Application of convolutional neural networks for fingerprint recognition

Tuong Lam & Simon Nilsson

June 14, 2018

Abstract

Fingerprint recognition is a well-known problem in pattern recognition and widely used in contemporary authentication technology such as access devices in mobile phones. The subject of this thesis is to investigate the applicability of convolutional neural networks for fingerprint recognition. This is accomplished by designing various network architectures for this task. Our starting-point is an architecture known as a siamese network, from which we build upon by including additional components as well as network architectures based on the siamese architecture. The networks are realized by implementation. Data for training and evaluating the networks is provided as gray-scale images of fingerprints and we implement a simple algorithm for generating ground truth labels. To evaluate our work, we measure the performance of all implemented models with common metrics for fingerprint recognition algorithms. Lastly problems with our approach are listed and potential future improvements are given.

Preface

This thesis was conducted between January and June, 2018 in association with the mathematics department at the faculty of engineering, Lund university and the company Precise Biometrics located in Lund. Most of the research was carried out at the office of Precise Biometrics. The purpose of this thesis was to investigate the potential of using methods for fingerprint recognition that differ from those commonly used. Our approach was to consider convolutional neural networks.

We would like to extend our gratitude to Anders Stålring, our supervisor at Precise Biometrics, Johan Windmark at Precise Biometrics and Magnus Oskarsson, our supervisor from LTH for providing ideas and feedback regarding our research and the establishment of this report.

Table of contents

1 Introduction **1**

1.1 Overview of Biometrics 1

1.2 Fingerprint Recognition 2

1.2.1 Brief history 2

1.2.2 Methods 3

1.2.3 Difficulties 5

1.3 Why Convolutional Neural Networks? 6

2 Problem formulation **8**

3 Convolutional Neural Networks **9**

3.1 Convolutional layer 9

3.2 Non-linear activation functions 14

3.3 Pooling layer 15

3.4 Fully connected layer 17

4 Siamese networks **18**

4.1 Architecture 18

4.2 Contrastive loss 19

5 Triplet networks **21**

5.1 Architecture 21

5.2 Triplet loss 22

5.3 Triplet selection 23

6 Inception networks **26**

6.1 Architecture 26

7 Capsule Networks **27**

7.1 Dynamic routing 29

7.1.1 Routing illustration 32

7.2 Loss function 35

7.2.1 Contrastive loss for capsules 36

8 Training Neural Networks **37**

8.1 Regularization 37

8.2 Batch normalization 37

8.3 Dropout 39

9 Method **41**

9.1 Training methodology 41

9.2 Data 42

9.3 Inference for trained models 44

9.4 Evaluation metrics 44

10 Results **47**

10.1 Siamese network 47

10.2 Triplet network 52

10.3 Inception network . . . 56
10.4 Capsule network . . . 60
10.5 Model comparison . . . 63

11 Discussion **66**
11.1 Decision errors - examples . . . 66
11.2 Potential issues with CNN . . . 69
11.3 Features on different scales . . . 70
11.4 Triplet network . . . 70
11.5 Problems with capsule networks . . . 70
11.6 Future work . . . 71
11.6.1 Improvements on Capsules . . . 72
11.6.2 Triplet network - improvement . . . 73
11.6.3 Patch based matching . . . 73

12 Conclusion **74**

References **75**

13 Appendix **78**
13.1 Inception modules . . . 78
13.2 Loss functions for capsule networks . . . 81
13.2.1 Triplet loss for capsules . . . 81
13.2.2 Scaled Contrastive loss for capsules . . . 81
13.2.3 Agreement loss . . . 82

1 Introduction

1.1 Overview of Biometrics

Biometrics refers to the science of determining the identity of an individual based on her physical, behavioural or chemical characteristics [1]. Examples of attributes include fingerprints, face, voice and odor, where the first two are physical traits, voice is a behavioural characteristic and odor a chemical attribute.

A biometric system establishes the identity of an individual by performing a series of operations. On a global level a system utilizing biometrics constitutes four components [1]:

1 **Sensor** - a physical device used for acquisition of input data to the biometric system. Depending on the application the functionality of the device may differ, for instance images of fingerprints can be obtained using optical sensors while a sample of voice is acquired by an audio-based device.

2 **Feature extraction** - pertinent features of the input data are extracted by some algorithm(s).

3 **Database** - a database contains data of templates of similar type to the input data e.g. a fingerprint database consists of data from fingerprints of several individuals. Raw data is often not stored in a database for security reasons, instead the data is stored in another format.

4 **Decision making** - features of input templates are compared to relevant stored data in the database component to produce a score for decision making.

Depending on the application a biometric system can be utilized for **verification** or **identification** [1]. A verification system associates each individual with a unique identifier, such as PIN or user name and verifies the identity of an individual by comparing data of input template with templates in the database associated with the identifier of that person. In contrast to verification, identification systems compare data of acquired templates with all templates stored in the database to confirm the identity of a person.

Since there are many human characteristics the selection of useful attributes is crucial for the implementation of successful biometric systems. In order for a trait to be considered useful for biometric applications, it is reasonable to assume that it satisfies some pertinent conditions. In [2] several such conditions are presented, we list a subset of these:

1 **Universality** - Every individual should possess the trait.

2 **Uniqueness** - There should exist sufficient variation in the trait between individuals.

3 **Permanence** - The trait should be invariant over a long period of time, in the sense that the features extracted in a biometric system remains similar during that period.

4 **Measurability** - This refers to the possibility of acquiring and processing the trait. It should be simple and convenient to obtain samples of the trait. In cases of poor quality samples it should be possible to improve quality with various preprocessing techniques.

One characteristic that satisfies the above conditions to some degree is fingerprints, which will be discussed further in the coming introductory sections.

1.2 Fingerprint Recognition

1.2.1 Brief history

Scientific study of fingerprints began in late sixteenth century [3],[4]. In 1788 the anatomic formation of fingerprints was described in detail, where various fingerprint ridge attributes were identified and characterized [5]. The first fingerprint classification scheme was introduced in 1823, where fingers were classified in nine different categories depending on ridge configuration [5]. Uniqueness of fingerprints was proposed by Henry Fauld in 1880 based on empirical observations [6]. This fact is one of the main reasons that fingerprints are suitable traits in biometrics. Minutiae features for fingerprint matching were introduced in late nineteenth century [4]. By the start of the twentieth century fingerprint recognition was acknowledged as a valid personal identification method and well established in the area of forensics [7]. Since then emphasis have been put on developing automatic fingerprint

recognition technology and the field of application have expanded to civilian applications such as authentication on PC or mobile phones.

1.2.2 Methods

In this section we introduce terminology related to fingerprints and briefly summarize common methods used in fingerprint recognition systems. We will use the term recognition in cases where it is not specified if a biometric system is designed for verification or identification.

A fingerprint consists of **ridges** and **valleys** as illustrated in Figure 1, where the ridges correspond to dark areas and valleys are bright.

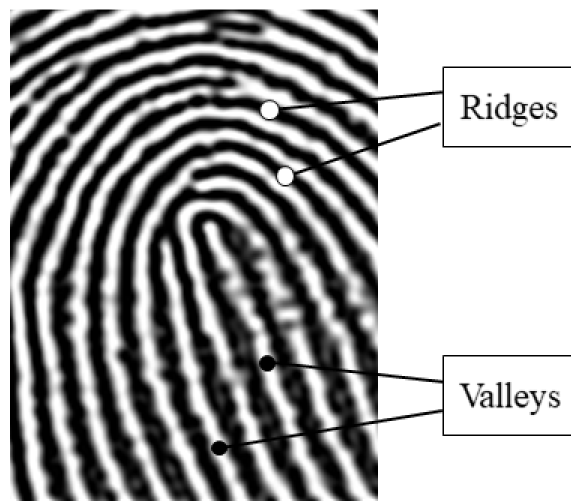

Figure 1: Illustration of ridges and valleys in a fingerprint. Ridge lines are black while valleys are white.

Minutiae are points where discontinuities occur in ridges [8]. These points constitute important features for some fingerprint recognition methods. Figure 2 illustrates two types of minutiae termination and bifurcation, the former is marked with white dots in the figure. Termination points are classified as points where ridges end and bifurcations are identified as points where ridges diverge in two parts [8]. There are other minutiae points but termination and bifurcation are the two most common minutiae points used for fingerprint matching.

Figure 2: Illustration of two types of minutiae, termination and bifurcation. Termination points are marked with white dots and bifurcation points with gray dots.

When it comes to common algorithms utilized for fingerprint matching, they can be categorized in three groups: correlation-based matching, minutiae-based matching and ridge feature-based matching [8].

Correlation-based algorithms use cross-correlation as a measure of similarity between two image pairs [8]. These algorithms use global and/or local cross-correlation techniques. Ridge feature-based techniques refers to methods utilizing features belonging to ridge patterns [6]. Both correlation-based and minutiae-based algorithms can be considered subsets of ridge feature-based algorithms, since they are based on information from ridges.

Some frequently used algorithms for matching are minutiae-based [6]. Such methods take sets of minutiae points from two images as input and determines the alignment between the sets that maximizes the number of minutiae pairings [6]. This is often referred to as the minutiae matching problem. A mathematical formulation of the problem and various solution methods are

summarized in [6]. Since minutiae-based matching algorithms have sets of minutiae points as input it relies on an efficient minutiae extractor. Minutiae extraction can be performed in various ways, most methods uses a binarization process to convert input images to a gray-scale images [6]. The output is then passed through a thinning stage, where the thickness of ridge lines is converted to 1 pixel [6]. This results in a representation in which minutiae points are easy to identify.

1.2.3 Difficulties

Fingerprint recognition has been extensively studied and many high-performing algorithms for fingerprint matching exist. In the FVC (fingerprint verification competition) website there are published algorithms achieving under 1% false reject rate for false acceptance rate of 10^{-4} on their benchmark tests [9]. Despite the presence of algorithms providing good performance, fingerprint recognition is in general a difficult problem due to high intra-class variations and occurrence of fingerprints with small inter-class variations [6]. Factors contributing to large intra-class variations include [6]:

1 **Translation** - Fingers placed at different locations in a sensor result in translated images.

2 **Rotation** - Orientation of fingers may differ between acquisitions of fingerprints.

3 **Partial overlap** - This is an implication of translation and rotation differences between images of fingerprints.

4 **Pressure** - Captured ridge structures in a fingerprint depend on the pressure on the acquisition device. Non-uniform pressure results in noisy images.

5 **Skin condition** - Despite empirical observations supporting uniqueness of fingerprints [6] changes in skin condition can occur when fingers are subjected to damage, environmental causes such as temperature etc.

1.3 Why Convolutional Neural Networks?

One of the first convolutional neural networks (CNN) was the LeNet5 in 1998 [10]. They realized that image features can be located across the entire image and that convolutional filters are great at extracting similar feature at multiple locations with a low number of parameters. But it was not until 2012 that the field really picked up speed when a CNN called AlexNet won by a large margin in the ImageNet Large-Scale Visual Recognition Challenge [11]. At this point the computational power had increased greatly since 1998 and computations could be parallelized to a wide extent on graphics cards (GPU). As the training of CNNs are very computationally heavy the use of CNNs has been able to increase as with the growth in computer power. Nowadays our smartphones have enough computing power to make good use of CNNs, though they only run an already trained network so they don't need to go through the cumbersome training. For instance Google's smartphone Pixel uses a CNN to do background/foreground segmentation to artificially create blurred background to mimic photos taken with expensive cameras such as DSLRs [12].

The advantage of using CNNs is that they erase the need of manually handcrafted feature extraction tools. A CNN will instead find the optimal feature extraction by only looking at the data. This way it's possible to find new features that would have been hard to find using handcrafted methods. This also poses a few problems as the data fed to the network has to be of great quality. Neither does the network get to use previous knowledge of the problem which could have been found during an attempt to design handcrafted feature extraction.

A difficult part of doing fingerprint matching using a CNN are the intra-class variation mentioned above, Section 1.2.3. By design convolutions are made on a grid structure which creates problem when images rotate as then the content of each cell changes, see Figure 3. This is a common problem with CNNs and significantly degrades the accuracy of models as documented in [13].

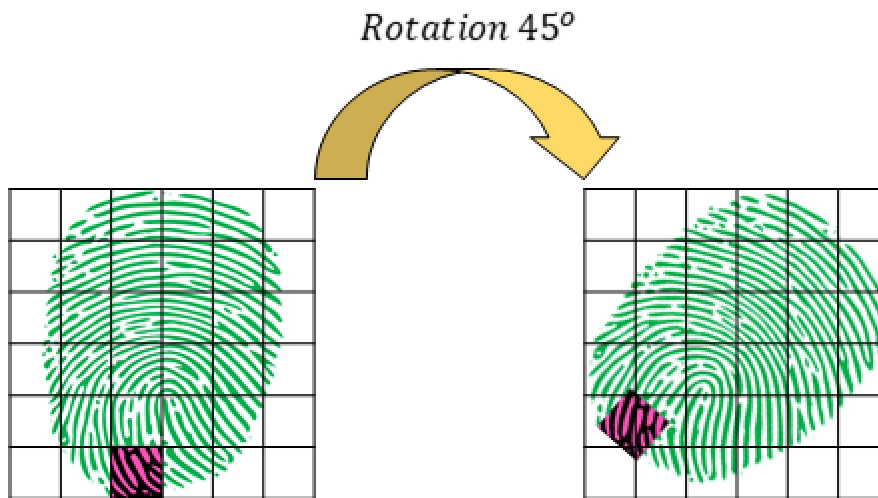

Figure 3: Example of changes in grid content when a rotation is applied to an image. Here it is possible to see that different parts of the fingerprint end up in different orientations as well as different cells in the two cases. The pink/black area is completely encapsulated in a cell in the left image but when rotated 45 degrees clockwise this part ends up partially included in six cells.

2 Problem formulation

The subject of this thesis is to investigate the potential of applying convolutional neural networks to solve a fingerprint matching problem, which can be formulated as:

P Given a set consisting of pairs of fingerprint images, determine for all elements of the set whether they correspond to **similar** fingerprints or not.

Note that the word "similar" is not defined in the formulation of problem P. Similar fingerprints could for instance be images of the same finger from a single person or perhaps one has to impose additional conditions in order to classify the images as similar. Consider e.g. the case of two images of the same finger from a specific person, but the images depict different parts of the finger. Then it might be more suitable to define this as a dissimilar pair of images. Our definition of similarity between fingerprint images will be given later (see definition 9.1 in Section 9.2).

3 Convolutional Neural Networks

This section describes the various components and their roles in a convolutional neural network, since these will be used frequently in our network architectures. For those already familiar with the building blocks of a CNN this section can be used for repetition or skipped entirely.

A general CNN consist of several layers, starting with an input layer followed by hidden layers and ending with an output layer. The input layer is simply where the input enters the network and serves as an input to the following layer. Each hidden layer consists of different components, which will be described further below, that operates on the input of the previous layer and produces an output to the next layer. The final output of a convolutional neural network varies depending on the application. In e.g. classification tasks the output is class labels, but for other problems it could for instance be feature vectors.

3.1 Convolutional layer

A convolutional layer consists of convolutional filters (a.k.a. kernels) that operate on the input to produce an output, often referred to as **feature maps** [14]. For simplicity we will consider a convolutional layer with one filter and one channel The crucial point is to understand the concept of convolutional filters, extending it to an arbitrary number of filters and channels will then be trivial. Consider an input $\mathbf{I}$ of dimensions $m \times n$ and a convolutional filter $\mathbf{W}$ of size $k \times l$. The output $\mathbf{O}$ of this convolutional layer is given by a two dimensional convolution (actually cross-correlation [14]):

$$\mathbf{O}(x, y) = \sum_{i,j} \mathbf{I}(i, j)\mathbf{W}(i - x, j - y). \tag{1}$$

To gain intuition on the convolution described by (1) we consider a small example illustrated in Figures 4 and 5. The input to the convolutional layer is in this case a 5×5 matrix and the convolutional filter is a 3×3 matrix. The convolution process can be described as sliding the filter across the image from left to right and top to bottom. In between the slides the filter is multiplied with the input matrix elementwise and then all elements are summed to produce a scalar output. Note that only elements of the input and filter that overlap are multiplied, in Figure 4 the overlap is indicated by

the green cells. The area of the input that overlaps with the filter is called **receptive field**. Figure 4 illustrates the first convolution step and Figure 5 the second step. In this example the **stride** used is 1 in both dimensions of the input. This implies that during the convolution the filter will slide across the image with one step in horizontal direction as long as the receptive field is not outside the input. Once it reaches the right edge of the input it starts at the left edge of the input but one step down in vertical direction. The total output will be a 3×3 matrix.

0	1	2	10	10
1	2	10	10	10
2	10	10	10	2
10	10	10	2	1
10	10	2	1	0

Figure 4: Example of two dimensional convolution where the input is the 5×5 matrix and the convolutional filter has the dimensions 3×3. The figure illustrates the first step in the convolution. The receptive field is indicated by the green area. Integers in the cells are elements of the input.

0	1	2	10	10
1	2	10	10	10
2	10	10	10	2
10	10	10	2	1
10	10	2	1	0

Figure 5: Example of two dimensional convolution where the input is the 5×5 matrix and the convolutional filter has the dimensions 3×3. The figure illustrates the second step in the convolution. The receptive field is indicated by the green area. Integers in the cells are elements of the input.

Notice that after the convolution step in the above example the output dimensions are smaller than the original input dimensions. In order to be able to maintain the dimensions of the input, **padding** is often utilized. One way of padding is to fill the original input with zeros outside its boundary. For example if one was to add a layer of zeros surrounding the matrix in Figure 4 the result would be the matrix illustrated in Figure 6. Convolving this matrix with the same filter yields an output with the same dimensions as the input. When one adds a layer of zeros we say that the padding is 1 in both input dimensions.

0	0	0	0	0	0	0
0	0	1	2	10	10	0
0	1	2	10	10	10	0
0	2	10	10	10	2	0
0	10	10	10	2	1	0
0	10	10	2	1	0	0
0	0	0	0	0	0	0

Figure 6: Padded version of the input matrix in Figure 4 with a padding of 1 in both input dimensions.

In general given an input $\mathbf{I}$ of dimensions $m \times n$, a filter $\mathbf{W}$ of size $k \times l$, a stride $\mathbf{S} = [S_1, S_2]$ and padding $\mathbf{P} = [P_1, P_2]$ the output $\mathbf{O}$ will have the dimensions

$$O_1 = \frac{m - k + 2P_1}{S_1} + 1, \tag{2}$$

$$O_2 = \frac{n - l + 2P_2}{S_2} + 1. \tag{3}$$

If the inputs and convolutional filters are square matrices and furthermore the stride and padding are the same in both dimensions the output will be a square matrix with size given by

$$O_{dim} = \frac{m - k + 2P}{S} + 1. \tag{4}$$

where S_1, S_2 are the strides in horizontal and vertical direction respectively, same applies for the elements of $\mathbf{P}$. For simplicity we assume that (4) holds

for the remainder of this report, when it does not hold it will be explicitly specified. The formulas that hold for the square case have analogues for the non-square case.

Now that we are familiar with the convolution process in a convolutional layer, generalizing it to inputs with an arbitrary amount of channels and convolutional filters is simple. Assume that the input, **I**, to a convolutional layer has the dimensions $m \times m \times c$. Then each filter in a convolutional layer must have c channels. The convolution process remains the same, a filter, now with several channels, slides across the input and between each slide elementwise multiplication followed by summation is performed. If the convolutional layer contains f convolutional filters the convolution process is repeated for all f filters. The output will therefore have f channels and first and second dimensions given by (4).

When constructing our networks we will use different padding in convolutional layers and we will therefore introduce some terminology related to padding. Assuming a stride $S = 1$, a padding P that satisfies (4) with l.h.s equal to m will be called **same padding**. If $P = 0$ and the output size is given by

$$O_{dim} = \left\lfloor \frac{m-k}{S} \right\rfloor + 1, \tag{5}$$

we call the corresponding padding **valid padding**.

The idea behind convolutional filters is that they should function as local feature identifiers. Kernels that represent e.g. edges should detect areas of the input where edges are present when convolved with the input, i.e. the corresponding output should be large for such areas of the input. Conversely the output should be small when convolving the filter with an area of the input where no edge is present. For a practical example consider the input in Figure 4. Let us assume that it represents an edge and that the convolutional filter is given by the matrix in Figure 7. The convolution step illustrated in Figure 4 would produce an output equal to 280. If the filter is placed at the center of the input the convolution output will be 700. This indicates a stronger presence of the edge represented by the filter at the center of the input than in the upper left corner. By sliding the filter across the image, features will be identified over the entire input.

0	10	10
10	10	10
10	10	0

Figure 7: Elements of the convolutional filter used in Figures 4 and 5.

3.2 Non-linear activation functions

Convolution is a linear operation, thus convolutional layers can only model linear dependence. The main purpose of non-linear activation functions is to introduce non-linearity in neural networks. An activation function operates on each element of its input. There are many such functions and we will list the ones that were utilized in this project.

One activation function that has proved to be successful in training deep neural networks for supervised tasks [15] is the rectified linear unit (relu) given by

$$y(x) = \max(0, x). \tag{6}$$

The relu activation functions introduces sparsity in the network which is a desirable characteristics since it implies computational advantages, such as accelerated training. Another advantage of relu is that it is unbounded for positive arguments, which mitigates the vanishing gradients problem that often occurs for saturated activation functions. However there are also drawbacks with this activation function. A common problem is that of dying relus which implies that the function will output zeros for most inputs, thus introducing the vanishing gradients problem [16].

In order to deal with the potential problem of dying relus a commonly used activation function is the leaky rectified linear unit (leaky relu) [16] which is

defined by

$$y(x) = \begin{cases} x & \text{, if } x > 0, \\ ax & \text{, otherwise,} \end{cases} \tag{7}$$

where a is a small positive number.

3.3 Pooling layer

A pooling layer reduces the dimensions of its input by performing various operations on the elements of its input. The operations performed varies depending on the type of pooling used. Pooling works similar to convolutional filters. A window slides across each channel of the input and between each slide it operates on the elements of the receptive field. However there are some key differences between a pooling layer and a convolutional layer. In a pooling layer one can think of the window that slides across the input as an imaginary matrix, in the sense that it does not exist in memory, it only operates on the elements within the receptive field.

Max pooling is a type of pooling used widely in neural networks. It simply selects the largest element within the receptive field of the input [14]. As an example consider the matrix in Figure 8. Applying max pooling with a window of size 2 and a stride of 2 will produce the output in Figure 9. Figure 8 also illustrates all receptive fields as cells of a specific color during the max pooling operation. Figure 9 illustrates the corresponding output, which has the same color coding. The width and height of the output from a max pool layer can be computed according to (4) or (5), while the number of channels is the same as the input.

12	20	30	0
8	12	2	0
34	70	37	4
112	100	25	12

Figure 8: Input for max pooling with window size 2 and stride 2

20	30
112	37

Figure 9: Max pooling output with input given in Figure 8 using window of size 2 and stride 2

Pooling layers serve many purposes. One of them is to reduce the total number of parameters in a network. This is trivial to see since a max pooling layer decreases the spatial resolution of its input, thus subsequent layers will use less parameters compared to the case where the pooling layer is absent. A consequence of this is that pooling decreases the likelihood of overfitting the trained model by reducing the model complexity.

Moreover pooling also introduces local translation and positional invariance [14]. To see this one could imagine a translated version of the input to a max pooling layer such that the receptive field in e.g. the upper left corner still contains the same largest element as the corresponding receptive field in the original input. The max pooling operation will output the same element in these cases. It follows that the degree of invariance depends on the size of the pooling window used. Local translation invariance is a desired property for a network since it implies that the model will generalize well for small translation of the input.

Including a max pooling layer in a network does have disadvantages. One crucial fact to note is the decrease of spatial resolution when applying a max pooling operation. A network consisting of many max pooling layers with large strides might lose too much spatial information regarding features in the input. One approach to solving this issue is simply reducing the strides of pooling layers and/or decreasing the number of such layers. It is also possible to skip the pooling layers entirely, but the network then loses the advantages

of pooling layers. One of the network architectures used in this project does not utilize any pooling (see Section 7).

3.4 Fully connected layer

A fully connected layer (a.k.a. dense layer) takes a vector $\mathbf{x}$ as input and outputs another vector $\mathbf{y}$, mathematically it is defined by a matrix-vector multiplication:

$$\mathbf{y} = \mathbf{A}\mathbf{x} \tag{8}$$

where $\mathbf{x} \in \mathbb{R}^n$ and $\mathbf{A} \in \mathbb{R}^{m \times n}$ is a trainable weight matrix. The name "fully connected" comes from the fact that each component of the output is a weighted sum of all components of the input.

The fully connected layer is used to aggregate all information in the input, in a CNN the input can be interpreted as a vector of features. For a classification task a fully connected layer will then combine features into objects/classes. The weighted sum corresponds to determining what features correlates the most to a particular object/class.

4 Siamese networks

4.1 Architecture

When solving matching tasks with convolutional neural networks, a commonly used architecture is the siamese network architecture. Figures 10 and 11 illustrate two general siamese networks for training and inference respectively. The two left-most blocks are input layers. Blocks labelled with CNN are convolutional neural networks, for siamese networks the output of CNN blocks are feature vectors. Determining whether two inputs correspond to a match is done in the decision layer, which outputs an answer ”match” or ”no match”. In a siamese network the convolutional neural networks have the same architecture, hence the name siamese network. The convolutional networks can even share weights, which is indicated by the bidirectional arrow in Figures 10 and 11. A siamese network is intended to function as a feature descriptor, features for both inputs are computed and then compared to determine whether there is a match or not. One benefit of having such an architecture from a computational perspective is that the convolutional blocks can be run in parallel.

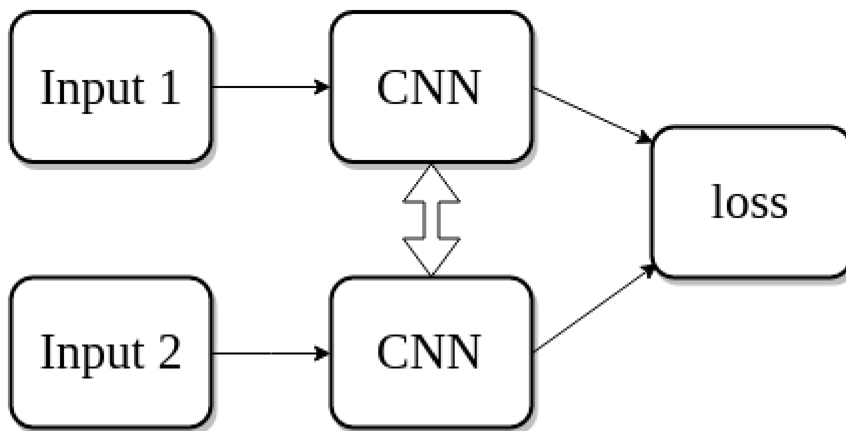

Figure 10: General siamese network architecture for training. The blocks with label CNN are convolutional neural networks. Black arrows indicate direction of flow of the output from each block. Potential weight sharing between the CNN blocks is indicated by the bidirectional arrow.

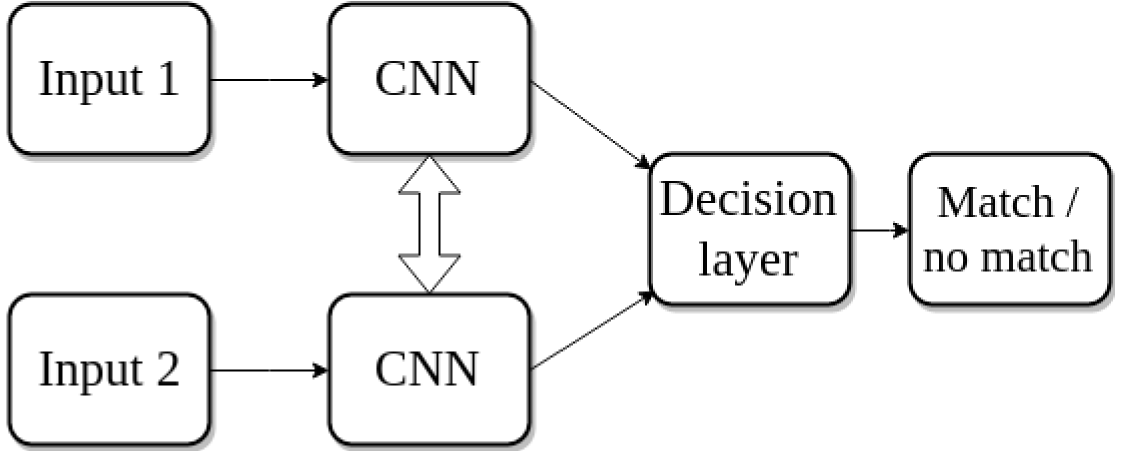

Figure 11: General siamese network architecture. The blocks with label CNN are convolutional neural networks. Black arrows indicate direction of flow of the output from each block. Potential weight sharing between the CNN blocks is indicated by the bidirectional arrow.

4.2 Contrastive loss

In order to train a model to match inputs one has to minimize a suitable loss function. The loss function we will be using is the margin based contrastive loss function [17]. Before we give the definition of this function some notation have to be introduced. Let $\mathcal{I}$ be the finite set of training pair samples of cardinality N given by

$$\mathcal{I} = \{(\mathbf{i}_1, \mathbf{i}_2)^1, (\mathbf{i}_1, \mathbf{i}_2)^2 \ldots, (\mathbf{i}_1, \mathbf{i}_2)^N\}, \tag{9}$$

where N is a positive integer, $(\mathbf{i}_1, \mathbf{i}_2)^k$ is sample pair k and $\mathbf{i}_1, \mathbf{i}_2 \in \mathbb{R}^D$. Moreover let $\mathbf{f}_\Omega : \mathbb{R}^D \to \mathbb{R}^d$ be a parametric function. Introduce the label l_k defined by

$$l_k \coloneqq \begin{cases} 1 & \text{, if } (\mathbf{i}_1, \mathbf{i}_2)^k \text{ correspond to a similar pair,} \\ 0 & \text{, otherwise.} \end{cases} \tag{10}$$

Define $D_\Omega^k \coloneqq \|\mathbf{f}_\Omega(\mathbf{i}_1) - \mathbf{f}_\Omega(\mathbf{i}_2)\|_2$ where $\mathbf{i}_1, \mathbf{i}_2 \in (\mathbf{i}_1, \mathbf{i}_2)^k$. The contrastive loss function is then given by

$$L(\Omega, \mathcal{I}, m) = \frac{1}{N} \sum_{k=1}^{N} \frac{1}{2} l_k (D_\Omega^k)^2 + \frac{1}{2}(1 - l_k) \left[\max(0,\ m - D_\Omega^k)\right]^2, \tag{11}$$

where $m > 0$ is the **margin**.

The idea behind (11) is to learn a function $\mathbf{f}_\Omega$ such that D_Ω^k is small for all k where $l_k = 1$ and larger for all k such that $l_k = 0$ [17]. In other words we want to learn a function that maps similar inputs to points close to each other and dissimilar inputs to points distant from each other. This can also be seen directly in (11). The first term in the sum will be larger for similar pairs that are far from each other in output space. We also observe that if the second term is non-zero for dissimilar pairs it contributes less to the loss function if the euclidean distance between the pairs in output space is large. Another important observation is that dissimilar pairs will only contribute to the loss if the euclidean distance between them in output space is less than the margin m. The importance of this fact will be highlighted in the training of our siamese networks.

5 Triplet networks

5.1 Architecture

An alternative to using siamese networks is the so called Triplet network. In Schroff, Kalenichenko and Philbin's paper on face recognition the Triplet network approach exceeded state of art performance in 2015 [18]. As with a siamese network, triplet networks reuses the weights of a single CNN for all three image batches involved in the network, see Figure 12.

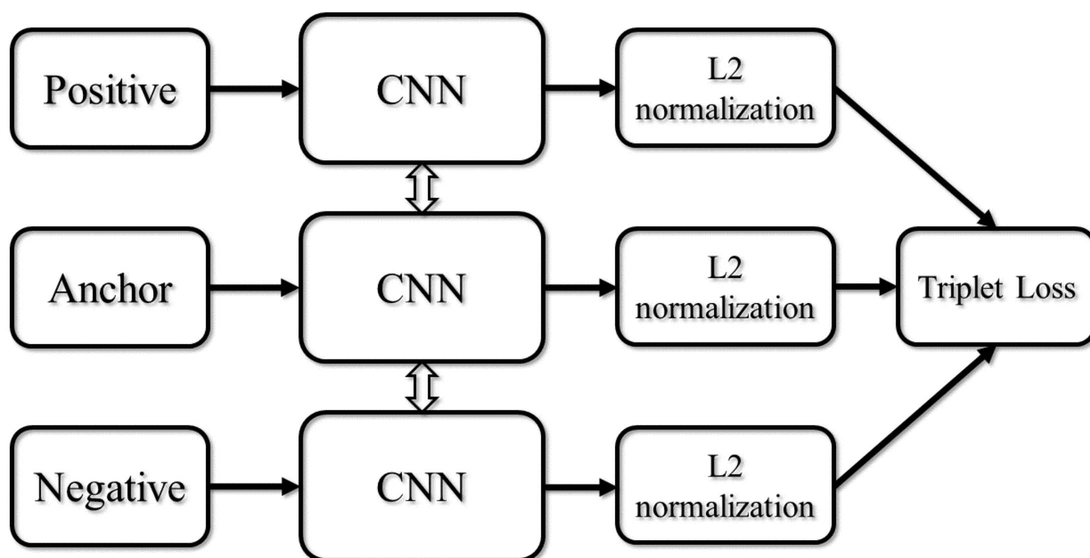

Figure 12: Triplet network architecture. The bidirectional arrows signify shared weights between CNN blocks. This architecture is used during training. At testing and validation time a siamese network is used with the same weights and the triplet loss block is exchanged with a decision layer. This layer typically consist of a Euclidean distance measure between the outputs and a threshold determining match/no match.

When training a triplet network the input to the network is divided into the three categories (anchor, positive and negative) which are all fed simultaneously to the network. Each anchor image has a positive and negative list of images associated with it, where the positive contains images matching to the anchor and the negative contains images dissimilar to the anchor.

5.2 Triplet loss

To achieve a good separation between matching and non matching images the so called Triplet Loss [18] is used

$$L = \sum_{k=1}^{N} \max\left(0,\ \|\mathbf{f}_\Omega(\mathbf{i}_k^a) - \mathbf{f}_\Omega(\mathbf{i}_k^p)\|_2^2 - \|\mathbf{f}_\Omega(\mathbf{i}_k^a) - \mathbf{f}_\Omega(\mathbf{i}_k^n)\|_2^2 + \alpha\right), \tag{12}$$

where $\mathbf{i}_k$ is the k:th image in a batch, the superscripts p, a, n denote positive, anchor and negative categories respectively. The function $\mathbf{f}_\Omega$ is the normalized output from the network $\mathbf{f}_\Omega(\mathbf{i}) \in \mathbb{R}^d$ with $\|\mathbf{f}_\Omega(\mathbf{i})\|_2 = 1$, hence the images are mapped to the d-dimensional hypersphere. N is the total number of triplets in a training batch. The α parameter is a hyperparameter functioning as a margin between the positive sample and the easy negatives in Figure 14. In this figure the semi hard and hard negatives will have a distance close enough to the anchor to produce a positive loss. Due to the max term in (12) easy negatives will not contribute to the loss function. During training the semi-hard and hard negatives will be "pushed away" from the anchor whilst the distance to the positive is minimized, see illustration in Figure 13.

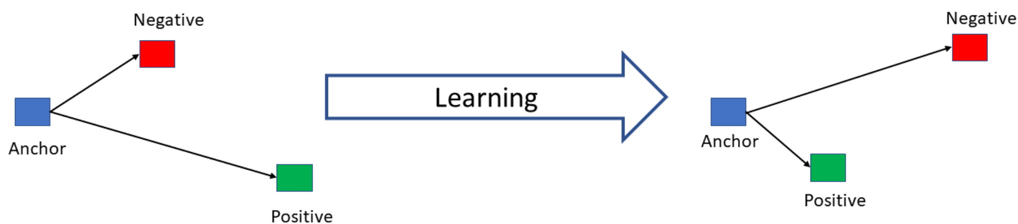

Figure 13: Illustrates how the network learns to separate the negative and positive images during training. The triplet loss will enforce the network to minimize the distance between anchors and positive images, in a Euclidean space, while maximizing the distance between the anchor and the negative images.

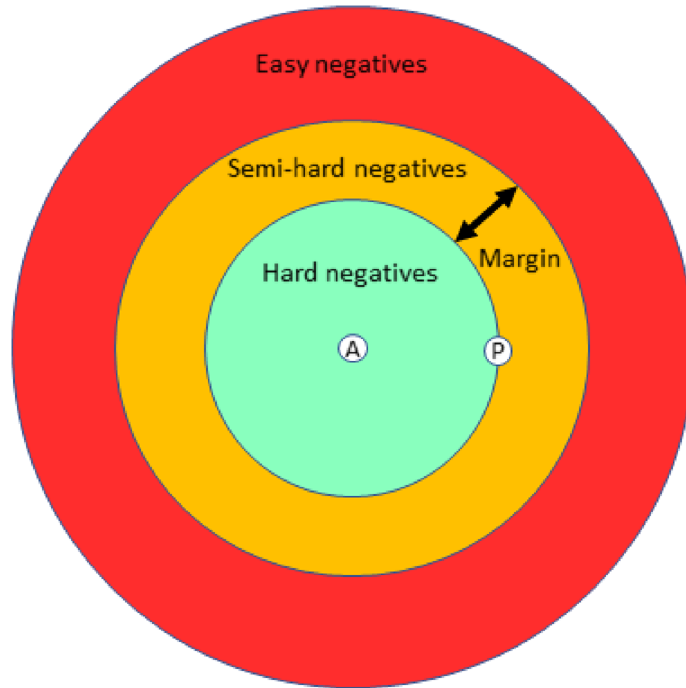

Figure 14: Interpretation of how the margin α effects the loss. The 'A' represent the anchor and 'P' the positive image. The margin defines a buffer zone between the positive image and the easy negative area. If a negative image ends up in the easy area the contribution to the triplet loss will be 0, hence the margin becomes essential to repel negative images further away from the anchor than the positive ones. It's worth noting that the boundary between the easy negative zone and the semi-hard negative is dependent upon the distance between the current positive image and the anchor. Thus the contribution to the loss function is highly dependent on how the whole triplet of images is chosen.

5.3 Triplet selection

As discussed in the caption of Figure 14 the choice of triplets greatly impacts the loss function and thus also the training, hence it's important how the data is selected during training to achieve faster convergence. To speed up convergence it's desired to select hard positives $\mathbf{i}_k^p$ such that

$$\underset{\mathbf{i}_k^p}{\operatorname{argmax}} \|\mathbf{f}_\Omega(\mathbf{i}_k^a) - \mathbf{f}_\Omega(\mathbf{i}_k^p)\|_2^2$$

and hard negatives $\mathbf{i}_k^n$ such that

$$\operatorname*{argmin}_{\mathbf{i}_k^n} \left\| \mathbf{f}_\Omega(\mathbf{i}_k^a) - \mathbf{f}_\Omega(\mathbf{i}_k^n) \right\|_2^2 .$$

As the data sets used in deep learning usually are very big it's impractical to compute argmax and argmin over the whole data set. To tackle this problem there are two obvious choices discussed in [18]:

- Sample a subset of the data set every n steps and calculate the argmax and argmin using the current network.
- Generate triplets online by calculating argmax and argmin in a mini-batch and use the hard positives and hard negatives from this mini-batch for training.

In this thesis a variation of the former approach has been used. Where the five hardest positives and three hardest negatives for each anchor have been used to increase the difficulty of the training set. A visualization of the hard negative and positives is given in Figure 15. In the paper [18] they used online triplet generation and used all positive data and only semi-hard negatives, defined as $\mathbf{i}_k^n$ such that

$$\left\| \mathbf{f}_\Omega(\mathbf{i}_k^a) - \mathbf{f}_\Omega(\mathbf{i}_k^p) \right\|_2^2 < \left\| \mathbf{f}_\Omega(\mathbf{i}_k^a) - \mathbf{f}_\Omega(\mathbf{i}_k^n) \right\|_2^2 .$$

These $\mathbf{i}_k^n$ are mapped to the semi-hard region defined by the margin, depicted in Figure 14.

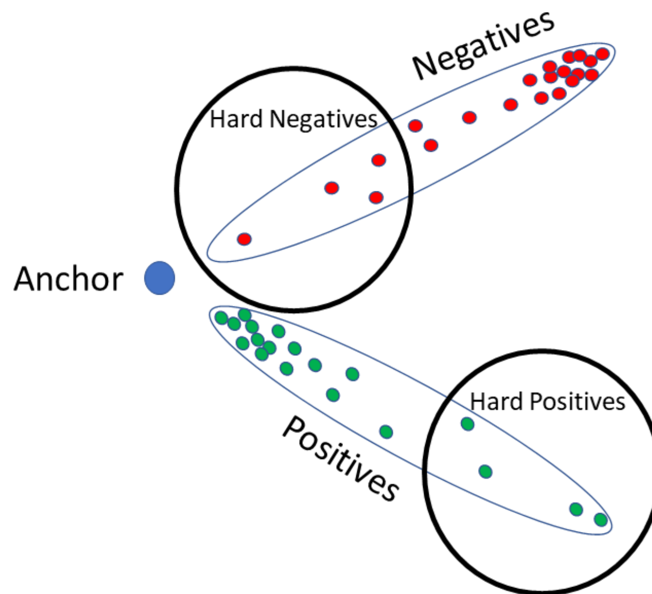

Figure 15: Illustration of which images to select when increasing the difficulty. Using hard positives and negatives for training will repel the hard negatives away and attract the hard positives. In the ideal case also the already close positives and distant negatives will remain well clustered but since the training advances on these hard cases chances are that some of the easy samples will move in an undesirable direction. Hence a new evaluation of hard negatives and positives is needed and new images will be selected for training.

6 Inception networks

The inception network architecture was first introduced in 2014 where it achieved new state of the art performance for classification and detection in that years ImageNet Large-Scale Visual Recognition Challenge [19]. Inception networks were designed with the purpose of optimizing utilization of computational resources within the network by approximating sparse structures within a network with dense components [19], since numerical computations with dense data structures are better optimized than computations with non-uniform sparse data structures [19].

6.1 Architecture

Inception networks contain components called **inception modules**, the modules in turn consist of basic building blocks in a regular convolutional neural network, which are described in Section 3. In Section 13.1 some inception modules are illustrated. Unlike a regular CNN, which consist of a series of convolutional components stacked on top of each other, an inception module can have convolutional components next to each other i.e. it can contain branches. One detail that is not specified directly in Figures 40 - 42 in Section 13.1 is that the input to the concatenation block should have the same width and height, however the number of channels may differ. The addition of branches allows an inception module to consider features of various scales by including convolutional layers with different filter sizes in a layer of an inception module. By design an inception module component is easily integrated into a conventional CNN simply by stacking it on top of the network. Another important observation is that the convolutional layers in the first layer of an inception module usually have 1×1 convolutional filters (see e.g. Figure 41). The reason for this is that larger filters in the first layer of an inception module adds more parameters to the network. Thus if the inputs to inception modules have many channels the number of parameters will increase quickly. Adding 1×1 filters in the first layer of a module adds less parameters than larger filters and therefore works as a reduction [19].

7 Capsule Networks

Capsule networks, or popularly referred to as CapsNet, are a new variation to the well established convolutional neural networks discussed in Section 3. CapsNet was first introduced in Geoffrey Hinton, Sara Sabour and Nicholas Frosst's paper *Dynamic Routing Between Capsules* [20]. This novel approach tries to mimic the human vision system by putting parts of an image in relation to each other to form an opinion on the whole picture [21]. For example in Figure 16 when a human tries to answer if the letter 'B' is mirrored or not we impose an upright coordinate system and apply a rotation to the 'B' on the left in Figure 16 to also get it upright [22]. This is a linear transformation. Capsule networks tries to learn linear transformations through backpropagation in the routing step, see Section 7.1.

In computer graphics objects are placed in space through a linear transformation matrix which applies scaling, translation and rotation [23]. Capsule networks intend to perform the inverse process where the object's location in the image is known but the transformation matrix is not. By learning suitable transformation matrices the objects detected in an image (which the transformation is applied to) can be pieced together to form the whole picture. By the routing procedure parts that fit well together will be propagated forward in the network.

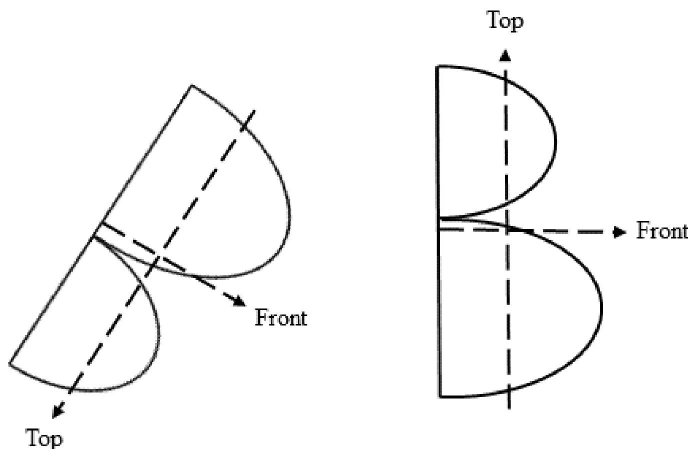

Figure 16: An easy example of how humans solves orientation; to find out whether the left 'B' is mirrored or not a human turns the 'B' to its upright position. Once there we conclude that it's mirrored.

When using an ordinary CNN it's common to use max pooling layers (see Section 3.3) to reduce dimensionality of the problem and still retain the most important information in the network. The issue with max pooling is that the precise location and orientation of objects in a picture get lost through the pooling layers, see Figure 17. Placing many max pooling layers in the architecture of a CNN causes an invariance in both orientation and translation. When considering matching of fingerprints the position and orientation between features are very important to get a reliable method for matching. Instead of invariance, equivariance is desired, that is if an image would be translated and rotated all features would change in the same way. Formally this can be expressed as

$$\mathbf{f}(\mathbf{g}(\mathbf{I})) = \mathbf{g}(\mathbf{f}(\mathbf{I})) \tag{13}$$

where $\mathbf{I}$ is an input image, $\mathbf{f}$ is a feature function and $\mathbf{g}$ is a distorting function. If $\mathbf{g}$ exist such that equation (13) is satisfied the feature function $\mathbf{f}$ is said to be equivariant to $\mathbf{g}$ [14]. The difference between CNN and CapsNet is in the output, ordinary CNNs output scalar values while CapsNet output vectors. The length of a CapsNet vector is interpreted as probability of existence of the feature represented by the vector and the orientation of the vector codes for the orientation of the feature. Hence CapsNet has an inherent way of handling orientation shifts. Thus a change of orientation in the input image as $\mathbf{g}(\mathbf{I})$, see equation (13), will cause a change in the orientation vector outputs of the CapsNet and ideally there exists a $\mathbf{g}$ such that equation (13) is satisfied. If that's the case the capsule network can handle orientational variance in an equivariant way.

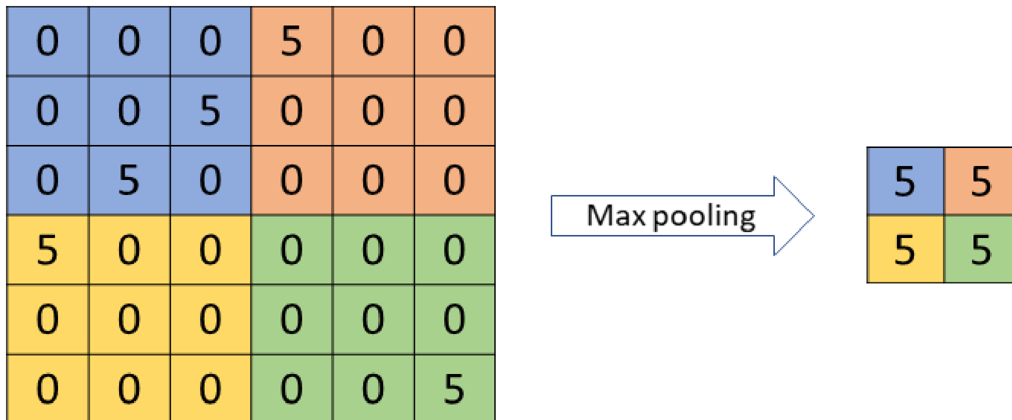

Figure 17: Example of max pooling with window size 3x3 and a stride of 3; pooling windows cover each colored patch. Even tough it's noticeable that there is a diagonal line consisting of 5s in the picture the output from the max pooling layer only consist of a homogeneous picture with 5s. Hence the orientation and separation between the 5 in the green patch and the 5s on the line is lost.

7.1 Dynamic routing

Dynamic routing is an iterative method for computing vector outputs in a capsule network. The capsule network builds initially on a regular CNN structure to extract useful features. After a few convolutional layers the output is sent into a **Convolutional Capsule layer**. This layer is an ordinary convolutional layer where the convolved output is reshaped into capsule modules each consisting of **capsules** on a grid structure, see Figure 18 and 19. A capsule is essentially a rebranding of a feature vector. The squashing function in equation (15) is applied to all capsules in this first Convolutional Capsule layer. It is all of these capsules in a layer that will be voting in the routing procedure to determine which entities in the next layer that should be activated.

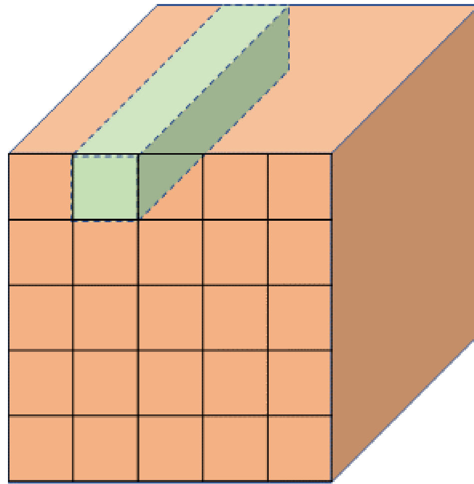

Figure 18: Illustration of a capsule element. The grid structure is corresponding to the original input image (parsed through convolutional layers) and the green rectangular parallelepiped represents a feature vector, i.e. a capsule, at one position in an image.

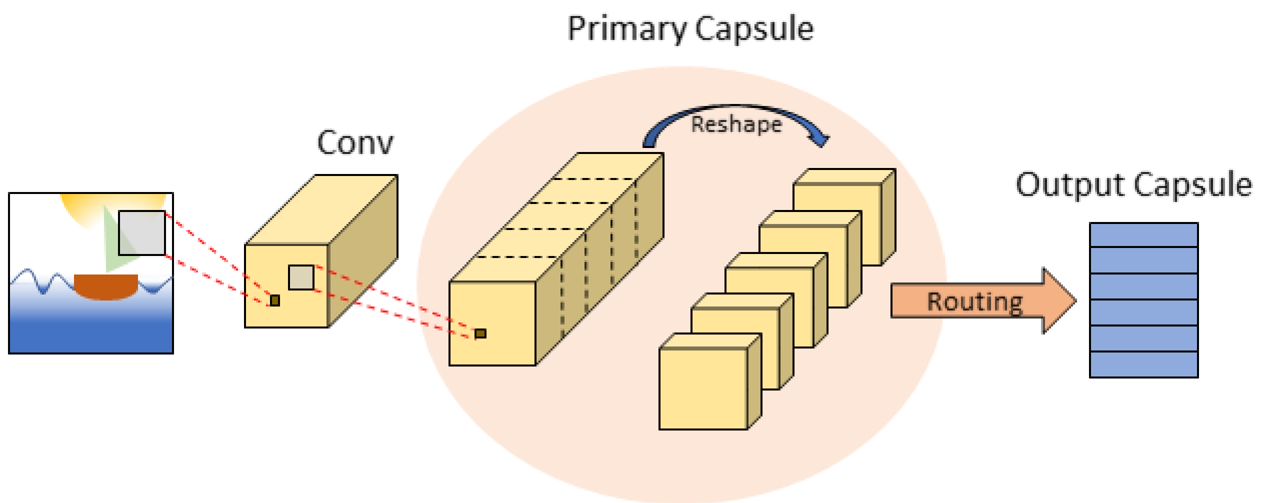

Figure 19: A simple capsule network where there is one convolutional layer before the first convolutional capsule layer here referred to as Primary Capsule. The Primary Capsule consists of an ordinary convolutional part whose output is reshaped into capsule modules. The network terminates with an Output Capsule layer which contains the predicted output vectors of the routing procedure.

The routing algorithm in this section can be found in paper [20]. Routing is an alternative to max pooling that forward output based on agreement. The agreement is found through clustering of transformed features using k-means algorithm [24]. Assume the layer l is a convolutional capsule layer. Denote its capsules by $\mathbf{u}_i$ and let $\mathbf{v}_j$ denote the outputs from layer l (which are capsules as well). Routing starts off by applying a trainable linear transformation matrix $\mathbf{W}_{ij}$ to a capsule $\mathbf{u}_i$

$$\hat{\mathbf{u}}_{j|i} = \mathbf{W}_{ij}\mathbf{u}_i, \tag{14}$$

where $\hat{\mathbf{u}}_{j|i}$ is referred to as a **prediction vector**.

To make the output vectors from a convolutional capsule layer more discriminant to existence of entities a non-linear "squashing" function is used

$$squash(\mathbf{x}) = \frac{\|\mathbf{x}\|_2^2}{1 + \|\mathbf{x}\|_2^2} \cdot \frac{\mathbf{x}}{\|\mathbf{x}\|_2}. \tag{15}$$

This function squashes short vectors to have lengths closer to zero and long vectors slightly below unit length. The outputs $\mathbf{v}_j$ from a convolutional capsule layer are then given by

$$\mathbf{v}_j = squash(\mathbf{s}_j). \tag{16}$$

Here $\mathbf{s}_j$ is the weighted sum of prediction vectors given by

$$\mathbf{s}_j = \sum_i c_{ij}\hat{\mathbf{u}}_{j|i}, \tag{17}$$

where c_{ij} is a **coupling coefficient** between capsule i and capsule j in the next layer. The coupling coefficient is calculated using the softmax function

$$c_{ij} = \frac{\exp(b_{ij})}{\sum_k \exp(b_{ik})}, \tag{18}$$

with initial **routing weights**, b_{ij}, which are set to zero at the beginning of every new routing procedure. Setting all b_{ij}:s to zero will result in a scaling of the prediction vectors with one over the number of capsules in layer $l+1$. To update the routing weights, i.e. moving the cluster means in a k-means algorithm, the scalar product between a prediction vector and the corresponding squashed output is computed according to

$$b_{ij} = b_{ij} + \mathbf{v}_j \cdot \hat{\mathbf{u}}_{j|i}. \tag{19}$$

With the updated routing weights the outputs from layer l, $\mathbf{v}_j$, can be computed iteratively through (18) and (17). This iterative process constitutes the dynamic routing algorithm. The routing algorithm in its entirety is summarized in pseudo code in algorithm 1.

Algorithm 1: Dynamic Routing

Input: Capsules $\mathbf{u}_i$, routing iterations r, layer number l, transformation matrices $\mathbf{W}_{ij}$
Output: $\mathbf{v}_j$

```
1  Procedure: Routing(u_i, r, l, W_ij)
2      for all capsules i and j: û_{j|i} ← W_ij u_i           // prediction vectors
3      for all capsules i in layer l and capsules j in layer (l+1): b_ij ← 0
4      for r iterations do
5          for all capsules i in layer l and capsules j in layer (l+1)
            compute c_ij using eq. (18)                     // coupling coefficients
6          for all capsules j in layer (l+1): s_j ← Σ_i c_ij û_{j|i}
                                                            // weighted sum
7          for all capsules j in layer (l+1) : v_j ← squash(s_j)
8          for all capsules i in layer l and capsules j in layer
            (l+1) : b_ij ← b_ij + v_j · û_{j|i}              // update routing weights
9      end
10     return v_j
```

7.1.1 Routing illustration

In this section a simple illustration of the dynamic routing procedure is presented. Blue arrows within the squares are capsules, $\mathbf{u}_i$, from layer l. The red arrows are prediction vectors from the capsules in the layer l. The row table numbered 1 through 6 are indices for output capsules and contains the predicted vectors illustrated with green arrows.

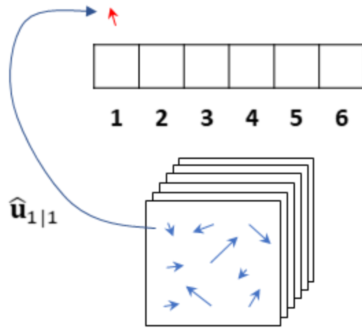

Figure 20: Computes a prediction vector for the first output capsule, $\hat{\mathbf{u}}_{1|1}$, by applying the transformation matrix $\mathbf{W}_{11}$ to the first capsule from the current layer, $\mathbf{u}_1$, see eq. (14).

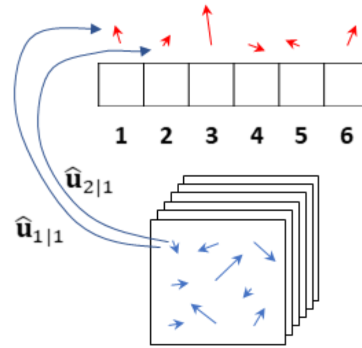

Figure 21: Continue to produce prediction vectors from the vector $\mathbf{u}_1$ to all different capsules in the next layer. When all prediction vectors have been computed for $\mathbf{u}_1$, advance to the next vector $\mathbf{u}_2$ and so on, see Figure 22.

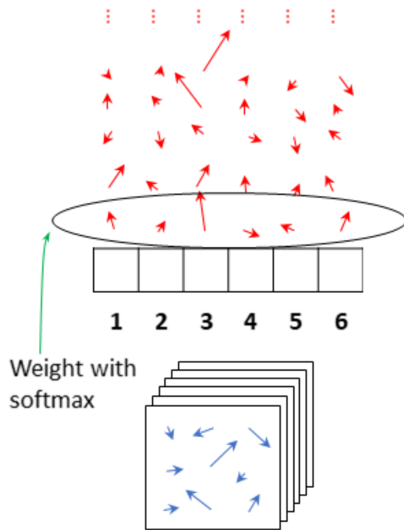

Figure 22: Once all prediction vectors have been computed we weight the batch of prediction vectors from $\mathbf{u}_1$ in the current layer using the coupling coefficients, see eq. (18). In the figure this is illustrated by the ellipse enclosing the first row of red prediction vectors. This is repeated for all prediction vectors and all capsules $\mathbf{u}_i$.

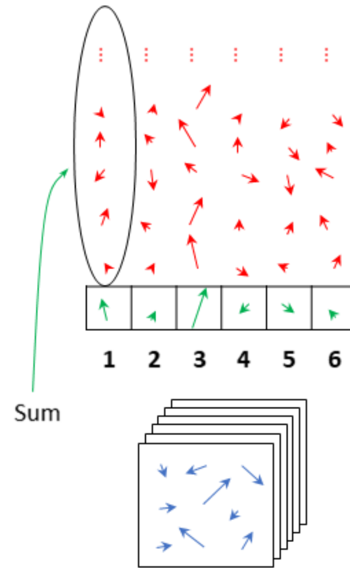

Figure 23: The prediction vectors have now been rescaled with the coupling coefficients, eq. (18). By taking the sum along all weighted predictions for each capsule in the next layer, weighted predictions are produced for all capsules in the next layer.

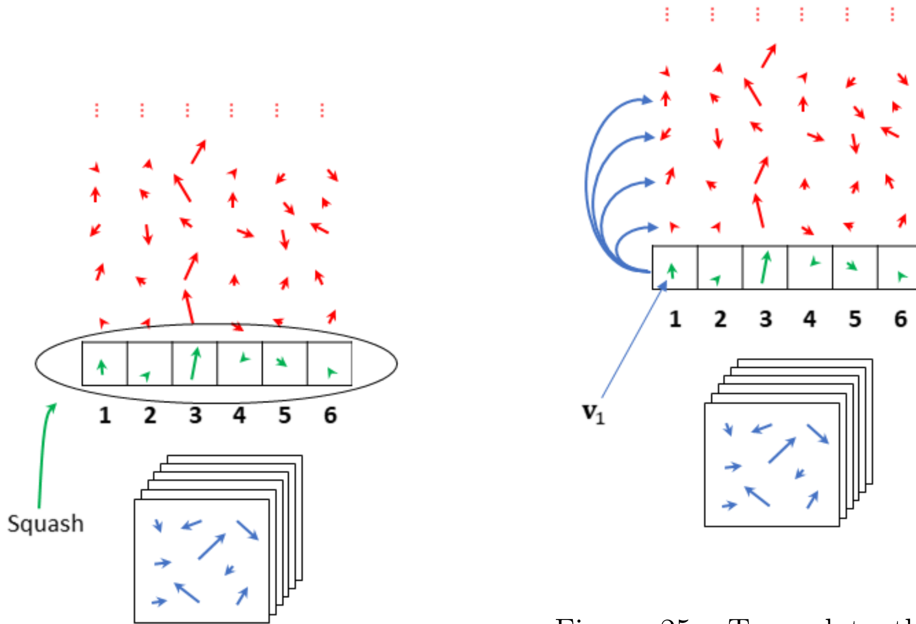

Figure 24: The squash function (15) is applied. These final predictions will then be used to update the routing weights used in the next iteration, see eq. (19).

Figure 25: To update the routing weights, b_{ij}, the agreement between prediction vector $\hat{\mathbf{u}}_{j|i}$ and the predicted vector for capsule j, $\mathbf{v}_j$, is calculated, see eq. (19). In this case it seems like the third capsule gives the longest vector and thus there is with high probability an entity encoded by capsule 3 in the image.

7.2 Loss function

Since capsule networks have not been used for matching tasks before, new loss functions are needed. Taking inspiration from the loss function, in equation (11), used in our implemented siamese networks a modified contrastive loss function is introduced in the following section.

7.2.1 Contrastive loss for capsules

When going from the regular contrastive loss (11) to contrastive loss for capsules the only difference is that in the capsule case the outputs are vectors instead of scalars. This is handled by measuring distance between corresponding capsules in the two branches of a siamese network. Let $\mathcal{I}$ be the set (9) and $\mathbf{f}(\mathbf{i})$ be a function that maps input image $\mathbf{i}$ to outputs in a capsule network. Moreover let $\mathbf{f}_j(\mathbf{i}) = \mathbf{v}_j(\mathbf{i})$ denote the j:th capsule output from image $\mathbf{i}$. Define the Euclidean distance between the j:th capsule outputs from image pair $(\mathbf{i}_1, \mathbf{i}_2)^k$ as

$$D_j^k := \left\|\mathbf{v}_j(\mathbf{i}_1) - \mathbf{v}_j(\mathbf{i}_2)\right\|_2, \quad \mathbf{i}_1, \mathbf{i}_2 \in (\mathbf{i}_1, \mathbf{i}_2)^k. \tag{20}$$

Then the contrastive loss function is defined as

$$L = \frac{1}{2N} \sum_{k=1}^{N} \sum_{j=1}^{C} l_k (D_j^k)^2 + (1 - l_k) \cdot \max\left(0, m - D_j^k\right)^2, \tag{21}$$

where l_k is given by (10), N is the number of image pairs, C the number of capsule outputs from the network and m the margin hyperparameter, similar as in (11).

In the matching case the first term within the double sum contributes to the loss function. This contribution will be small if both $\mathbf{v}_j(\mathbf{i}_1)$ and $\mathbf{v}_j(\mathbf{i}_2)$ are similarly oriented or if both vectors are very short. The maximum loss is achieved if two vectors with high probability are orientated opposite of one another and a high loss is assigned if one vector is long while the other is short. Thus the loss function will push the network to produce either short vectors in both images or long vectors with similar orientation.

In the non matching case a margin is imposed to punish outputs that have small distances between each other. If the network produces outputs such that $D_j^k \geq m$ the loss contribution will even be zero.

8 Training Neural Networks

Training a constructed network is not always trivial since there are numerous factors to take into account in order to train a successful network. This section summarizes common factors to consider when training a network, most of which we utilized in our implemented networks.

8.1 Regularization

One sign that a trained network is overfitted to its training data is that the trained parameters are very large in absolute value. To reduce the degree of overfitting regularization is often incorporated into the loss function. Regularization penalizes large parameters and thus constrains the trained model to adopt parameters that are not too large in absolute value. There are various methods for introducing regularization, one of them is the so called **l2-regularization**. It is done by simply adding a weighted sum of the 2-norms of all network parameters to the loss function to be minimized [14]. The weights are regularization parameters that determine the degree of penalty for network parameters.

8.2 Batch normalization

Batch normalization (BN) was introduced 2015 in [25] with the intention of facilitating training of deep neural networks. The authors of [25] list several benefits of incorporating batch normalization in neural networks, most notably are:

i BN enables higher learning rates when training a network

ii Network parameters are less dependent on careful initialization

iii BN works as a partial regularizer

To accomplish this the idea of batch normalization is to reduce internal covariance shift [25], which is defined as change in distribution of node values in a network due to changes in network parameters. Internal covariance shift

obstructs training of networks since they have to account for changes in the distributions of node values. Batch normalization reduces internal covariance shift by constraining all node values to have standard normal distribution with zero mean and unit variance.

We start by formulating the batch normalization algorithm for nodes in a regular neural network (e.g. an activation in a fully connected layer). Consider a node value, which we denote by x, then the batch normalization algorithm can be formulated according to Algorithm 2. Note that Algorithm 2 returns a linear transformation of the normalized node value. The reason for this transform is that simply normalizing a node's value changes the features it is able to represent, thus the transformation is added in order to compensate for this. When performing the normalization step the denominator contains an $\epsilon > 0$ for numerical stability.

Algorithm 2: Batch normalization transform

Input: Batch $\mathcal{B} = \{x_1, x_2, \ldots x_n\}$ of values of x,
trainable parameters γ , β
Output: y

1. $\mu_\beta \leftarrow \frac{1}{n}\sum_{k=1}^{n} x_k$ // batch mean
2. $\sigma_\beta^2 \leftarrow \frac{1}{n}\sum_{k=1}^{n}(x_k - \mu_\beta)^2$ // batch variance
3. $\hat{x} \leftarrow \frac{x_i - \mu_\beta}{\sqrt{\sigma_\beta^2 + \epsilon}}$; for some $1 \leq i \leq n$ and $\epsilon > 0$ // normalize
4. $y \leftarrow \gamma\hat{x} + \beta$ // scale and shift
5. **return** y

For convolutional layers the node values in a feature map are jointly normalized across both the elements of a batch and spatial dimension. Thus the input to Algorithm 2 now consists of all node values in a feature map for all elements of a batch. For example if we have a feature map of size $m \times n$ and use a batch of size b then the input to the batch normalization algorithm will have the size $m \cdot n \cdot b$. The parameters γ anf β are now common for an entire feature map instead of a single node.

When training a batch normalized network the values of the nodes for which batch normalization is applied to are replaced with their corresponding batch normalized transforms. The set of parameters to optimize is then the union of all original network parameters and the parameters introduced by batch normalization. During testing of a network it is not desirable to normalize w.r.t. a batch, the normalization step is instead done using moving averages of the batch means and variances from training. More details of batch

normalization during inference is found in [25].

8.3 Dropout

The idea of dropout was published in 2014 in the Journal of Machine Learning Research 15 as a means of reducing overfitting in neural networks [26]. One method of reducing overfitting is model combination, i.e. to train several different networks to accomplish the same task and then averaging their outputs to compute the final output. However this approach of reducing overfitting also comes with various practical limitations. If the trained networks have large depths it will require a considerable amount of time to train the networks. Moreover inference with several deep networks can become impractical for applications requiring low response time.

Dropout resembles model combination but achieves this through a different method. Applying dropout to a neural network involves temporarily excluding neurons contained in the network during training. A node is excluded from the network with probability $0 \leq p \leq 1$, independent of other nodes. Excluding a neuron is done simply by setting its value and all weights connected to it to zero. This is of course applicable for all neurons in a network, thus all neurons for which dropout is applied to are Bernoulli distributed with probability $0 \leq p \leq 1$. Applying dropout to a network can be interpreted as selecting a sub-network consisting of a subset of nodes from the original network. From this interpretation it is trivial to conclude that training a network with dropout is equivalent to training numerous of its sub-networks. As mentioned at the beginning of this section the output from several trained networks can be combined by averaging to produce the inference output. Even in the case of dropout, where sub-networks of a network are sampled, averaging over trained models can turn out to be impractical since there are exponentially many sub-networks. Given a network consisting of n nodes there are 2^n sub-networks. Therefore in a network with dropout, the averaging is approximated by weight scaling, which implies that given a node with dropout probability p the outgoing weights from this node are scaled with p during inference. Thus during inference the original network is used and its trained parameters are scaled in accordance with the dropout probabilities of its nodes. It should be emphasized that the exclusion of nodes is only done during training, not during inference.

To gain some intuition on how dropout reduces overfitting one can consider

a particular node in a regular neural network and all neurons connected to it. Assume that there is a large amount of neurons connected to our node of interest and suppose that this node has a poor representation of some feature of the input to the whole network. Then during training without dropout, the neurons connected to our node of interest might compensate for their lack of performance by adapting their weight connections. This can occur for numerous nodes in a network, which results in many co-adaptations between elements in a network. Such a network will in turn generalize poorly to unseen data. Ideally each node of a network should be able to represent features of an input without relying heavily on the presence of other neurons to adjust for its inability to represent some features. Dropout achieves this by temporarily removing neurons from a network, forcing the remaining neurons to learn to work with each other. Since nodes are randomly removed during training each neuron learns to work with random subsets of other nodes. This implies that each node will not become dependent on a particular set of nodes and above all they will strive towards representing useful features by themselves. The result is a network with increased robustness towards new data.

Since our neural network architectures will have a siamese design we would also like to point out an important detail when training siamese networks with dropout. In Section 4.1 we mentioned that siamese networks have the same convolutional architecture in both of its branches (for a triplet network all three branches have the same convolutional architecture). This puts a constraint on using dropout on siamese networks. After the dropout operation a siamese network must still have the same convolutional structure in both branches. In other words the same neurons must be dropped from each branch of a siamese network. This can be accomplished in practice by fixing a seed when applying dropout.

9 Method

In the first subsection of this section the training methodology used for training our networks is discussed, which includes among other things: parameter initialization, optimization algorithms and hyperparameter tuning. The second subsection will discuss the data utilized to train our implemented networks and introduce a definition of the term similar that appeared in the formulation of problem P in Section 2. Following two subsections will describe decision making in trained networks and common metrics used to evaluate fingerprint recognition algorithms respectively.

9.1 Training methodology

All network architectures are implemented using Python with TensorFlow API. The models are trained on a computer with Intel Core i7-8700K 3.70GHz, 2x Geforce GTX 1080 Ti and 32 Gb RAM. We did not utilize multi-GPU for training one model, instead we trained two models simultaneously using single GPUs.

Most network parameters are initialized using an uniform Xavier initializer, $U(-x, x)$, where $x = \sqrt{\frac{6}{in+out}}$ and *in* and *out* are the number of input and output neurons respectively. Consider for example an input with dimensions $5 \times 5 \times 1$ to a convolutional layer with same padding, stride of one and 8 convolutional filters, then $in = 5 \cdot 5 \cdot 1$ and $out = 5 \cdot 5 \cdot 8$. All biases are initialized as zero. Our networks will also use batch normalization and leaky-relu activations, where the hyperparameters to these methods are TensorFlow's default settings [27],[28]. To minimize the loss functions for our networks we varied between two optimizers: stochastic gradient descent with momentum [29] and Adam optimizer [30].

The contrastive loss function (11) contains a margin parameter that determines if dissimilar pairs of images should contribute to the loss function. As the network trains ideally it should learn to separate matching and non-matching pairs. The distance between similar pairs should be lower than for dissimilar pairs. With this in mind we decided to increase the margin during training, since if the network has trained for a long period and the distance between dissimilar pairs is still low then one should penalize such a pair. Our implemented siamese network will output normalized feature

vectors to the decision layer,see Figure 11. The maximum distance between such feature vectors is 2, which follows from the triangle inequality. This provides a natural upper bound for the margin parameter in the contrastive loss function, having a margin larger than two implies that all dissimilar pairs will contribute to the loss function regardless of the distance between them.

9.2 Data

To train our networks we used a dataset provided by Precise Biometrics. The data consists of 192×192 grayscale images of fingerprints. Each image is also associated with the following additional information:

1 person_id - integer indexing the person that the current image originates from

2 finger_id - integer indexing which finger the current image depicts

3 translation - a pair of real numbers representing the two dimensional translation of the image relative a determined origin

4 rotation - scalar representing the rotation of the current image relative a determined origin

The translation data was converted to pixel coordinates before use, since it was originally given in another format. Similarly the rotation data was transformed to degrees. Also note that the translation and rotation are relative to some determined origin, where the origin is unique for each fingerprint from a specific person. However we will only be interested in the difference in translation and rotation between images so the location of the origins are not important.

Having presented the data utilized in this project we now define the notion of similarity used in problem P:

Definition 9.1. Let $\mathbf{t}(\mathbf{i}) = (t_x(\mathbf{i}), t_y(\mathbf{i}))$ be the two dimensional translation in pixel coordinates for image $\mathbf{i}$ and let $r(\mathbf{i})$ denote the rotation of image $\mathbf{i}$ in degrees. Two images $\mathbf{i}_1$ and $\mathbf{i}_2$ are similar if they originate from the same person, depict the same finger and the following conditions are satisfied: $\|t_x(\mathbf{i}_1) - t_x(\mathbf{i}_2)\|_2 < \Delta t$, $\|t_y(\mathbf{i}_1) - t_y(\mathbf{i}_2)\|_2 < \Delta t$ and $\|r(\mathbf{i}_1) - r(\mathbf{i}_2)\|_2 < \Delta r$. Here Δt and Δr are small positive scalars.

In definition 9.1 there are two adjustable parameters, namely Δt and Δr. Intuitively the parameters should be set to small numbers, since images with small rotation and translation differences have high overlap and thus are more likely to be similar. Images that do not satisfy definition 9.1 are considered dissimilar.

Since our networks will solve a matching task using siamese-based architectures the inputs to some networks will be pairs of images. We use definition 9.1 to generate ground truth labels for pairs of fingerprint images. Similar images are labelled with 1 and dissimilar images with 0. Given a set of fingerprint images with cardinality n there are $\binom{n}{2}$ pairs, if one disregards the ordering of elements within a pair. The fraction of similar and dissimilar images will depend on the parameters Δt and Δr in definition 9.1. In practice we found that setting Δt and Δr to small numbers yielded significantly more dissimilar images than similar. Thus we could remove some non-matching pairs. Table 1 summarizes the various datasets of pairs used in this project. Before generating datasets of pairs the original database from Precise Biometrics is partitioned into training, validation and test sets. The training set constitutes 80% of the original dataset and the validation and test set consist of 10% each. From these partitions the corresponding partitions for image pairs are generated. During training the training and validation sets are shuffled before use and reshuffled for each epoch.

Datasets of pairs					
Name	Δt	Δr	number of similar pairs	number of dissimilar pairs	total number of image pairs
data1	10	5	7178	1172430	1179608
data2	30	5	57799	1172430	1230229

Table 1: List of datasets utilized to train networks with two image inputs. The parameters Δt and Δr are part of definition 9.1

When working with triplet networks the data set has to be divided into three categories: anchors, positives and negatives. As described in Section 5.1 positive images are matching to the anchor and negative images are non matching. To define whether images are matching the same definition as for the siamese network was used, Def. 9.1. The two datasets, data3 and data4, were partitioned into training, validation and test sets with the same distribution and sensitivity to rotation and translation as above in table 1.

Datasets of triplets					
Name	Δt	Δr	number of similar pairs	number of dissimilar pairs	total number of image pairs
data3	10	5	7178	1172430	1179608
data4	30	5	57799	1172430	1230229

Table 2: List of datasets utilized to train and evaluate triplet networks. The parameters Δt and Δr are a part of definition 9.1. These datasets have the same translation and rotational difference as data1 and data2 in table 1 but due to implementational reasons they have different dissimilar image pairs.

9.3 Inference for trained models

To evaluate our trained siamese/inception/triplet network models we use a decision layer (see e.g. Figure 11) inspired by the intuition of the contrastive loss function (11) and triplet loss function (12). Similar images should be close to each other in output space and dissimilar images distant from each other. The decision layer discriminate between similar and dissimilar pairs of images by using a threshold $\epsilon > 0$ such that all pairs $(\mathbf{i}_1, \mathbf{i}_2)$ that satisfies $\|\mathbf{f}_\Omega(\mathbf{i}_1) - \mathbf{f}_\Omega(\mathbf{i}_2)\|_2 < \epsilon$ are considered similar, otherwise they are dissimilar. Here $\mathbf{f}_\Omega$ is some parametric function mapping input to some output space. For instance in a siamese network $\mathbf{f}_\Omega$ maps input to a vector space (normalized in our implemented siamese networks).

We use a similar decision layer for capsule networks, the difference is that a capsule network outputs capsules (which are vectors). Thus we compute euclidean distance between corresponding capsules in an image pair followed by taking mean over the number of capsules. More specifically two images $\mathbf{i}_1$, $\mathbf{i}_2$ are considered similar if $\frac{1}{C}\sum_{j=1}^{C} \|\mathbf{v}_j(\mathbf{i}_1) - \mathbf{v}_j(\mathbf{i}_2)\|_2 < \epsilon$, where C is the number of output capsules, $\mathbf{v}_j(\mathbf{i}_1)$ is capsule j from image $\mathbf{i}_1$ and $\epsilon > 0$ is a decision threshold.

9.4 Evaluation metrics

In this section we summarize metrics commonly used to benchmark fingerprint recognition algorithms. Assume that we have a trained model and want to evaluate it using a test set of image pairs. Moreover introduce the

following terms:

- P - number of positive examples (examples corresponding to a match)
- N - number of negative examples
- TP - number of true positives predicted by a model
- TN - number of true negatives predicted by a model
- FP - number of false positives
- FN - number of false negatives

then we can define the metrics described by the equations (22) - (25):

$$recall = \frac{TP}{P} \tag{22}$$

$$accuracy = \frac{TP + TN}{P + N} \tag{23}$$

$$FPR = \frac{FP}{N} \tag{24}$$

$$FNR = \frac{FN}{P} \tag{25}$$

where FPR, FNR are short for false positive rate (a.k.a. false acceptance rate) and false negative rate (a.k.a. false reject rate) respectively.

Depending on application the emphasis on false accepts and false rejects may vary. In security critical applications it is desirable to have a low false positive rate. Therefore it might be of interest to consider the performance of a model for small false positive rates. We will measure recall for small values of FPR to benchmark our networks for this special case. In general the relation between recall and FPR is given by a receiver operating characteristic (ROC) curve. A ROC curve illustrates recall as a function of false positive rate for varying decision thresholds, however the threshold values are suppressed in the plot. When evaluating fingerprint recognition systems one is particularly interested in the behaviour of the model for small FPR, thus we put a log-scale (base 10) on the independent variable axis (FPR) in the ROC curves.

Using the decision layer for siamese networks described in Section 9.3 allows for a model that is flexible, in the sense that the decision threshold can be adjusted to suit different applications. This of course requires the model to perform reasonably well for various thresholds. We will therefore compute recall, false positive rate and false negative rate as functions of threshold to evaluate our implemented models for various decision thresholds.

10 Results

10.1 Siamese network

Table 3 summarizes the siamese network architecture that we implemented for dataset data1 in table 1. The model uses the contrastive loss function (11) and was trained using stochastic gradient descent with momentum. During training the margin in (11) is changed dynamically in accordance with the discussion in Section 9.1. In table 5 the hyperparameters for the implemented siamese network are summarized along with their values. The network uses the decision layer from Section 9.3 to determine whether two fingerprints match. Total time spent training the model was 35 minutes. The test set consists of 582 similar pairs and 116804 non-matching pairs.

The architecture used for data2 in table 1 is found in table 4 and this network is trained using SGD with momentum for 20 minutes. We used the same hyperparameter configuration as the siamese network for data1 during training (see table 5). As listed in table 1 the matching pairs in data2 are allowed to have larger translation differences. Just as for data1 we report the corresponding results for this dataset. In this case the test set consists of 5777 similar images and 116804 dissimilar image pairs.

layer	input dims	output dims	kernel/window	params
bn1	$192 \times 192 \times 1$	$192 \times 192 \times 1$		2
conv1	$192 \times 192 \times 1$	$93 \times 93 \times 16$	7×7, $S = 2$	800
bn2	$93 \times 93 \times 16$	$93 \times 93 \times 16$		32
leaky-relu	$93 \times 93 \times 16$	$93 \times 93 \times 16$		0
max pool 1	$93 \times 93 \times 16$	$46 \times 46 \times 16$	2×2 , $S = 2$	0
conv2	$46 \times 46 \times 16$	$21 \times 21 \times 16$	5×5, $S = 2$	6416
bn3	$21 \times 21 \times 16$	$21 \times 21 \times 16$		32
leaky-relu	$21 \times 21 \times 16$	$21 \times 21 \times 16$		0
max pool 2	$21 \times 21 \times 16$	$10 \times 10 \times 16$	2×2 , $S = 2$	0
conv3	$10 \times 10 \times 16$	$10 \times 10 \times 32$	3×3, $S = 1$	4640
bn4	$10 \times 10 \times 32$	$10 \times 10 \times 32$		64
leaky-relu	$10 \times 10 \times 32$	$10 \times 10 \times 32$		0
conv4	$10 \times 10 \times 32$	$10 \times 10 \times 64$	3×3, $S = 1$	18496
bn5	$10 \times 10 \times 64$	$10 \times 10 \times 64$		128
leaky-relu	$10 \times 10 \times 64$	$10 \times 10 \times 64$		0
flat	$10 \times 10 \times 64$	6400×1		0
fc1	6400×1	1024×1		6.55M
bn6	1024×1	1024×1		2048
leaky-relu	1024×1	1024×1		0
l2-norm	1024×1	1024×1		0
total				6.58M

Table 3: Siamese network architecture for dataset data1 (see table 1). S is the stride (given as one integer if the stride is the same in both spatial dimensions) and the column params lists the number of trainable parameters in each layer, where M stands for million. l2-regularization is used for all convolutional and fully connected components. The loss function used to train the network is given by (11) with addition of regularization terms. Dropout layers are used after all leaky-relu layers, the dropout rate is specified in table 5.

layer	input dims	output dims	kernel/window	params
bn1	$192 \times 192 \times 1$	$192 \times 192 \times 1$		2
conv1	$192 \times 192 \times 1$	$93 \times 93 \times 64$	7×7, $S = 2$	3200
bn2	$93 \times 93 \times 64$	$93 \times 93 \times 64$		128
leaky-relu	$93 \times 93 \times 64$	$93 \times 93 \times 64$		0
max pool 1	$93 \times 93 \times 64$	$46 \times 46 \times 64$	2×2 , $S = 2$	0
conv2	$46 \times 46 \times 64$	$21 \times 21 \times 64$	5×5, $S = 2$	$102K$
bn3	$21 \times 21 \times 64$	$21 \times 21 \times 64$		128
leaky-relu	$21 \times 21 \times 64$	$21 \times 21 \times 64$		0
max pool 2	$21 \times 21 \times 64$	$10 \times 10 \times 64$	2×2 , $S = 2$	0
conv3	$10 \times 10 \times 64$	$10 \times 10 \times 64$	3×3, $S = 1$	36928
bn4	$10 \times 10 \times 64$	$10 \times 10 \times 64$		128
leaky-relu	$10 \times 10 \times 64$	$10 \times 10 \times 64$		0
conv4	$10 \times 10 \times 64$	$10 \times 10 \times 128$	3×3, $S = 1$	73856
bn5	$10 \times 10 \times 128$	$10 \times 10 \times 128$		256
leaky-relu	$10 \times 10 \times 128$	$10 \times 10 \times 128$		0
flat	$10 \times 10 \times 128$	12800×1		0
fc1	12800×1	1024×1		13.1M
bn6	1024×1	1024×1		2048
leaky-relu	1024×1	1024×1		0
l2-norm	1024×1	1024×1		0
total				13.3M

Table 4: Siamese network architecture for dataset data2 (see table 1). S is the stride (given as one integer if the stride is the same in both spatial dimensions) and the column params lists the number of trainable parameters in each layer, where M stands for million and K thousand. l2-regularization is used for all convolutional and fully connected components. The loss function used to train the network is given by (11) with addition of regularization terms. Dropout layers are used after all leaky-relu layers, the dropout rate is specified in table 5.

Hyperparameters	
learning rate	10^{-4}
m_0	0.5
m_{max}	1.3
m_{factor}	1.1
m_{itr}	400
batch size	250
momentum	0.99
dropout rate	0.5
λ	0.3

Table 5: Hyperparameters used to train the networks in table 3 and 4. m_0 is the initial margin in (11) which is multiplied by m_{factor} every m_{itr} iterations during training. We also put an upper bound m_{max} on the margin. Momentum is the value used for the stochastic gradient descent optimizer, dropout rate is the rate used for all dropout layers and λ is the regularization parameter.

Figure 26 illustrates histograms over Euclidean distances between image pairs (in output space) in a subset of the test set of data1 (left figure) and data2 (right figure). To compute the histograms we used all similar pairs and an equal amount of dissimilar pairs from the corresponding test sets. The green histogram depicts the distance distribution for similar pairs and the red part shows the corresponding for dissimilar pairs. One can observe an overlap between the histograms, which corresponds to the brown area. This implies that the decision layer will make errors when trying to discriminate between similar and dissimilar pairs regardless of the choice of threshold. We obtain an imperfect discrimination between similar and dissimilar images in both figures.

In Figure 27 several metrics to measure the performance of the trained networks are plotted as functions of the decision threshold. For vectors that are normalized in output space an upper bound for the distance between any two pairs of images is 2, thus it is sufficient to consider thresholds in the interval $[0, 2]$. The equal error rate (EER, the threshold value where FPR is equal to FNR) is approximately 0.16 according to the left figure in Figure 27. According to the right figure in Figure 27 the EER is approximately 0.18. A lower EER indicates better performance.

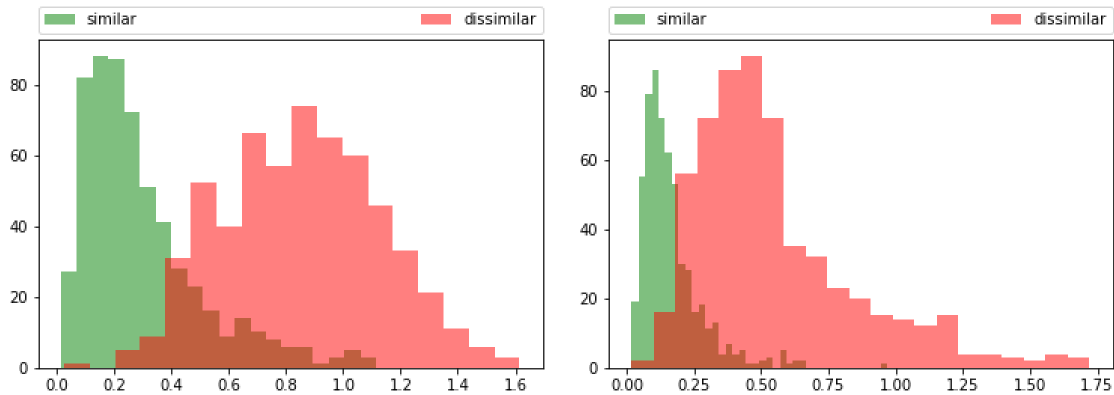

Figure 26: Histograms over Euclidean distances in output space between all image pairs in a subset of the corresponding test sets consisting of all similar pairs and an equal amount of dissimilar pairs from the test sets. The green histogram illustrates distances between similar images and the red histogram depicts distances between dissimilar images. In the left figure the results for data1 is illustrated and to the right the histograms for data2 are depicted.

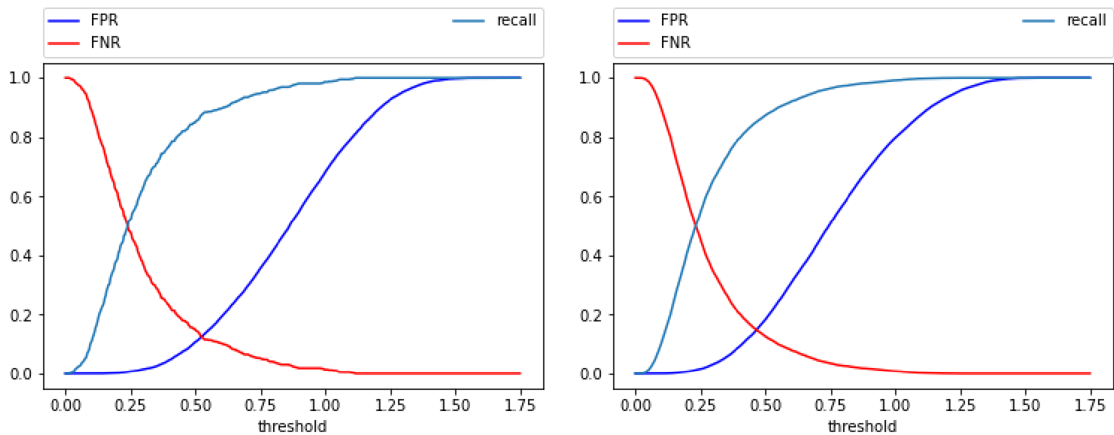

Figure 27: Plot of evaluation metrics as functions of the decision threshold for the models in table 3 and 4 using a subset of the test set of data1 (left figure) and data2 (right figure) respectively. The subset consists of all similar pairs and twenty times as many dissimilar image pairs for data1. In the case of data2 all similar examples are used and the number of negative examples is a factor 3 larger.

10.2 Triplet network

Table 6 contains the architecture of the triplet network used for datasets data3 and data4, see table 2. The model was trained using Adam optimization [30] on the triplet loss function defined in equation (12). Hyperparameters used for training the model are summarized in table 7. The network used for data3 and data4 was trained for 3 minutes (300 steps) and 25 minutes (1267 steps) respectively and both adopted an early stopping technique. To increase difficulty during training the network used an offline method. Every I_{itr}, see table 7, the dataset was sampled and evaluated using the latest model. The 3 hardest negative and the 5 hardest positive images per anchor where selected as the current training set.

layer	input dims	output dims	kernel/window	params
conv1	$192 \times 192 \times 1$	$186 \times 186 \times 16$	7×7, $S = 1$	800
relu	$186 \times 186 \times 16$	$186 \times 186 \times 16$		0
max pool 1	$186 \times 186 \times 16$	$93 \times 93 \times 16$	2×2 , $S = 2$	0
conv2	$93 \times 93 \times 16$	$93 \times 93 \times 16$	5×5, $S = 1$	6416
relu	$93 \times 93 \times 16$	$93 \times 93 \times 16$		0
max pool 2	$93 \times 93 \times 16$	$46 \times 46 \times 16$	2×2 , $S = 2$	0
conv3	$46 \times 46 \times 16$	$44 \times 44 \times 32$	3×3, $S = 1$	4640
relu	$44 \times 44 \times 32$	$44 \times 44 \times 32$		0
max pool 3	$44 \times 44 \times 32$	$22 \times 22 \times 32$	2×2 , $S = 2$	0
conv4	$22 \times 22 \times 32$	$20 \times 20 \times 64$	3×3, $S = 1$	18496
relu	$20 \times 20 \times 64$	$20 \times 20 \times 64$		0
max pool 4	$20 \times 20 \times 64$	$10 \times 10 \times 64$	2×2 , $S = 2$	0
flat	$10 \times 10 \times 64$	6400×1		0
fc1	6400×1	1000×1		6.4M
relu	1000×1	1000×1		0
total				6.43M

Table 6: Triplet network architecture for both datasets data3 and data4 (see table 2). S is the stride (given as one integer if the stride is the same in both spatial dimensions) and the column params lists the number of trainable parameters in each layer, where M stands for million. l2-regularization is used for all convolutional and fully connected layers. The loss function used to train the network is given by (12) with addition of regularization terms. Training set difficulty was increased twice during training and an early stop technique was adopted. The first two convolutional layers were initialized using an uniform distribution between -1 and 1, the rest of the parameters were initialized using the standard Tensorflow initialization stated in Section 9.1.

Hyperparameters	
learning rate	10^{-3}
β_1	0.9
β_2	0.999
I_{itr}	100/500
batch size	300
λ	1.0
α	4.0

Table 7: Hyperparameters used to train the network in table 6. I_{itr} indicates the number of iterations until the difficulty of the training set is increased as described in Section 5.3. For data3 $I_{itr} = 100$ and $I_{itr} = 500$ for data4. The β_1 and β_2 parameters are the forget rates of the gradients and the second moments of gradients [30]. λ is the regularization parameter used in all regularizers in the network in table 6. α is the margin discussed in Section 5.2.

Figure 28 shows the same metric as Figure 26 but for the triplet network. One can note that the distances approximately are in the range $[0, 8]$ which is a much wider span than that of the siamese network. This is due to no normalization of the output of the triplet network. It's is also apparent that there is no overlap between the matching and non matching images for very small distance. This is eminent in fingerprint matching where a low false positive rate is demanded. However there is still a considerable overlap between the distribution, hence the model will make faulty predictions. The equal error rate is approximately 0.15 in both plots in Figure 29. One can take notice of that the FPR in the left plot in Figure 29 stays close to zero for slightly higher threshold values than in the plot to the right. Hence the model will perform better on dataset data3 as shown in table 12.

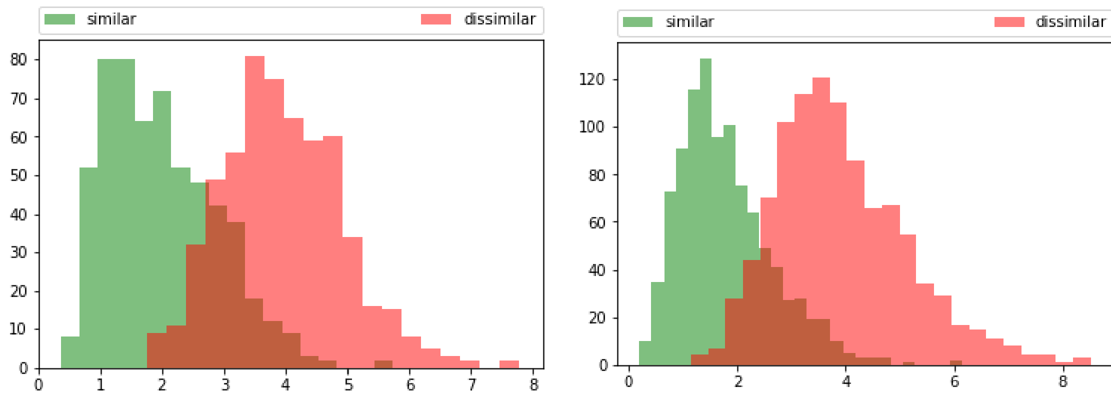

Figure 28: Histograms over Euclidean distances in output space between all image pairs in a subset of the corresponding test set consisting of all similar pairs and an equal amount of dissimilar pairs from the test set. The green histogram illustrates distances between similar images and the red histogram depicts distances between dissimilar images. In the left figure the results for data3 is illustrated and to the right the histograms for data4 are depicted.

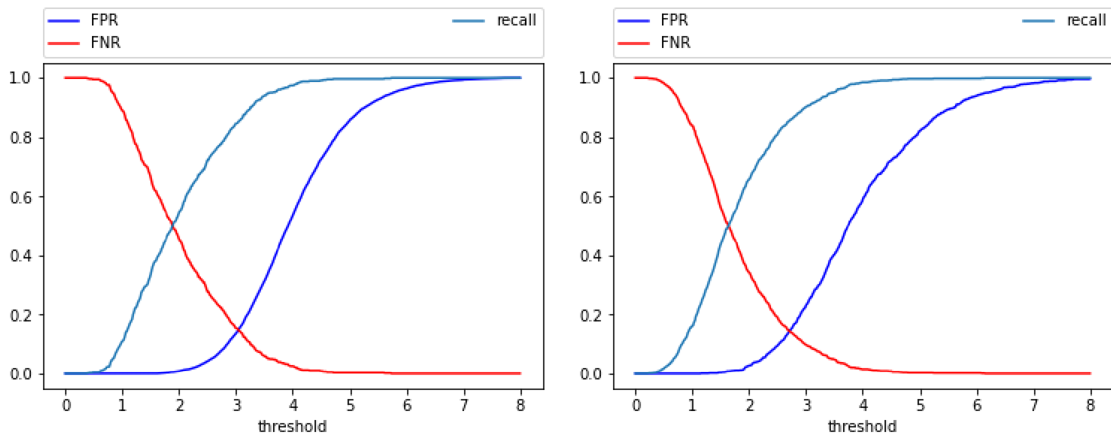

Figure 29: Plot of evaluation metrics as functions of the decision threshold for the model in table 6 using a subset of the test set of data3 (left figure) and data4 (right figure) respectively. The subset consists of all similar pairs and three times as many dissimilar image pairs.

10.3 Inception network

Implemented inception architectures will be represented in the same linear table format as for previous networks (see e.g. table 3), where inception modules are included as rows in a table. For details regarding the structure of inception modules we refer to the appendix Section 13.1.

We used the same inception architecture, training methodology and hyperparameter configuration for both data1 and data2 (see table 1). In table 8 the inception network is illustrated. The loss function for the model is given by (11) and it is trained with SGD with momentum for 45 minutes. Hyperparameter configuration is listed in table 9. We compute the evaluation metrics as in Section 10.1 using the same subsets of test sets of data1 and data2.

layer	input dims	output dims	kernel/window	params
bn1	$192 \times 192 \times 1$	$192 \times 192 \times 1$		2
conv1	$192 \times 192 \times 1$	$186 \times 186 \times 16$	7×7, $S = 1$	800
bn2	$186 \times 186 \times 16$	$186 \times 186 \times 16$		32
leaky-relu	$186 \times 186 \times 16$	$186 \times 186 \times 16$		0
max pool 1	$186 \times 186 \times 16$	$93 \times 93 \times 16$	2×2 , $S = 2$	0
conv2	$93 \times 93 \times 16$	$89 \times 89 \times 16$	5×5, $S = 1$	6416
bn3	$89 \times 89 \times 16$	$89 \times 89 \times 16$		32
leaky-relu	$89 \times 89 \times 16$	$89 \times 89 \times 16$		0
max pool 2	$89 \times 89 \times 16$	$44 \times 44 \times 16$	2×2 , $S = 2$	0
inception_a	$44 \times 44 \times 16$	$44 \times 44 \times 128$		30368
inception_a	$44 \times 44 \times 128$	$44 \times 44 \times 128$		44704
inception_a	$44 \times 44 \times 128$	$44 \times 44 \times 128$		44704
reduction	$44 \times 44 \times 128$	$21 \times 21 \times 256$		156K
inception_b	$21 \times 21 \times 256$	$21 \times 21 \times 192$		80736
max pool 3	$21 \times 21 \times 192$	$10 \times 10 \times 192$	2×2, $S = 2$	0
flat	$10 \times 10 \times 192$	19200×1		0
fc1	19200×1	512×1		9.8M
leaky-relu	512×1	512×1		0
l2-norm	512×1	512×1		0
total				10.1M

Table 8: Inception network architecture for dataset data1 and data2 (see table 1). S is the stride (given as one integer if the stride is the same in both spatial dimensions) and the column params lists the number of trainable parameters in each layer, where M stands for million and K thousand. l2-regularization is used for all convolutional and fully connected components as well as inception modules. The loss function used to train the network is given by (11) with addition of regularization terms. Dropout layers are used after the two first leaky-relu layers, the dropout rate is specified in table 9.

Hyperparameters	
learning rate	10^{-4}
m_0	0.5
m_{max}	1.3
m_{factor}	1.1
m_{itr}	400
batch size	75
momentum	0.99
dropout rate	0.5
λ	0.3

Table 9: Hyperparameters used to train the network in table 8 on data1 and data2. m_0 is the initial margin in (11) which is multiplied by m_{factor} every m_{itr} iterations during training. We also put an upper bound m_{max} on the margin. Momentum is the value used for the stochastic gradient descent optimizer, dropout rate is the rate used for all dropout layers and λ is the regularization parameter.

Inspection of the left figure in Figure 30 shows that the distance distribution between image pairs is spread over the interval $[0, 0.8]$, which is approximately a factor 2 less than the corresponding interval for the siamese network (see left figure in Figure 26). Moreover the means of the distributions are smaller. This might be slightly contradictory since the contrastive loss function (11) is designed to separate similar and dissimilar examples by mapping the latter to larger distances. But as we shall see later, this inception network actually performs better than the corresponding siamese network in table 3 w.r.t. our evaluation metrics. The equal error rate is approximately 0.10 according to the left figure in Figure 31. According to the right figure in Figure 31 the EER is approximately 0.13.

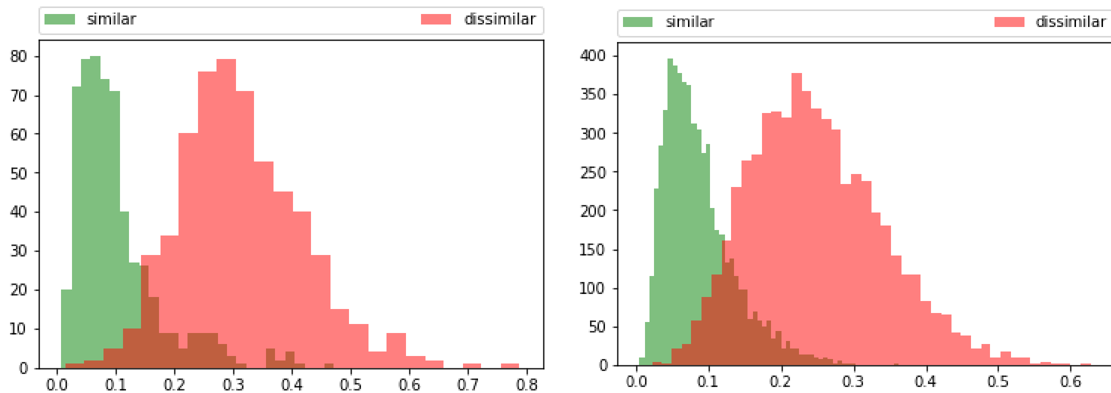

Figure 30: Histograms over euclidean distances in output space between all image pairs in a subset of the corresponding test set consisting of all similar pairs and an equal amount of dissimilar pairs from the test set. The green histogram illustrates distances between similar images and the red histogram depicts distances between dissimilar images. In the left figure the results for data1 is illustrated and to the right the histograms for data2 are depicted.

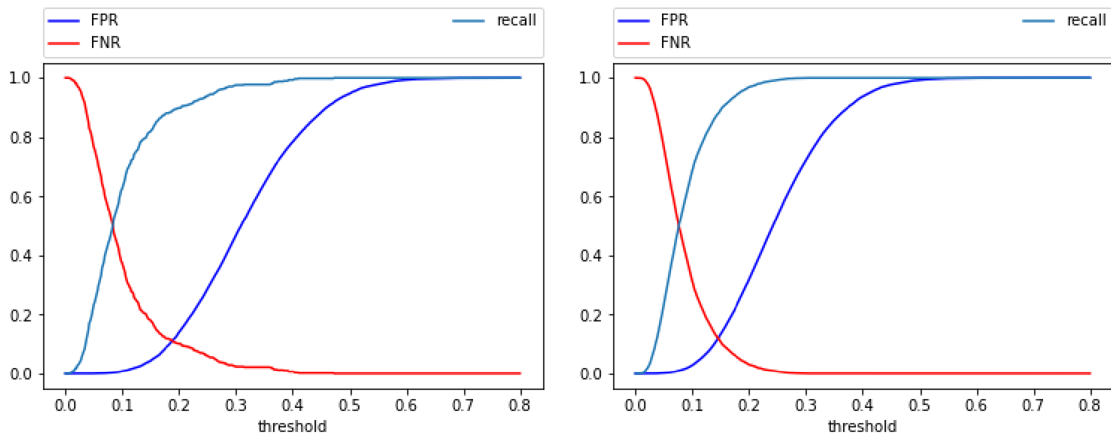

Figure 31: Plot of evaluation metrics as functions of the decision threshold for the model in table 8 using a subset of the test set of data1 (left figure) and data2 (right figure) respectively. The subset consists of all similar pairs and twenty times as many dissimilar image pairs for data1. In the case of data2 all similar examples are used and the number of negative examples is a factor 3 larger.

10.4 Capsule network

In table 10 the capsule network architecture for both data1 and data2 (see table 1) is summarized. The model uses the contrastive loss function (21) and was trained using Adam optimizer for 40 minutes for data1 and 20 minutes for data2. All convolutional filters and biases were initialized using kernels and biases from a pretrained siamese network, transformation matrices (14) are initialized as $\mathcal{N}(0, 0.1^2)$. The pretrained network was a siamese network trained with contrastive loss with almost the same configuration as the capsule network in table 10. The distinguishing part was that the siamese network terminated with a fully connected layer after the relu following conv3. The pretraining was performed to speed up the convergence of network and to focus the learning to the transformation matrix.

layer	input dims	output dims	kernel/window	params
conv1	$192 \times 192 \times 1$	$92 \times 92 \times 32$	9×9, $S = 2$	2624
relu	$92 \times 92 \times 32$	$92 \times 92 \times 32$		0
conv2	$92 \times 92 \times 32$	$42 \times 42 \times 32$	9×9, $S = 2$	82976
relu	$42 \times 42 \times 32$	$42 \times 42 \times 32$		0
conv3	$42 \times 42 \times 32$	$17 \times 17 \times 128$	9×9, $S = 2$	332K
relu	$17 \times 17 \times 128$	$17 \times 17 \times 128$		0
reshape	$17 \times 17 \times 128$	$17\times17\times8\times16$		0
squash	$17\times17\times8\times16$	$17\times17\times8\times16$		0
routing	$17\times17\times8\times16$	8×100		29.6M
total				30M

Table 10: Capsule network architecture for dataset data1 and data2 (see table 1). S is the stride (given as one integer if the stride is the same in both spatial dimensions) and the column params lists the number of trainable parameters in each layer, where M stands for million and K thousand. The loss function used to train the network is given by (21). The layer components marked in bold constitutes the first (and only) convolutional capsule layer. In the routing layer a learnable 8×8 transformation matrix, found in (14), is applied to each capsule in the previous layer.

Hyperparameters	
learning rate	10^{-3}
β_1	0.9
β_2	0.999
routing iterations	2
batch size	20/10
m	4.0

Table 11: Hyperparameters used to train the network in table 10 on data1 and data2. The β_1 and β_2 parameters are the forget rates of the gradients and the second moments of gradients [30]. Routing iterations is the number of iterations used in the routing procedure (algorithm 1). Batch sizes of 20 and 10 were utilized for data1 and data2 respectively. The margin in (21) is denoted by m.

Histograms over distances between similar and dissimilar examples can be seen in Figure 32 for image pairs from data1 (left figure) and data2 (right figure). A comparison of the left figure in Figure 32 with the corresponding plots for a siamese and inception network (left figures of Figures 26 and 30) shows that the distances between similar and dissimilar pairs are significantly less in our capsule network. The same can be said for the right figure in Figure 32. In Figure 33 the metrics (22), (24) and (25) as plotted as functions of the decision threshold for data1 (left figure) and data2 (right figure). The EER is approximately 0.22 for data1 and 0.37 for data2, as illustrated in Figure 33.

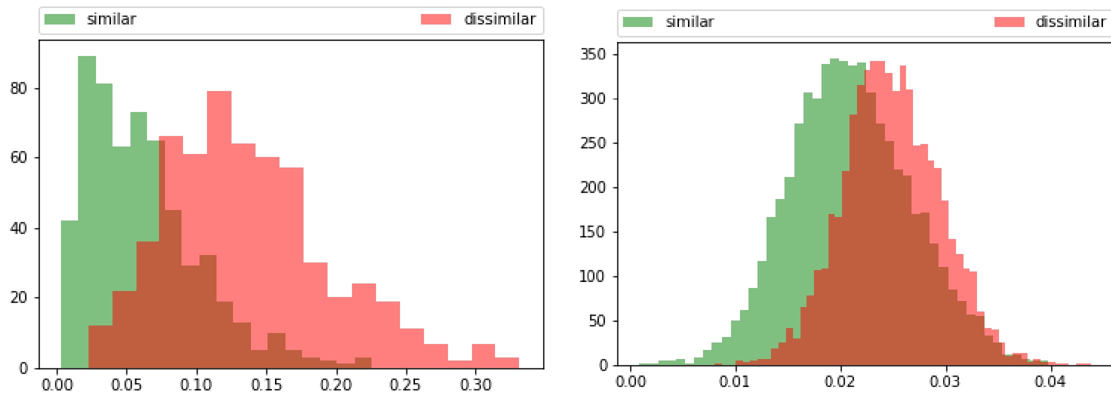

Figure 32: Histograms over euclidean distances in output space between all image pairs in a subset of the corresponding test set consisting of all similar pairs and an equal amount of dissimilar pairs from the test set. The green histogram illustrates distances between similar images and the red histogram depicts distances between dissimilar images. In the left figure the results for data1 is illustrated and to the right the histograms for data2 are depicted.

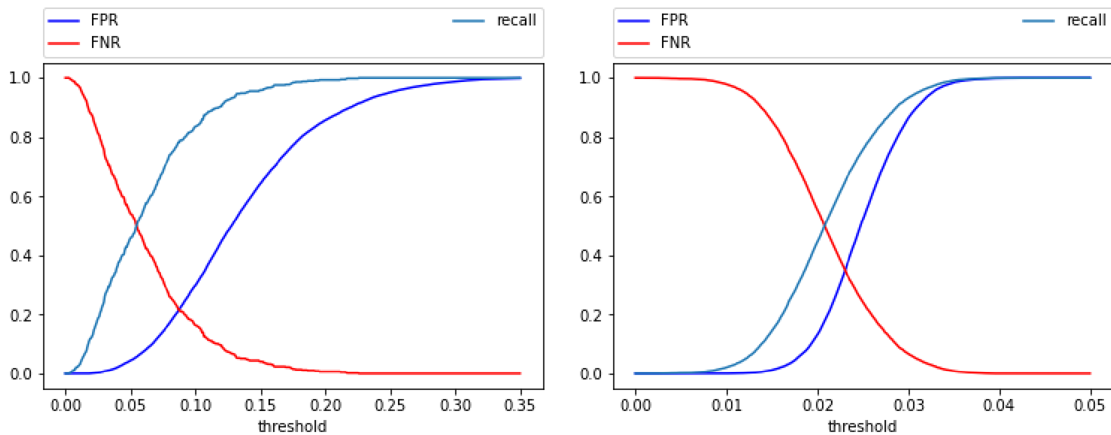

Figure 33: Plot of evaluation metrics as functions of the decision threshold for the model in table 10 using a subset of the test set of data1 (left figure) and data2 (right figure) respectively. The subset consists of all similar pairs and twenty times as many dissimilar image pairs for data1. In the case of data2 all similar examples are used and the number of negative examples is a factor 3 larger.

10.5 Model comparison

Figures 34 and 35 illustrate ROC-curves for implemented models on the various datasets in table 1 and 2. For models using data1 and data3 the results are computed using all similar test image pairs and 20 times more dissimilar pairs. In the case of data2 and data4 all similar test image pairs were used and 3 times as many dissimilar examples. Table 12 summarizes the recall for various small false acceptance rates.

As indicated by Figure 34 and table 12 the triplet network for data3 has the highest recall for small false positive rates roughly around 10^{-4} to 10^{-3}. For larger false positive rates the inception network has a higher recall than the triplet network. Moreover one can also observe that the ROC-curve belonging to the siamese network lies below the ROC-curve of the inception network (with some small exceptions close to 10^{-3}), thus we can conclude that the inception network has a better overall performance. The capsule network has higher recall than the siamese network for FPR close to 10^{-4}, but in general the performance of the capsule network is below that of the other models.

Inspection of Figure 35 shows that training on data with higher allowed translation difference between matching pairs reduces the performance of all models, compared to the corresponding results for smaller translation difference (see Figure 34). In this case there is a finer distinction between the performance of the triplet and inception networks. The triplet network performs better for most FPR values. Furthermore the performance of the siamese network is similar to the inception network for small FPR, while the capsule network still has the lowest performance.

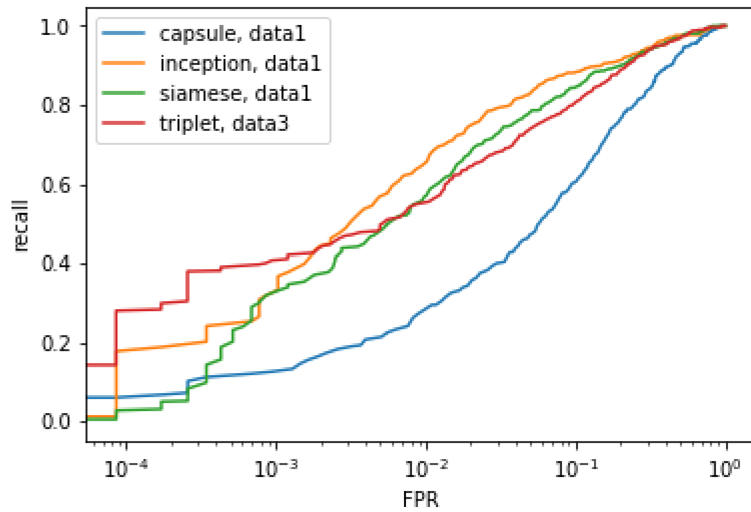

Figure 34: Illustration of ROC-curves for models in table 3, 6, 8 and 10 on subsets of test sets of data1 and data3. The subsets contain all matching test examples and 20 times as many non-matching test examples.

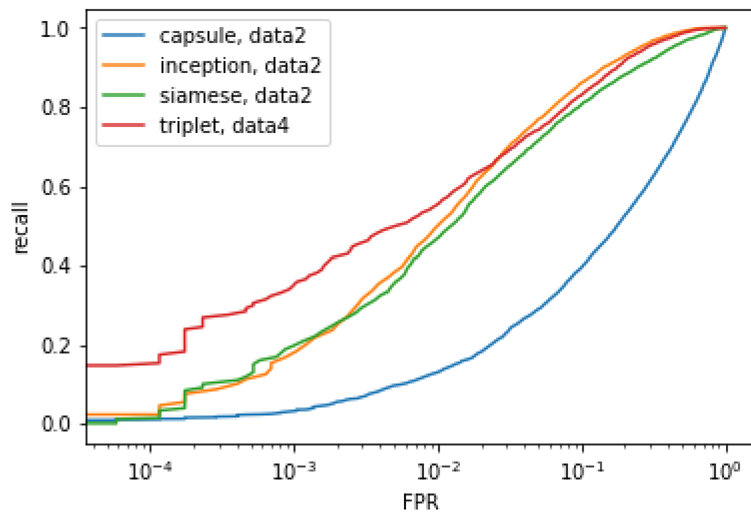

Figure 35: Illustration of ROC-curves for models in table 4, 6, 8 and 10 on subsets of test sets of data2 and data4. The subsets contain all matching test examples and 3 times as many non-matching test examples.

Model, data set	FPR = 10^{-1}	FPR = 10^{-2}	FPR = 10^{-3}	FPR = 10^{-4}
siamese, data1	0.847	0.571	0.331	0.050
triplet, data3	0.809	0.555	0.421	0.299
inception, data1	0.883	0.657	0.367	0.188
capsule, data1	0.607	0.295	0.133	0.067
siamese, data2	0.810	0.472	0.209	0.034
triplet, data4	0.837	0.555	0.371	0.171
inception, data2	0.865	0.508	0.184	0.046
capsule, data2	0.401	0.133	0.035	0.015

Table 12: Recall for various small false acceptance rates for all implemented models on their corresponding test sets.

11 Discussion

11.1 Decision errors - examples

This section discusses some test results from one of our implemented models, more specifically the inception model in table 8 using data1.

A difficult positive example is illustrated in Figure 36. The bounding boxes contain minutiae, which can be used to justify that the fingerprints are similar. The distance between the corresponding output vectors of our inception network is 0.246, which is larger than most matching pairs according to the left histogram in Figure 30. Even at the EER of 0.10 (see left figure in Figure 31) the decision layer would produce a false negative in this case.

Figure 36: Illustration of a hard positive example. The distance between the feature vectors corresponding to the image pair is 0.246, for the inception model in table 8 using data1.

In Figure 37 two images are illustrated. Ignoring the scars in the right figure the images look similar, but the ground truth states that they are dissimilar. In fact these fingerprints originate from two individuals. However the distance between the corresponding output vectors is 0.099, thus according to the left histogram in Figure 30 this example is more likely to be interpreted as a matching pair by the inception network.

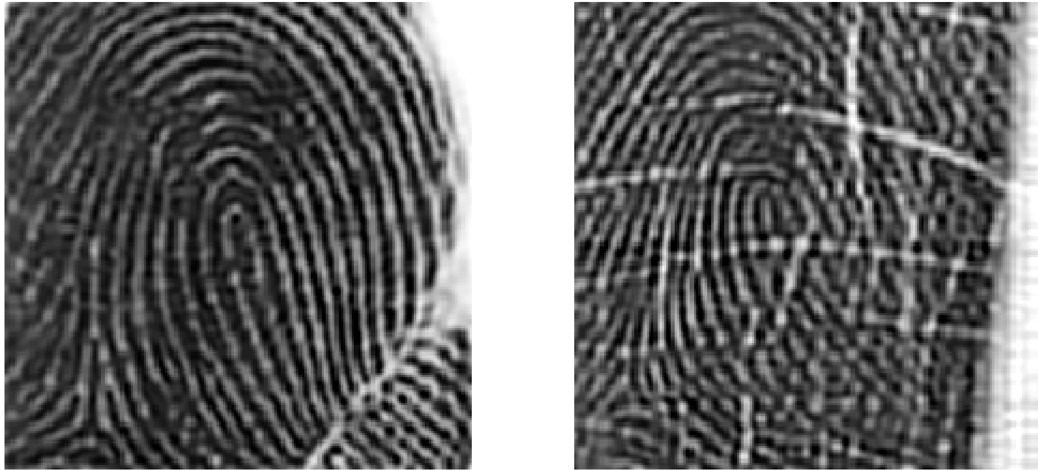

Figure 37: Illustration of a hard negative example. The distance between the feature vectors corresponding to the image pair is 0.099, for the inception model in table 8 using data1. If one disregards the scars the images look quite similar, but the ground truth label states that they are dissimilar.

Inspecting Figure 38 one can easily observe that the images correspond to a match. However the distance between the output vectors is 0.153, which is larger than most similar pairs according to the left histogram in Figure 30. The larger distance might be due to the partial absence of fingerprint data in the white area of the left image in Figure 38.

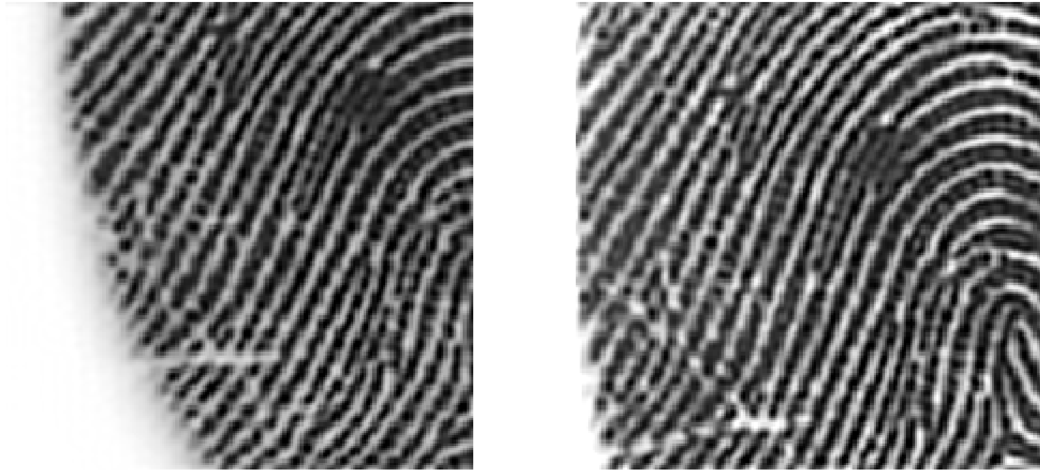

Figure 38: Example of a simple positive example with small translation. Distance between the feature vectors is 0.153, for the inception model in table 8 using data1.

Figure 39 depicts a case where an error occurs in the generation of ground truth labels. According to definition 9.1 the images in this figure are similar, by manual inspection it seems reasonable to assume that they do not match. Moreover the inception network gives a distance of 0.472 between the output vectors, which is more likely to correspond to a negative example (see left histogram in Figure 30). The reason for such errors in the ground truth labelling might be due to incorrect alignment data used in definition 9.1.

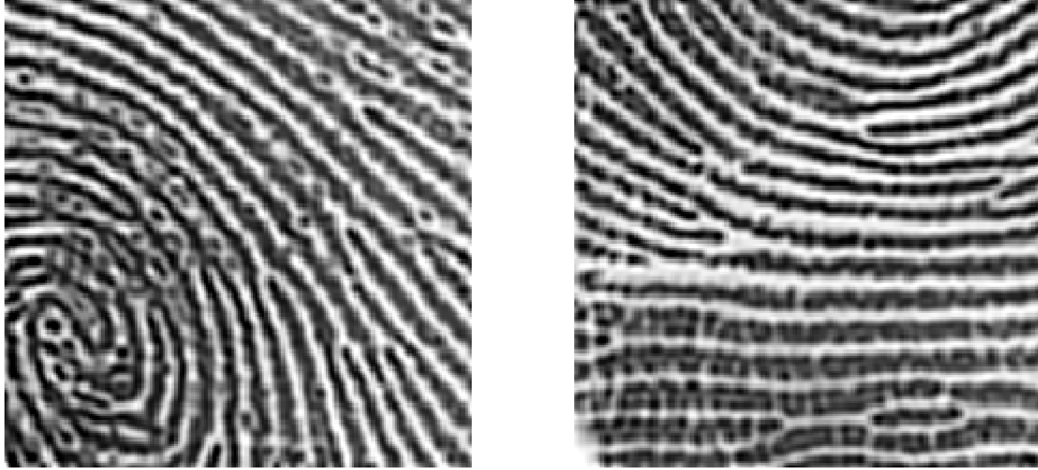

Figure 39: Illustration of wrong ground truth labelling. According to definition 9.1 these fingerprints correspond to a matching pair, however by inspection one can easily conclude that they are dissimilar. The distance between corresponding feature vectors is 0.472, for the inception model in table 8 using data1.

11.2 Potential issues with CNN

CNNs are exceptional at identifying and extracting useful features in an image, but in the context of fingerprint recognition there are potential issues which were not/partially addressed in this thesis. Features in fingerprints (e.g. minutiae points) can exist at various locations on the finger, have different orientations and are common across individuals. Thus simply identifying the existence of such features is not sufficient to discriminate between fingerprints originating from different persons. The motivation for investigating capsule networks for fingerprint recognition was to solve the problem of varying orientations of features, which is encoded by capsule entities in a capsule network. However the first and last aforementioned problems still remain in a capsule network. One possible solution to these issues is to encode relative positions of features. A reasonable constraint for identified features originating from the same finger is that they should be positioned in the same area of the finger. With knowledge about relative positions it is possible to differentiate fingerprints of various persons containing roughly the same features, since they are not likely to be located in the same areas

of the finger.

11.3 Features on different scales

According to Figure 34 the inception network performs better than the siamese network. The networks are trained using the same loss function given by (11) and both are built from the same components described in Section 3. Consequently it might seem peculiar that the inception network has higher performance. The only difference between these architectures is the addition of inception modules. This suggests that features of fingerprints exist on different scales, since inception modules are designed to identify and extract features on various scales.

11.4 Triplet network

In Figure 34 it's noticeable that the triplet network performs better than both the siamese and inception models for low FPRs but is superseded by the latter as the FPR increases. Our hypothesis for why the triplet network achieves better results at low FPRs is related to the increase in difficulty during training. As the training data sets get sampled after a fix amount of steps only the hardest positives and the hardest negatives are used in further training. Training on the hard negative images will lead to an increase in distances between these pairs. As we concentrate the learning to the hardest examples we expect to see fewer cases where negative images have very small distance between one another. This holds true as the triplet network performs the best at low FPR. It's made even more apparent on the harder data set where the triplet network is almost always better than it's competitors, see Figure 35. Since a low FPR is desirable for fingerprint authentication, where security is of essence, this behaviour of the model is highly appreciated.

11.5 Problems with capsule networks

From Sections 10.4 and 10.5 we can conclude that the performance of the capsule network were inferior compared to that of the other networks. In this section we highlight potential problems with our implementation of capsule networks for matching tasks.

The contrastive loss function given by (21) has several drawbacks, which contribute to the poor performance of our capsule network. By inspecting the definition of the loss function (21) one can observe that it does not incorporate the length of individual vectors, which codes for the probability of existence of a particular feature. Another drawback which is partly due to the loss function but is also affected by the non-linear squashing function (15) and initialization of network parameters, is that the network tends to train towards only recognizing dissimilar examples. This can occur when the first term in the double sum in (21) is close to zero. The implication of this is that matching image will only contribute with small loss values, while dissimilar examples tend to give larger loss values. Consequently the network will become better at distinguishing non-matching images. Most of the trainable parameters in all our networks were initialized from uniform distributions with small supports or normal distributions with zero mean and small standard deviations. In practice we observed that in most cases the resulting trained parameters were small in absolute value, this includes the pretrained siamese network used to initialize the convolutional components in our capsule networks. Propagating values close to zero through a network will yield small activations. In a capsule network this effect is further amplified by the squash function (15). Thus the first term in (21) becomes small for similar images and the second term is large for dissimilar images. By inspecting the length of each capsule output of our capsule models we observed that the majority of the output capsules have a length close to 1.

The aforementioned issues with the contrastive loss function give rise to additional problems related to the decision making component (see Section 9.3) of our capsule networks. If all output capsules have lengths close to 1 and are similarly oriented, which was empirically observed, then the distance metric used in our decision layer will become very small. Distance between dissimilar images will therefore be small. This contradicts the functionality of our decision layer for capsule network in the sense that the distance between dissimilar images should be large.

11.6 Future work

Due to lack of time we didn't have time to pursue all of our ideas. In this section we outline some possible paths for future work within this field.

11.6.1 Improvements on Capsules

As discussed above there are issues with how we do matching with our capsule network. To solve these problems there are some promising solutions. The first one is the use of other loss functions that makes better use of the individual length of each output capsule, i.e. the probability of entities, as well as orientation. In Section 13.2 in appendix there are a few suggestions of potentially useful loss functions.

Another improvement that could be made to the capsule network that we believe is bound to make the greatest progress of this model, is to do routing between images. The routing algorithm is designed to piece together features and put them in relation to each other, which is exactly what we think would make a great fingerprint matcher. As it's very important to know how features are related to one another both within and between fingerprints when doing matching we suggest stacking the output from two branches of a siamese network before doing the final routing step to the output capsule. In this step the routing should not be done within output from the same image but only between the two images output. This way the network would learn linear transformations between images and features and thus be able to put the location of features in different images in relation to each other.

Above we mentioned problems with the decision making process when using capsules. Alternative similarity metrics could turn out to increase the separation between matching and non matching images. We propose a decision process that focus more on the orientation combined with the agreement of length between corresponding capsules. An implementation closely related to the loss function in Section 13.2.3 in appendix could be a good starting point in designing a new decision process.

One possible improvement to our current capsule network is addition of several convolutional capsule layers. This allows for local routing procedures, whereas the present implementation only considers global routing of capsules. Local dynamic routing between consecutive convolutional capsule layers should work similar to algorithm 1 but only consider a subset of capsules within some local receptive field. Moreover the presence of multiple convolutional capsule layers will add more non-linearity to the network.

11.6.2 Triplet network - improvement

Our results for inception models in Figures 34 and 35 indicate that the inclusion of inception modules is favourable in terms of increasing the evaluation performance. The implemented triplet network (see table 6) does not utilize any inception modules. One improvement to this network might be achieved by incorporating inception modules into the triplet network architecture.

11.6.3 Patch based matching

Instead of trying to match whole fingerprint images a patched based matching problem could be interesting to investigate. We suggest dividing the fingerprint images into a grid structure and do exhaustive matching between all patches. By doing this we will be able to find patches that match between images. As we know the location in the image of the patches we can find the best transformation between the images such that matching patches overlap the most. The fitting problem would then output a score on how well patches overlap and with a threshold value it would possible to make inference on matching fingerprints.

12 Conclusion

The subject of this thesis was to investigate the applicability of convolutional neural networks for fingerprint recognition. With our implemented networks and the corresponding results (found in Section 10), we conclude that most of our CNNs provide sufficient performance for larger FPRs (approximately 10^{-2} and upwards), while we noticed a significant decrease in performance for small FPRs. In Section 11 potential problems with CNNs and capsule networks are brought up. Suggestions for improvements and new ideas are given, which can serve as a starting-point for future work.

References

[1] A.K.Jain, P.Flynn, and A.A.Ross, “Introduction to biometrics,” in *Handbook of Biometrics.* Springer Science & Business Media, LLC., 2008, ch. 1, pp. 1–22.

[2] A. K. Jain, R. Bolle, and S. Pankanti, *Biometrics: personal identification in networked society.* Springer Science & Business Media, 2006, vol. 479.

[3] H. Cummins and C. Midlo, *Palms and Soles: An Introduction to Dermatoglyphics.* Dover Publications, 1961.

[4] F. Galton, *Finger Prints.* Mcmillan, 1892.

[5] A.A.Moenssens, *Fingerprint Techniques.* Chilton, 1971.

[6] D.Maltoni, D.Maio, A.K.Jain, and S.Prabhakar, *Handbook of Fingerprint Recognition.* Springer-Verlag, 2003.

[7] H.C.Lee and R.E.Gaensslen, *Advances in Fingerprint Technology*, 2nd ed. CRC Press Inc, 2001.

[8] A.K.Jain, P.Flynn, and A.A.Ross, *Handbook of Biometrics.* Springer Science+Business Media, LLC., 2008.

[9] B. Dorizzi, R. Cappelli, M. Ferrara, D. Maio, D. Maltoni, N. Houmani, S. Garcia-Salicetti, and A. Mayoue, *Fingerprint and On-Line Signature Verification Competitions at ICB 2009.* Springer, June 2009, pp. 725–732, in proceedings of the International Conference on Biometrics (ICB), Alghero, Italy.

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

[11] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “Imagenet classification with deep convolutional neural networks,” in *Advances in neural information processing systems*, 2012, pp. 1097–1105.

[12] L.-C. Chen and Y.Zhu, “Semantic image segmentation with deeplab in tensorflow,” March 2018. [Online]. Available: https://ai.googleblog.com/2018/03/semantic-image-segmentation-with.html?utm=1

[13] L. Engstrom, D. Tsipras, L. Schmidt, and A. Madry, "A rotation and a translation suffice: Fooling cnns with simple transformations," *CoRR*, vol. abs/1712.02779, 2017. [Online]. Available: http://arxiv.org/abs/1712.02779

[14] I. Goodfellow, Y. Bengio, A. Courville, and Y. Bengio, *Deep learning.* MIT press Cambridge, 2016, vol. 1.

[15] X. Glorot, A. Bordes, and Y. Bengio, "Deep sparse rectifier neural networks," in *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, 2011, pp. 315–323.

[16] A. L. Maas, A. Y. Hannun, and A. Y. Ng, "Rectifier nonlinearities improve neural network acoustic models," in *Proc. icml*, vol. 30, no. 1, 2013.

[17] R. Hadsell, S. Chopra, and Y. LeCun, "Dimensionality reduction by learning an invariant mapping," in *Computer vision and pattern recognition, 2006 IEEE computer society conference on*, vol. 2. IEEE, 2006, pp. 1735–1742.

[18] F. Schroff, D. Kalenichenko, and J. Philbin, "Facenet: A unified embedding for face recognition and clustering," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp. 815–823.

[19] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, A. Rabinovich *et al.*, "Going deeper with convolutions." Cvpr, 2015.

[20] S. Sabour, N. Frosst, and G. E. Hinton, "Dynamic routing between capsules," in *Advances in Neural Information Processing Systems*, 2017, pp. 3859–3869.

[21] J. Taubert, D. Apthorp, D. Aagten-Murphy, and D. Alais, "The role of holistic processing in face perception: Evidence from the face inversion effect," *Vision research*, vol. 51, no. 11, pp. 1273–1278, 2011.

[22] G. Hinton, "Taking inverse graphics seriously," 2013. [Online]. Available: https://www.cs.toronto.edu/~hinton/csc2535/notes/lec6b.pdf

[23] D. J. Eck, *Introduction to Computer Graphics*, January 2018, version 1.2.

[24] J. Neyman, "Some methods for classification and analysis of multivariate observations," *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability*, vol. 1, pp. 281–297, 1967.

[25] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *Proceedings of the 32Nd International Conference on International Conference on Machine Learning - Volume 37*, ser. ICML'15. JMLR.org, 2015, pp. 448–456. [Online]. Available: http://dl.acm.org/citation.cfm?id=3045118.3045167

[26] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *The Journal of Machine Learning Research*, vol. 15, no. 1, pp. 1929–1958, 2014.

[27] Google, "tf.layers.batch_normalization," April 2018. [Online]. Available: https://www.tensorflow.org/api_docs/python/tf/layers/batch_normalization

[28] ——, "tf.nn.leaky_relu," April 2018. [Online]. Available: https://www.tensorflow.org/api_docs/python/tf/nn/leaky_relu

[29] I. Sutskever, J. Martens, G. Dahl, and G. Hinton, "On the importance of initialization and momentum in deep learning," in *International conference on machine learning*, 2013, pp. 1139–1147.

[30] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *CoRR*, vol. abs/1412.6980, 2014. [Online]. Available: http://arxiv.org/abs/1412.6980

[31] A. Heckert and J. J. Filliben, "Weighted variance," in *Dataplot Reference Manual*. NIST, September 1996, vol. 2, ch. 2, p. 68.

13 Appendix

13.1 Inception modules

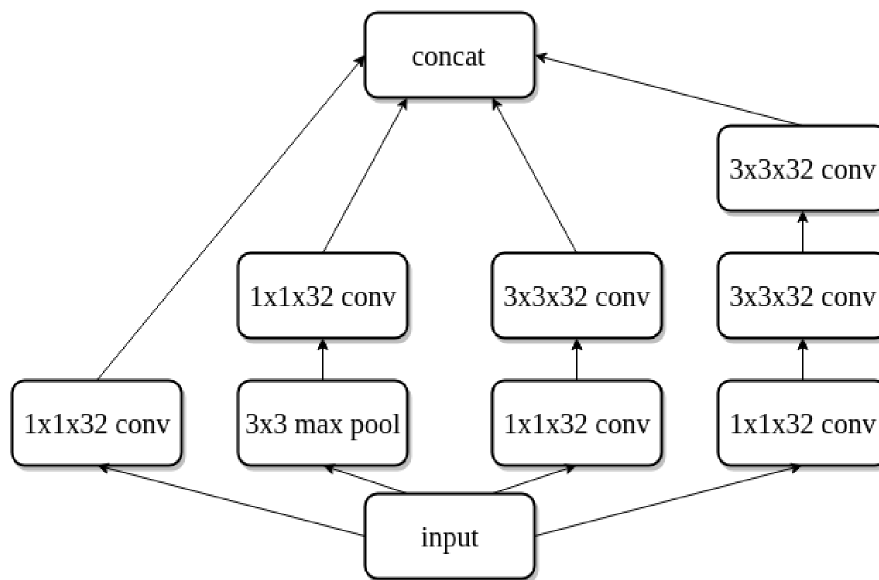

Figure 40: Illustration of inception_a module. All convolutional blocks are followed by batch normalization and leaky-relu. If stride and padding are not specified within pertinent blocks we assume a stride of 1 and same padding. The concatenation block (concat) aggregates its inputs along the last dimension, which corresponds to concatenating all channels.

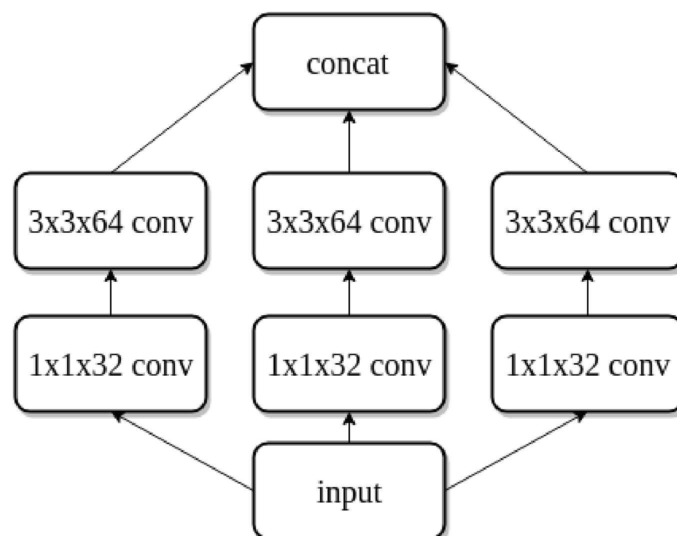

Figure 41: Illustration of inception_b module. All convolutional blocks are followed by batch normalization and leaky-relu. If stride and padding are not specified within pertinent blocks we assume a stride of 1 and same padding. The concatenation block (concat) aggregates its inputs along the last dimension, which corresponds to concatenating all channels.

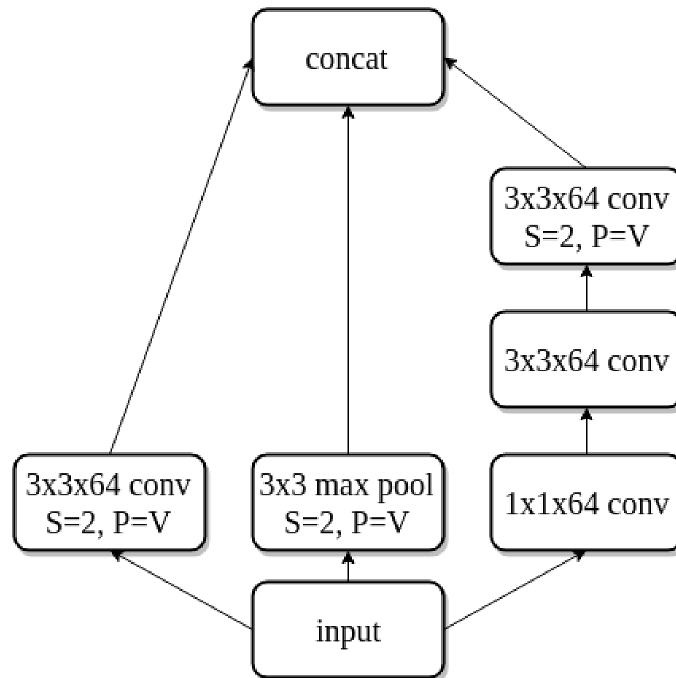

Figure 42: Illustration of reduction module. All convolutional blocks are followed by batch normalization and leaky-relu. If stride and padding are not specified within pertinent blocks we assume a stride of 1 and same padding. Valid padding is indicated by P=V and strides different from one with S=n, where n is an integer larger than 1. The concatenation block (concat) aggregates its inputs along the last dimension, which corresponds to concatenating all channels.

13.2 Loss functions for capsule networks

13.2.1 Triplet loss for capsules

As the triplet network had the best performance it could be interesting to use a triplet loss function tailored to capsules when trying to increase performance of the capsule network. The difference between the regular triplet loss, equation (12), is that the output from a capsule network are vectors instead of scalars which introduces an extra summation over all capsules in the output layer. The loss function is given as

$$L = \frac{1}{N}\sum_{i}^{N}\sum_{j}^{C} \max\left(0, \left\|\mathbf{v}_j^{(a)} - \mathbf{v}_j^{(p)}\right\|_2^2 - \left\|\mathbf{v}_j^{(a)} - \mathbf{v}_j^{(n)}\right\|_2^2 + \alpha\right), \tag{26}$$

where N is the number of image triplets in a batch, $\mathbf{v}_j$ is output capsule j, (a), (p), and (n) denote anchor, positive and negative images respectively in the triplet. C is the total number of capsules in the output layer and α is a margin hyperparameter.

13.2.2 Scaled Contrastive loss for capsules

To make the contrastive loss more sensitive to lengths of the output capsule, i.e. the probability of entities, we suggest the use of a scaled contrastive loss where the loss contribution from matching images is scaled with the inverse of the sum of lengths of output capsules. The contribution to the loss for the non matching images is also scaled with the sum of the lengths of the output capsules. By doing this we will discourage the network to have high probability in output capsules when the inputs are non matching, while on the other hand enforce long vectors for matching cases. Using the notation in (20) and the original contrastive loss for capsules, equation (21), we define the scaled constrastive loss

$$\begin{aligned} L = \frac{1}{2n}\sum_{k=1}^{n}\sum_{j=1}^{C} l_k (D_j^k)^2 \frac{1}{\|\mathbf{v}_j(\mathbf{i}_1)\|_2 + \|\mathbf{v}_j(\mathbf{i}_2)\|_2} \\ + (1-l_k)\max\left(0, m - D_j^k\right)^2 \left(\|\mathbf{v}_j(\mathbf{i}_1)\|_2 + \|\mathbf{v}_j(\mathbf{i}_2)\|_2\right), \end{aligned} \tag{27}$$

where all notations are explained in Section 7.2.1. This loss function has the advantage of utilizing the inherent orientational loss contribution of the contrastive loss for capsules while the scaling will push the network to produce outputs that have a good mix of active and non active capsules.

13.2.3 Agreement loss

In order to take full advantage of the capsules ability to encode both orientation and existence of features we propose a different approach than those suggested above. This loss function will take into account how the orientation of the output capsule between images change by letting the variance of change in orientation of the most active features contribute to the loss. The variance measure will then be scaled by the agreement of existence of features between the images. To make matching and non matching pairs contribute differently a contrastive loss type approach with a margin and use of ground truth labels is adopted.

Let active capsules be those whose length exceeds a hyperparameter threshold value, denoted by α. Moreover let $\mathbf{w}_1^*$ be the vector from the first branch of a siamese network, whose elements are defined by

$$w_{1j}^* = \max(0, \|\mathbf{v}_j(\mathbf{i}_1)\|_2 - \alpha). \tag{28}$$

This vector contains non-zero positive elements for active capsules and zero otherwise. $\mathbf{v}_j(\mathbf{i}_1)$ denotes output capsule j from the first branch. The same calculation is made for the second branch and is denoted $\mathbf{w}_2^*$. To find out which active capsules from the first branch that have a corresponding active capsule in the second branch elementwise multiplication between the two vectors is calculated

$$\mathbf{w} = \mathbf{w}_1^* \odot \mathbf{w}_2^*. \tag{29}$$

The angle between capsules in the two branches of the siamese capsule network is computed as the dot product between normalized capsules, resulting in the cosine of the angle between corresponding capsules in the two branches

$$c_j = \frac{\mathbf{v}_j(\mathbf{i}_1)}{\|\mathbf{v}_j(\mathbf{i}_1)\|_2} \cdot \frac{\mathbf{v}_j(\mathbf{i}_2)}{\|\mathbf{v}_j(\mathbf{i}_2)\|_2}.$$

In order to get the angle variance between corresponding output capsules a weighted variance is used, where the weights are the activity vector $\mathbf{w}$. As we want the capsule network to be equivariant, changes in orientation of images should produce low variance. To get the weighted variance we first need to calculate the weighted mean

$$\mu^* = \frac{\sum_j w_j c_j}{\sum_j w_j},$$

where w_j is element j in $\mathbf{w}$. Now the weighted variance [31] is given by

$$V^* = \frac{\sum_j w_j(c_j - \mu^*)^2}{\frac{(N'-1)\sum_j w_j}{N'}}$$

,where N' is the number of non zeros weights in $\mathbf{w}$.

To make use of the length of capsules we will scale the weighted variance with the agreement on active/inactive capsules. In order to do this we need to find inactive capsules in both branches, $\mathbf{z}_1^*$ and $\mathbf{z}_2^*$. The same procedure as in equation (28) is used but with a different hyperparameter, β, that regulates the threshold for when capsules are considered inactive

$$z_{1j}^* = \max(0, \beta - \|\mathbf{v}_j(\mathbf{i}_1)\|_2).$$

The corresponding inactive capsules are found through an elementwise multiplication, as in equation (29)

$$\mathbf{z} = \mathbf{z}_1^* \odot \mathbf{z}_2^*.$$

To get the final agreement vector

$$\mathbf{a} = \mathbf{w} + \mathbf{z},$$

that contains both agreement for active and inactive capsules.

Now we are ready to define the loss for matching image pairs

$$L_M = \frac{V^*}{\sum_j a_j}$$

By dividing the weighted variance with the sum of agreement we reduce the loss contribution from the variance if the image pair has a lot of agreement and amplifies it in the other case.

In the non matching case the same approach as with all contrastive loss functions is used. Hence we introduce a margin, m, and use the labels l_k, in equation (10), together with the loss L_M to end up with the final loss function

$$L = \frac{1}{n}\sum_{k=1}^{n} l_k L_M^{(k)} + (1 - l_k)\max\left(0,\ m - L_M^{(k)}\right), \tag{30}$$

where n is the number of images pairs in a batch and $L_M^{(k)}$ is the loss associated with k-th image pair in a batch.