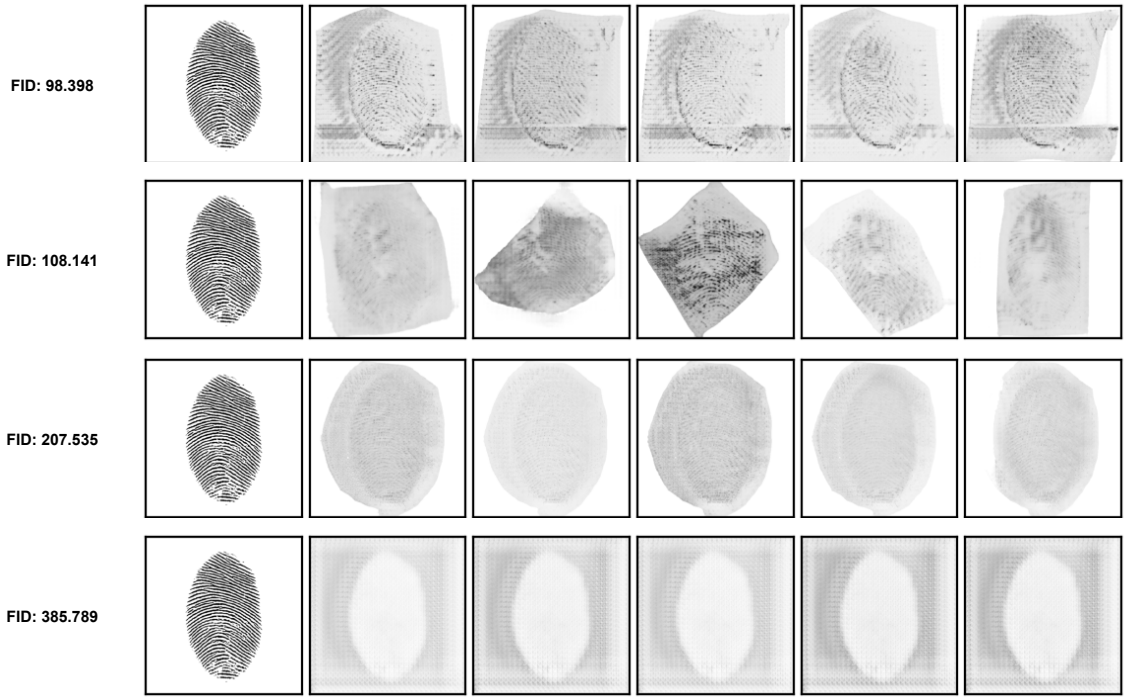


Figure 5.7: Comparison of calculated FID scores on NIST SD302 latent dataset. Despite the fact that there are some minor discrepancies (first row has subjectively worse image realism, but achieved better FID score than the second), the FID metric in general seems to be a good way to evaluate individual models.

not previously included in FID calculation. Complete hyperparameters of this evaluation are summarized in Table 5.7. Note that FID and Rank-n accuracy were calculated at datasets with different sizes, again due to computational capacity, but with the exact same setup. In all of the following experiments, the batch size of 16 was used.

Table 5.7: Hyperparameters of final evaluation.

Model versions	26
Checkpoints	38
Gaussian noise parameters	$(\mu, \sigma) \in \{(0.0, 1.0), (0.0, 0.5), (0.5, 0.5), (-0.5, 0.5)\}$
Datasets	152
Images per dataset (Rank-n)	100
Identities per dataset (Rank-n)	20
Vectors per identity (Rank-n)	5
Images per dataset (FID)	4000
Identities per dataset (FID)	200
Vectors per identity (FID)	20
Negative latent images per sample	160
Negative binarized images per sample	39
Evaluated image pairs (Rank-n)	3040000

To get an overview of all results, I made a plot depicting relationship between Rank-1 accuracy and FID for each generated dataset (Figure 5.8). As it can be seen from the graph, there is some form of negative correlation between Rank-1 accuracy and FID. The FID seems to generally decrease with increasing Rank-1 accuracy, which means that image realism and identity preservation are tied together. There are a few exceptions, which will be discussed in following subsections.

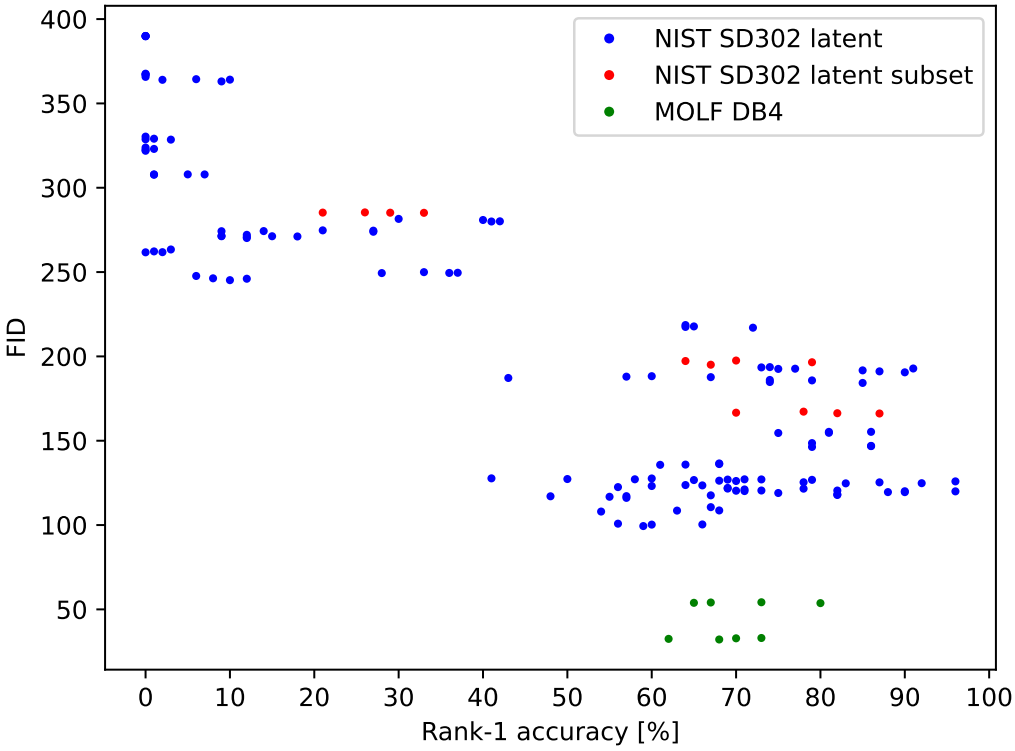


Figure 5.8: Relationship between Rank-1 accuracy and FID score. Individual points represent different generated datasets. Color of the point marks latent dataset used for training, as well as evaluation.

WGAN-GP vs. LSGAN

After finding effective generator architecture and subsequently fine-tuning WGAN-GP algorithm parameters, 2 separate model training runs with LSGAN loss function were conducted. I compared best results from LSGAN algorithm to best results from WGAN-GP algorithm (given that data and model architecture stayed the same), and compared them together in Table 5.8. All statistics are calculated with respect to NIST SD302 latent dataset. As it can be seen, WGAN-GP algorithm achieves better FID score, while LSGAN achieves better Rank-1 accuracy. The WGAN-GP algorithm also needed more iterations to converge than the LSGAN. On the contrary, the WGAN-GP algorithm was much less prone to instability during training, and showed better robustness to many hyperparameter changes, hence much more experiments were conducted with it, as opposed to LSGAN.

Table 5.8: Comparison of WGAN-GP and LSGAN algorithms. 2 experiments from each algorithm are selected, the one with the best FID and the one with the best Rank-1 accuracy. All statistics are calculated with respect to NIST SD302 latent dataset.

Algorithm	**Epochs**	**Best metric**	**FID**	**Rank-1 accuracy**
WGAN-GP	600	FID	**99.399**	59%
WGAN-GP	360	Rank-1-accuracy	155.270	86%
LSGAN	76	FID	184.269	85%
LSGAN	72	Rank-1-accuracy	192.831	**91%**

Generator architectures

As was briefly mentioned in section 5.7, the change of generator architecture massively reduced mode collapse problem, and resulted in increased image quality. The former architecture is shown in Figure 4.3, the improved architecture is in Figure 5.6. The increase in diversity of generated images can be seen in Figure 5.5 (before) and Figure 5.9 (after). The difference in measured metrics is summarized in Table 5.9. Both of the models used NIST SD302 latent training dataset, the training algorithm used was WGAN-GP, the only difference was in the generator architecture. All statistics are calculated with respect to NIST SD302 latent dataset. The mode collapse reduction problem is reflected in the rapid change in FID metric.

As can be seen in Figure 5.6, the AdaIN block is a part of each convolutional layer. I also experimented with replacing AdaIN with classic learnable instance normalization in first 4 downsampling layers. The motivation behind this was that the first 4 layers should serve as an identity extractor, so no distortions from the latent vector should be incorporated in this part. Table 5.10 summarizes the difference. All statistics are calculated with respect to NIST SD302 latent dataset. Despite the expectations, the version with AdaIN blocks in all layers showed better results, outperforming the other version in both FID and Rank-1 accuracy metrics.

Table 5.9: Comparison of generator architectures. 2 experiments from each architecture are selected, the one with the best FID and the one with the best Rank-1 accuracy. All statistics are calculated with respect to NIST SD302 latent dataset.

Generator architecture	**Best metric**	**FID**	**Rank-1 accuracy**
Former architecture	FID	245.211	10%
Former architecture	Rank-1-accuracy	246.052	12%
Improved architecture	FID	**146.320**	79%
Improved architecture	Rank-1-accuracy	155.270	**86%**

Training process

The specific adaptation of WGAN-GP for training our model is described in Algorithm 6. As it can be seen, the binarized fingerprints are sampled randomly from training set. Before this alternative, there were also experiments with keeping the binarized fingerprint constant, to focus more of the training to latent fingerprint style adaptation. As Table 5.11 shows, the algorithm with sampling random binarized fingerprints achieved better FID

Table 5.10: Comparison of generator architectures. 2 experiments from each architecture are selected, the one with the best FID and the one with the best Rank-1 accuracy. All statistics are calculated with respect to NIST SD302 latent dataset.

Generator architecture	**Best metric**	**FID**	**Rank-1 accuracy**
AdaIN in all layers	FID	**99.399**	59%
AdaIN in all layers	Rank-1-accuracy	117.827	**82%**
No AdaIN in downsampling layers	FID	122.506	56%
No AdaIN in downsampling layers	Rank-1-accuracy	123.518	66%

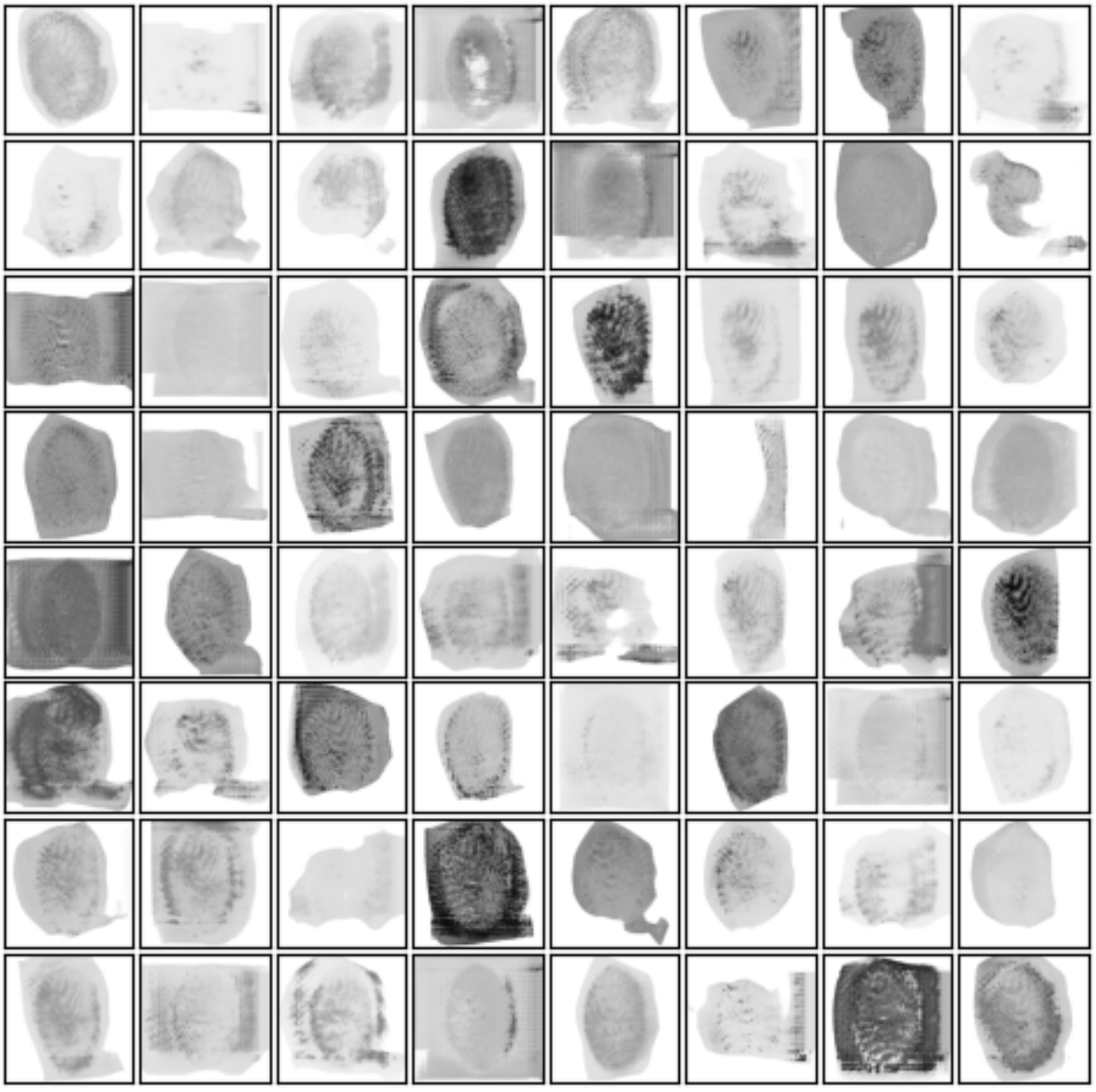


Figure 5.9: Generator output after solving mode collapse problem.

score, and the other one was slightly better at Rank-1 accuracy. The difference in best Rank-1 accuracy is however not that significant, as in FID score.

Table 5.11: Comparison of generator input in training algorithm. 2 experiments from each architecture are selected, the one with the best FID and the one with the best Rank-1 accuracy. All statistics are calculated with respect to NIST SD302 latent dataset.

Generator input	**Best metric**	**FID**	**Rank-1 accuracy**
Random binarized fingerprints	FID	**99.399**	59%
Random binarized fingerprints	Rank-1-accuracy	117.827	82%
Constant binarized fingerprint	FID	146.320	79%
Constant binarized fingerprint	Rank-1-accuracy	155.270	**86%**

Discriminator architecture

For the most experiments, discriminator with the same architecture was used, shown in Figure 2.5 (b). The output of the last layer was 32×32 matrix, which was then averaged to a single scalar providing final network output. Because most of the experiments in the later stages used WGAN-GP training algorithm, I decided to take a look at reference implementations of this algorithm. In most cases, such as this[4] GitHub repo, fully connected dense layer was used after the convolutions, instead of simple average operation. Since the core idea of WGAN-GP is that the discriminator should be able to approximate Wasserstein distance, it makes sense to put this final layer after the convolutions, as it can learn more complex operations.

I compared 2 experiments, first with the average operation, and the second with the fully connected layer. Despite my expectations, as can be seen in Table 5.12, the original architecture performed better in both metrics at first. However this is not taking into account hyperparameters that needed to be adjusted after the architecture change, it is comparing the exact same setup, just the different discriminator.

There is another reason I think that the version without fully connected layer achieved better results. Figure 5.10 compares total loss of the version without the final layer (a) to the version with the final layer (b) and to an example taken from WGAN-GP [18] paper (c). Although the version without the final layer achieves better results, it is obvious that its loss is completely different, than in (b) or (c). The version with the final layer is much more similar to the original paper. In addition, according to graphs (d) and (e), the model without the final layer has optimized gradient penalty term much quicker than the other model, which is suspicious. In my opinion, the discriminator without the final layer, due to its simplicity, found some local optimum, in which it did not estimate Wasserstein-1 distance correctly, thus leading also to better FID scores (since FID is an approximation of Wasserstein-2 distance). Besides that, it is also obvious, that the model with the final layer should have been trained for much longer, or with higher learning rate. For that reasons, I used the version with the final layer in the following experiments, and tried to find optimal hyperparameters for the training.

WGAN-GP hyperparameters

As was mentioned in the previous subsection, the new discriminator architecture needed some hyperparameter tuning to be more optimal. Learning rate was set to $1e-4$ in all previous WGAN-GP experiments, as recommended in the paper. Because the updated discriminator seemed to be training slower, I decided to increase learning rate. I also

[4] https://github.com/caogang/wgan-gp

Table 5.12: Comparison of discriminator architectures. 2 experiments from each architecture are selected, the one with the best FID and the one with the best Rank-1 accuracy. All statistics are calculated with respect to NIST SD302 latent dataset.

Discriminator architecture	**Best metric**	**FID**	**Rank-1 accuracy**
Average at the end	FID	**99.399**	59%
Average at the end	Rank-1-accuracy	117.827	**82%**
Fully connected layer at the end	FID	108.004	54%
Fully connected layer at the end	Rank-1-accuracy	108.664	68%

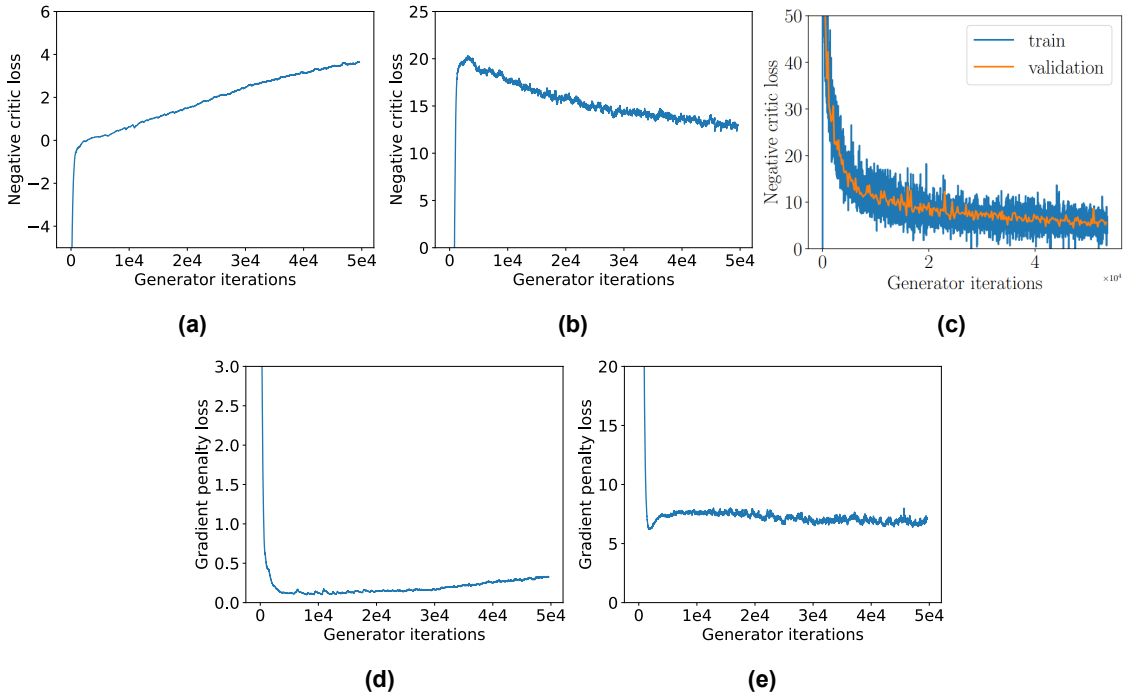


Figure 5.10: Comparison of WGAN-GP losses. (a) Total loss using discriminator without final layer. (b) Total loss using discriminator with final layer. (c) Example loss from WGAN-GP paper. [18] (d) Gradient penalty loss term using discriminator without final layer. (e) Gradient penalty loss term using discriminator with final layer.

tuned λ parameter, the gradient penalty weight. In all of the WGAN-GP experiments, n_{critic} (number of training steps of discriminator per one generator step) was set to 5, as recommended in the original paper. The recommended optimizer and its hyperparameters were also used (Adam [24], $\beta_1 = 0.0, \beta_2 = 0.9$). Results are summarized in Table 5.13. While the original architecture achieved better FID score, I did not evaluate these results in the time of conducting experiments, so I was only deciding based on subjective image quality rating. For the last few experiments, I actually used learning rate of $5e-4$ and gradient penalty of 20, which, maybe coincidentally, achieved best Rank-1 accuracy not only from this subset of experiments, but also out of all experiments.

Table 5.13: Comparison of different WGAN-GP training hyperparameters. 2 experiments from each architecture are selected, the one with the best FID and the one with the best Rank-1 accuracy, except for the last one. All statistics are calculated with respect to NIST SD302 latent dataset. $\lambda_{\mathbf{GP}}$ represents gradient penalty weight.

Learning rate	$\lambda_{\mathbf{GP}}$	**Epochs**	**Best metric**	**FID**	**Rank-1 accuracy**
$1e-4$	10.0	400	FID	**108.004**	54%
$1e-4$	10.0	400	Rank-1-accuracy	108.664	68%
$5e-4$	10.0	400	FID	126.686	65%
$5e-4$	10.0	16	Rank-1-accuracy	192.706	77%
$5e-4$	20.0	200	FID	120.358	70%
$5e-4$	20.0	200	Rank-1-accuracy	120.460	82%
$5e-4$	20.0	264	FID	119.517	90%
$5e-4$	20.0	264	Rank-1-accuracy	119.979	**96%**
$5e-4$	5.0	200	FID	135.709	61%
$5e-4$	5.0	200	Rank-1-accuracy	136.527	68%
$5e-4$	40.0	200	Both	116.117	57%

Image normalization

For all of the experiments up to this point, training images were normalized in range $[0; 1]$. In the paper about DCGAN [30], they introduce a few general recommendations for convolutional GAN models. One of them is normalizing images in range $[-1, 1]$ instead. These 2 image normalization methods are compared in Table 5.14. In both of these cases, best Rank-1 accuracy achieved was 96%. The original $[0; 1]$ normalization achieved slightly better FID score. Besides that, it does not seem like there is any significant difference in these methods. When using $[-1; 1]$ normalization, the sigmoid layer in generator is replaced with *hyperbolic tangent* function (TanH).

Table 5.14: Comparison of training image normalization. 2 experiments from each architecture are selected, the one with the best FID and the one with the best Rank-1 accuracy, except for the case where both are the best. All statistics are calculated with respect to NIST SD302 latent dataset.

Normalization	**Epochs**	**Best metric**	**FID**	**Rank-1 accuracy**
$[0; 1]$	200	FID	120.358	70%
$[0; 1]$	200	Rank-1-accuracy	120.460	82%
$[0; 1]$	264	FID	**119.517**	90%
$[0; 1]$	264	Rank-1-accuracy	119.979	**96%**
$[-1; 1]$	200	Both	120.153	71%
$[-1; 1]$	400	FID	124.724	83%
$[-1; 1]$	400	Rank-1-accuracy	125.893	**96%**

MOLF DB4 dataset

After finding optimal WGAN-GP parameters, I tried to run the same training with another latent dataset, namely MOLF DB4 [32]. This dataset contains less diverse and less challenging latent prints than MOLF DB4, so I expected a lot more different results. Indeed,

the FID scores for the MOLF dataset are significantly lower, but that is expected, due to mentioned reasons about data. Upon visual inspection of generated images after each saved checkpoint, I decided to save epochs 80 and 400, because the model looked like it was overfitting after 80 epochs. It converged much faster, than in the case of NIST SD302. Although Rank-1 accuracy increased after training for 400 epochs, the image quality got worse. Nevertheless, this dataset showed consistently lower Rank-1 accuracy scores than NIST SD302. The reason for that might be that some of the latent fingerprints had the same identity as the binarized ones, since the binarized fingerprints were taken from MOLF DB1 dataset. That might have caused some identity mixing. In Figure 5.11 it can be seen in the second, third, and the last image from the left. It seems like the original identity is overshadowed by some different fingerprint from the latent dataset.

Table 5.15: Comparison of different latent training datasets. 2 experiments from each architecture are selected, the one with the best FID and the one with the best Rank-1 accuracy. Statistics are calculated with respect to given dataset.

Dataset	**Epochs**	**Best metric**	**FID**	**Rank-1 accuracy**
NIST SD302 latent	200	FID	120.358	70%
NIST SD302 latent	200	Rank-1-accuracy	120.460	82%
NIST SD302 latent	264	FID	119.517	90%
NIST SD302 latent	264	Rank-1-accuracy	119.979	**96%**
MOLF DB4	80	FID	**32.153**	68%
MOLF DB4	400	Rank-1-accuracy	53.713	80%

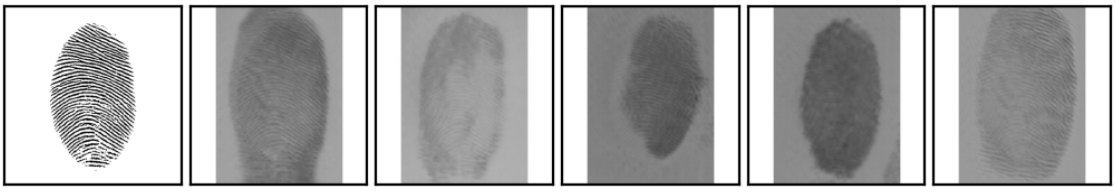


Figure 5.11: Examples of generated fingerprints after training on MOLF DB4 dataset. The fingerprint on the left is binarized template, the other ones are generated by the model.

5.10 Summary

Despite the fact that Rank-1 accuracy and FID are generally negatively related (Figure 5.8), it is not possible to select single best performing setup. Because of these reasons, I summarize the best results as a Pareto front for each of the 2 datasets: NIST SD302 latent (Table 5.16) and MOLF DB4 (Table 5.17). Aligned subset of NIST SD302 latent is not included, since it was only used in early experiments, which did not provide satisfying results. For each of these setups, CMC curve is also included, with ranked accuracy up to 25 (Figure 5.12, Figure 5.13). All of these setups are included in the storage media.

Table 5.16: Pareto front for datasets generated by models trained on NIST SD302 latent dataset.

Setup	FID	Rank-1 accuracy
Setup 1	99.399	59%
Setup 2	100.263	60%
Setup 3	100.354	66%
Setup 4	108.664	68%
Setup 5	117.827	82%
Setup 6	119.517	90%
Setup 7	119.979	96%

Table 5.17: Pareto front for datasets generated by models trained on MOLF DB4 dataset.

Setup	FID	Rank-1 accuracy
Setup 1	32.153	68%
Setup 2	32.843	70%
Setup 3	33.034	73%
Setup 4	53.713	80%

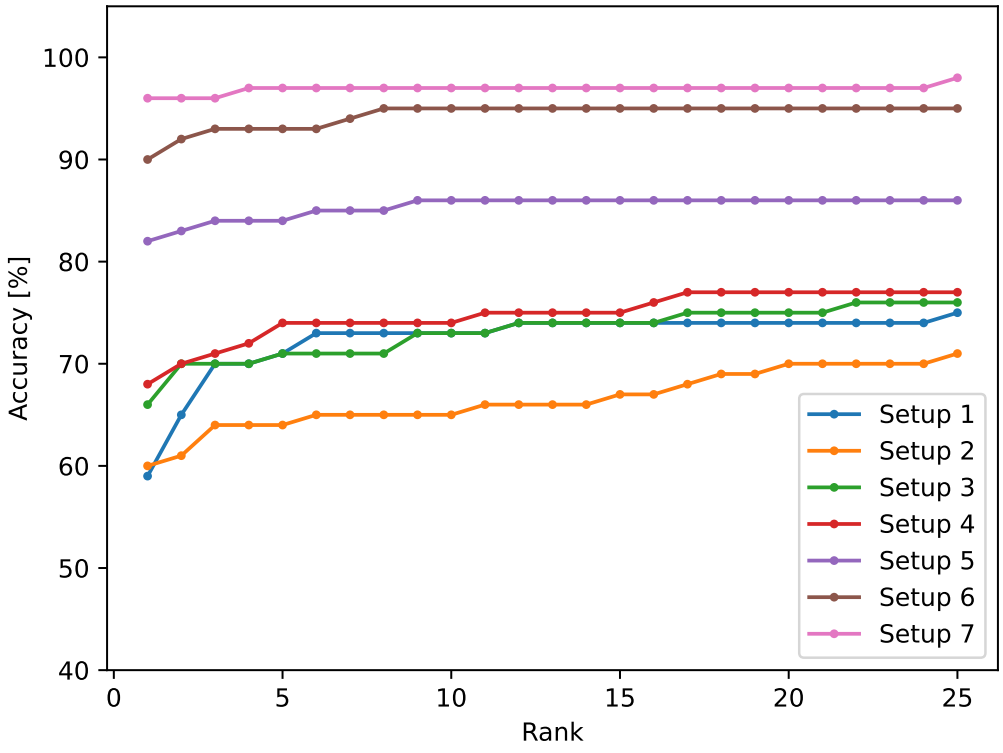


Figure 5.12: CMC curves of datasets generated by Pareto optimal setups trained on NIST SD302 latent dataset.

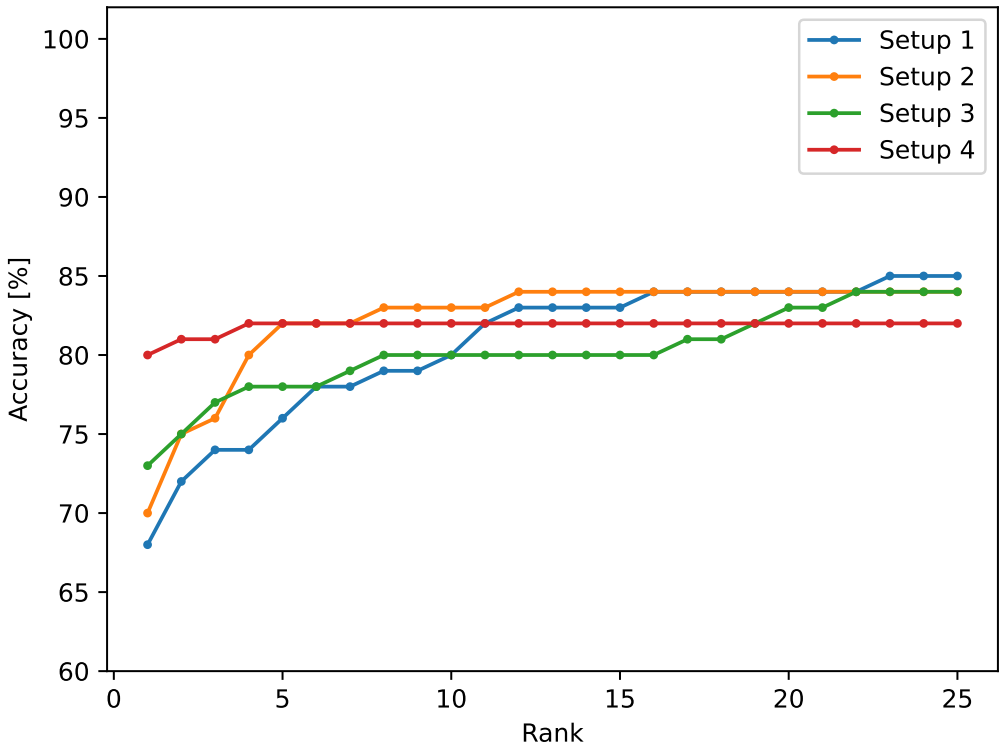


Figure 5.13: CMC curves of datasets generated by Pareto optimal setups trained on MOLF DB4 dataset.

Comparison to other methods

As was talked about in section 5.8, quantitative comparison to other state-of-the-art methods was not done due to difficulties in replicating evaluation methods. Because of this reasons, only visual comparison of a few samples is provided in Figure 5.14. Samples (a) and (b) are from study [5], trained on MSP latent dataset [40]. Image (c) was generated by Aug-cGAN, baseline version of AugNet [39], by which image (d) was generated. They used MOLF DB4 as a latent training dataset. All other images (e-h) are samples from datasets generated by some of the setups, all of them trained on NIST SD302 latent dataset.

Possibilities for future improvements

This work was dedicated mainly to training conditional GAN model, inspired by first stage of training AugNet [39] model. Despite the original plan of replicating and possibly improving the second stage of AugNet training, it was not possible due to the problems with mode collapse, which were eventually solved. The fact that the generator architecture change had such a big impact on results (Table 5.9) shows that there was, and still is a lot more room for improvement in conditional GAN training for latent fingerprint generation. Besides that, more experiments with training algorithm hyperparameters, discriminator architecture, and various training data could improve these results in my opinion. It would be also interesting to see, if the second AugNet training stage would actually improve image quality

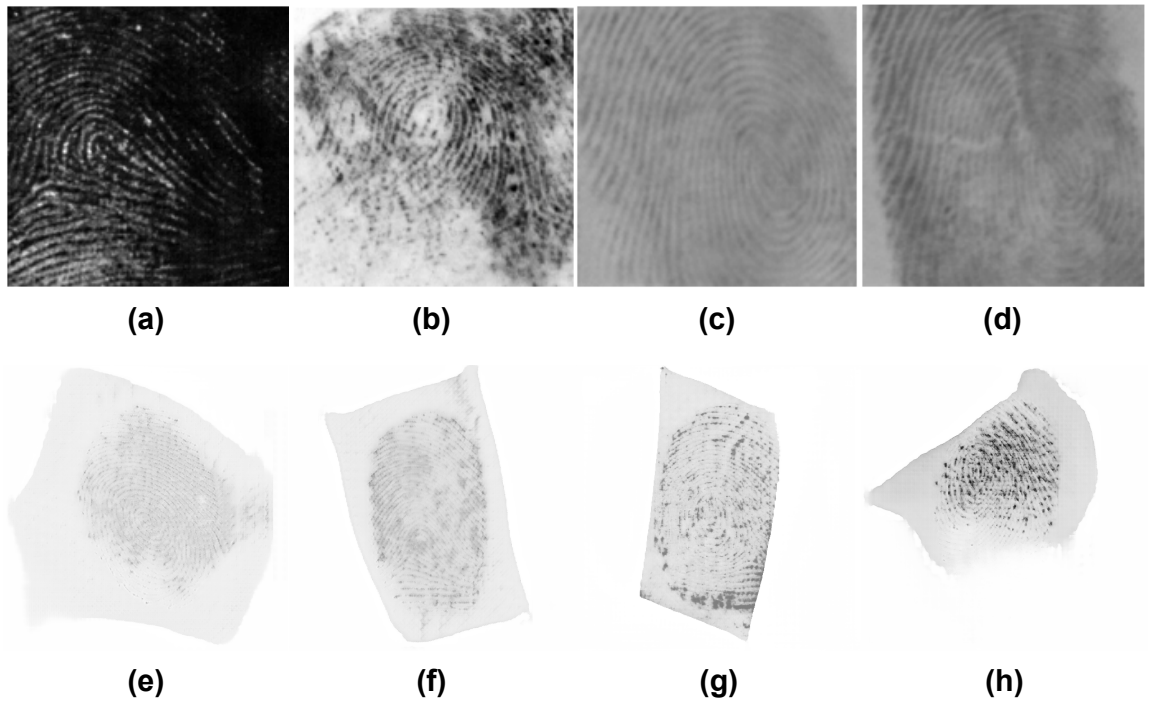


Figure 5.14: Visual comparison of generated images to other methods. (a), (b) Images from study [5]. (c), (d) Images generated by Aug-cGAN and AugNet [39]. (e-h) Images generated by methods in this work.

and identity preservation, as the authors claim. More progress could also be done in evaluation methods, either setting some baseline for identity preservation testing, or training some fingerprint recognition model on original vs. generated data, and seeing the difference in its performance. As this work shows, there is a lot of variables that have effect on the results, and there is still a lot to discover, when it comes to generating latent fingerprints via GANs.

Chapter 6

Conclusion

Aim of this thesis was to design, implement, and evaluate algorithm for fingerprint generation using Generative adversarial networks. The emphasis was put on generating latent fingerprints from their clean counterparts, preserving their identity, generating multiple impressions per fingerprint, and being able to control the identity and style separately.

Multiple state of the art methods for this task were explored. Based on given criteria, augmentation framework inspired by AugNet [39] was selected, as it at least partially fulfilled all of them. Initial task was to replicate the AugNet model, and to find out, if it produces the results declared in the paper. That has proven to be difficult, as multiple details about the model were not present in the paper, such as combining latent vector with the clean fingerprint. For that, StyleGAN [23] based style adaptation was chosen, using AdaIN [20] block, as it has shown to be able combine multiple levels of feature details together.

First experiments with the original AugNet algorithm did not yield any usable results. For the rest of the thesis, I decided to experiment only with conditional GAN training, mainly because of mode collapse problem. Many different strategies were tested to combat this problem, such as changing the dataset, training algorithm, or using multiple discriminators instead of one. None of these changes helped in reducing the mode collapse. Eventually, training objective was changed from LSGAN [26] to WGAN-GP [18], as it made training more stable. However, the single most effective improvement turned out to be change of the generator architecture, specifically in placement of AdaIN [20] blocks. This change also dramatically reduced mode collapse.

As it was difficult to replicate evaluation methods used in other papers to compare the results to them, custom methods were designed to select best performing setups. Individual model setups were evaluated using combination of FID and Rank-1 accuracy on matching generated latent fingerprints to their original binarized templates. Out of all setups, the one with Pareto optimal combinations of these 2 metrics were selected as the best, which are also included in the storage media.

There is a lot of opportunities for future improvements of this work. Given the fact that minor change in network architecture could significantly improve results, it seems like there is still a lot of hyperparameters to tune, which could push the image quality even further. It would also be interesting to see, how would the original AugNet [39] training algorithm perform with the improved baseline conditional GAN. Finally, it comes to the subject of evaluation itself. More robust metrics could be designed to evaluate such algorithms, such as training some publicly available model with real and generated data, and comparing the differences in its performance on these 2 datasets.

Bibliography

[1] Ansari, A. H. Generation and Storage of Large Synthetic Fingerprint Database. In:. 2011.

[2] Arjovsky, M., Chintala, S. and Bottou, L. *Wasserstein GAN*. 2017.

[3] Bausinger, O. and Tabassi, E. Fingerprint Sample Quality Metric NFIQ 2.0. In:. January 2011, p. 167–171.

[4] Borji, A. *Pros and Cons of GAN Evaluation Measures*. 2018.

[5] Brasil Vieira Wyzykowski, A. and Jain, A. K. Synthetic Latent Fingerprint Generator. In: *2023 IEEE/CVF Winter Conference on Applications of Computer Vision (WACV)*. 2023, p. 971–980. DOI: 10.1109/WACV56688.2023.00103.

[6] Brock, A., Donahue, J. and Simonyan, K. Large Scale GAN Training for High Fidelity Natural Image Synthesis. *CoRR*. 2018, abs/1809.11096. Available at: http://arxiv.org/abs/1809.11096.

[7] Cappelli, R. SFinGe: an Approach to Synthetic Fingerprint Generation. *International Workshop on Biometric Technologies*. january 2004.

[8] Cappelli, R., Ferrara, M., Franco, A. and Maltoni, D. Fingerprint verification competition 2006. *Biometric Technology Today*. 2007, vol. 15, no. 7, p. 7–9. DOI: https://doi.org/10.1016/S0969-4765(07)70140-6. ISSN 0969-4765. Available at: https://www.sciencedirect.com/science/article/pii/S0969476507701406.

[9] Chambolle, A. An Algorithm for Total Variation Minimization and Applications: Special Issue on Mathematics and Image Analysis. *Journal of Mathematical Imaging and Vision*. january 2004, vol. 20.

[10] Choi, J. and Han, B. *MCL-GAN: Generative Adversarial Networks with Multiple Specialized Discriminators*. 2021.

[11] Engelsma, J. J., Cao, K. and Jain, A. K. Learning a Fixed-Length Fingerprint Representation. *CoRR*. 2019, abs/1909.09901. Available at: http://arxiv.org/abs/1909.09901.

[12] Engelsma, J. J., Grosz, S. A. and Jain, A. K. PrintsGAN: Synthetic Fingerprint Generator. *CoRR*. 2022, abs/2201.03674. Available at: https://arxiv.org/abs/2201.03674.

[13] Feng, J., Yoon, S. and Jain, A. K. Latent Fingerprint Matching: Fusion of Rolled and Plain Fingerprints. In: Tistarelli, M. and Nixon, M. S., ed. *Advances in Biometrics*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, p. 695–704.

[14] Fiumara, G., Tabassi, E., Flanagan, P., Grantham, J., Ko, K. et al. *Nail to Nail Fingerprint Challenge: Prize Analysis*. NIST Interagency/Internal Report (NISTIR), National Institute of Standards and Technology, Gaithersburg, MD, 2018-05-03 2018. DOI: https://doi.org/10.6028/NIST.IR.8210.

[15] Fiumara, G. P., Flanagan, P. A., Grantham, J. D., Ko, K., Marshall, K. et al. NIST special database 302: Nail to nail fingerprint challenge. 2019.

[16] Garris, M. D. and Garris, M. D. *NIST special database 27: Fingerprint minutiae from latent and matching tenprint images*. US Department of Commerce, National Institute of Standards and Technology . . . , 2000.

[17] Goodfellow, I. J., Pouget Abadie, J., Mirza, M., Xu, B., Warde Farley, D. et al. *Generative Adversarial Networks*. arXiv, 2014. DOI: 10.48550/ARXIV.1406.2661. Available at: https://arxiv.org/abs/1406.2661.

[18] Gulrajani, I., Ahmed, F., Arjovsky, M., Dumoulin, V. and Courville, A. *Improved Training of Wasserstein GANs*. 2017.

[19] He, K., Zhang, X., Ren, S. and Sun, J. Identity Mappings in Deep Residual Networks. *CoRR*. 2016, abs/1603.05027. Available at: http://arxiv.org/abs/1603.05027.

[20] Huang, X. and Belongie, S. J. Arbitrary Style Transfer in Real-time with Adaptive Instance Normalization. *CoRR*. 2017, abs/1703.06868. Available at: http://arxiv.org/abs/1703.06868.

[21] Isola, P., Zhu, J., Zhou, T. and Efros, A. A. Image-to-Image Translation with Conditional Adversarial Networks. *CoRR*. 2016, abs/1611.07004. Available at: http://arxiv.org/abs/1611.07004.

[22] Jain, A. K., Ross, A. A. and Nandakumar, K. *Introduction to Biometrics*. Springer Publishing Company, Incorporated, 2011. ISBN 0387773258.

[23] Karras, T., Laine, S. and Aila, T. A Style-Based Generator Architecture for Generative Adversarial Networks. *CoRR*. 2018, abs/1812.04948. Available at: http://arxiv.org/abs/1812.04948.

[24] Kingma, D. and Ba, J. Adam: A Method for Stochastic Optimization. *International Conference on Learning Representations*. december 2014.

[25] Lenton, D. *Part II: Projective Transformations in 2D*. Jun 2019. Available at: https://medium.com/@unifyai/part-ii-projective-transformations-in-2d-2e99ac9c7e9f.

[26] Mao, X., Li, Q., Xie, H., Lau, R. Y. K. and Wang, Z. Multi-class Generative Adversarial Networks with the L2 Loss Function. *CoRR*. 2016, abs/1611.04076. Available at: http://arxiv.org/abs/1611.04076.

[27] Minaee, S. and Abdolrashidi, A. Finger-GAN: Generating Realistic Fingerprint Images Using Connectivity Imposed GAN. *CoRR*. 2018, abs/1812.10482. Available at: http://arxiv.org/abs/1812.10482.

[28] Mirza, M. and Osindero, S. Conditional Generative Adversarial Nets. *CoRR*. 2014, abs/1411.1784. Available at: http://arxiv.org/abs/1411.1784.

[29] Otsu, N. A Threshold Selection Method from Gray-Level Histograms. *IEEE Transactions on Systems, Man, and Cybernetics*. 1979, vol. 9, no. 1, p. 62–66. DOI: 10.1109/TSMC.1979.4310076.

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

[31] Sankaran, A., Agarwal, A., Keshari, R., Ghosh, S., Sharma, A. et al. Latent fingerprint from multiple surfaces: Database and quality analysis. In: *2015 IEEE 7th International Conference on Biometrics Theory, Applications and Systems (BTAS)*. 2015, p. 1–6. DOI: 10.1109/BTAS.2015.7358773.

[32] Sankaran, A., Vatsa, M. and Singh, R. Multisensor Optical and Latent Fingerprint Database. *IEEE Access*. 2015, vol. 3, p. 653–665. DOI: 10.1109/ACCESS.2015.2428631.

[33] Seidlitz, S., Jürgens, K., Makrushin, A., Kraetzer, C. and Dittmann, J. Generation of Privacy-friendly Datasets of Latent Fingerprint Images using Generative Adversarial Networks. In:. January 2021, p. 345–352. DOI: 10.5220/0010251603450352.

[34] Ulery, B., Hicklin, R., Watson, C. I., Indovina, M. D., Hanaoka, K. et al. Slap fingerprint segmentation evaluation 2004 analysis report. 2005.

[35] Vatsa, M., Singh, R., Noore, A. and Morris, K. Simultaneous latent fingerprint recognition. *Applied Soft Computing*. 2011, vol. 11, no. 7, p. 4260–4266. DOI: https://doi.org/10.1016/j.asoc.2011.02.005. ISSN 1568-4946. Soft Computing for Information System Security. Available at: https://www.sciencedirect.com/science/article/pii/S1568494611000652.

[36] Vieira Wyzykowski, A. B., Segundo, M. P. and Paula Lemes, R. de. Level Three Synthetic Fingerprint Generation. In: *2020 25th International Conference on Pattern Recognition (ICPR)*. 2021, p. 9250–9257. DOI: 10.1109/ICPR48806.2021.9412304.

[37] Watson, C. I. NIST special database 14. *Fingerprint Database, US National Institute of Standards and Technology*. Citeseer. 1993.

[38] Watson, C. I. and Wilson, C. L. NIST special database 4. *Fingerprint Database, National Institute of Standards and Technology*. Citeseer. 1992, vol. 17, no. 77, p. 5.

[39] Xu, Y., Wang, Y., Liang, J. and Jiang, Y. Augmentation Data Synthesis Via Gans: Boosting Latent Fingerprint Reconstruction. In: *ICASSP 2020 - 2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. 2020, p. 2932–2936. DOI: 10.1109/ICASSP40776.2020.9053801.

[40] Yoon, S. and Jain, A. K. Longitudinal study of fingerprint recognition. *Proceedings of the National Academy of Sciences*. 2015, vol. 112, no. 28, p. 8555–8560. DOI: 10.1073/pnas.1410272112. Available at: https://www.pnas.org/doi/abs/10.1073/pnas.1410272112.

[41] Zhao, Q., Zhang, D., Zhang, L. and Luo, N. High resolution partial fingerprint alignment using pore–valley descriptors. *Pattern Recognition*. 2010, vol. 43, no. 3, p. 1050–1061. DOI: https://doi.org/10.1016/j.patcog.2009.08.004. ISSN 0031-3203. Available at: https://www.sciencedirect.com/science/article/pii/S0031320309003045.

[42] Zhu, J., Park, T., Isola, P. and Efros, A. A. Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks. *CoRR*. 2017, abs/1703.10593. Available at: http://arxiv.org/abs/1703.10593.