On-line Handwritten Signature Verification using Machine Learning Techniques with a Deep Learning Approach

Beatrice Drott, Thomas Hassan-Reza

Master's thesis
2015:E40

LUND UNIVERSITY

Faculty of Engineering
Centre for Mathematical Sciences
Mathematics

CENTRUM SCIENTIARUM MATHEMATICARUM

The Faculty of Engineering at Lund University, LTH

Master Thesis
at the Centre for Mathematical Sciences

On-line Handwritten Signature Verification using Machine Learning Techniques with a Deep Learning Approach

September 16, 2015

Author:
Beatrice Drott
Thomas Hassan-Reza

Supervisor:
Kalle Åström
Björn Samuelsson
Anders Sjögren
Peter Julin

Preface

This project is a master thesis for the Centre for Mathematical Science at the Faculty of Engineering, LTH, at Lund University. The work has been carried out at the company Anoto. We would like to thank our supervisors Kalle Åström, Björn Samuelsson, Anders Sjögren and Peter Juhlin. The help from Anoto has been key in developing the entire project and special thanks to Daniel Zaar and Johnny Sjöberg for the understanding of the equipment of Anoto.

B. Drott, T. Hassan-Reza
September 16, 2015

Abstract

The problem to be solved in this project is to distinguish two signatures from each other, with help of machine learning techniques. The main technique used is the comparison between two signatures and classifying if they are written by the same person (match) or not (no-match). The binary classification problem is then tackled with a few alternatives to better understand it. First by a simple engineered feature, then by the machine learning techniques as logistic regression, multi-layer perceptron and finally a deep learning approach with a convolutional neural network.

The evaluation method for the different algorithms was a plot of true positive rate (sensitivity) versus false positive rate (fall-out). The results of the alternative algorithms gave a different understanding of the problem. The engineered feature performed unexpectedly well. The logistic regression and multi-layer perceptron performed similarly. The main results from the final model, which was a max-pooling, convolutional neural network, were a true positive rate of 96.7 % and a false positive rate of 0.6 %.

The deep learning approach on the signature verification problem shows promising results but there is still room for improvement.

Contents

1 Introduction

Signatures have been used for centuries [2] to verify the identity of a person. In today's society signatures are used as a formal and important step in an agreement, but the correctness of the signature is not questioned before any legal issue arises.

This project will investigate if it is possible to connect a person to his/her signature. This will be done with different approaches for signature verification, using machine learning algorithms. The signatures in this project will be recorded by an Anoto pen, which measures *coordinates*, *angles*, *pressure* and *time*. The signatures will be written on ordinary paper compared to many previous signature verification methods, where digital tablets have been used.

The focus in this project will be on a deep learning algorithm, which is built on convolution. But even some simpler methods will be tried on the way. The deep learning model in this project is built on different non-linear layers, that will process the input data. The last layer will do a classification to decide if two signatures belong to the same person or are written by two different persons.

Deep learning is a form of artificial neural network, which has been compared with the human brain and it's development [23]. Biological neural networks were also a starting point in the research field of artificial neural networks in the 1960s [30], [31].

"... the infant's brain seems to organize itself under the influence of waves of so-called trophic-factors ... different regions of the brain become connected sequentially, with one layer of tissue maturing before another and so on until the whole brain is mature" [12].

An artificial neural network seems to have a similar organization as the human brain. A distinctive characteristic property for the human brain is that it remains relatively adaptable until late years, compared to many other animals. This enables the human brain to get a longer training period of the biological neural network on the changing features in the environment.

The idea of deep learning is to have layers of neurons, which will do a non-linear transformation of the input signal. Those neurons will together build a network and like the human brain the network will be trained to make specific decisions. The algorithm in this project will be trained to make one decision, distinguish if the two signatures are written by the same person or not.

1.1 The Signature Verification Problem

This section is intended to provide sufficient information about the general signature verification problem. It will discuss different approaches of signature verification and in Section 1.3 the approaches used in this project will be presented.

1.1.1 On-line and Off-line Signature Verification

There are two kinds of signature verification, **on-line** and **off-line** verification. Off-line verification is when the information from a picture of the signature is the only information that is available. This means that there is no information about how the strokes were drawn, in which order and at what velocity. In on-line signature verification the information about how the signature were drawn is available.

The on-line and off-line techniques differ a lot regarding the information belonging to the object, take for example an artist painting a painting. An on-line approach could be to place a video camera and look at each individual stroke from the paint brush and in the end see the entire painting, compared to the off-line approach, which could be to just look at the final picture.

In this project on-line signature verification is used. The information that is available in this project about how the signature was written are the pen tip coordinates, pressure, velocity and angles.

1.1.2 Different Classes of Forgeries

The forgeries can be separated into different classes, depending on how it was made. The classes can be [1] :

- **1:** Forger has no previous knowledge.
- **2:** Forger knows the name of the original writer.
- **3:** Forger can observe the written signature on a piece of paper.
- **4:** Forger has got access to the signature on a piece of paper and traces the signature.
- **5:** Forger has got access to the on-line information of the signature.

The first form of a forgery is where the forger has no knowledge what so ever. An example of this would be to just give a pen and a paper and say forge a signature. In an authentication system this can be considered to occur when a user enters the wrong user-name and tries to authenticate it with their own signature. This is the class of forgery tried out in this project.

The second class of forgery is when the forger has access to the name of the person they are trying to forge. Then some assumptions can be made on how the signature looks, for example, which letters it might contains.

But it is not as easy as in the third class where the forger has access to the off-line version of the signature. In this case the forger knows how the picture of the signature looks and can copy the signature. But the forger has no information about how the signature was written.

In the fourth class the forger can trace the signature, this means the image of the forged signature probably will look very similar to the original signature.

[1] This is the extended definition that was used by Mattisson [42].

The final class of forgeries is when the forger has some access to the on-line information of the signature. This class could probably be divided into different sub-classes, dependent on which on-line information the forger has access to.

Another aspect to remember is that the quality of the forged signature can vary depending on how artistic the forger is or how much the forger has practiced on writing the signature in advance. This will probably affect how easy it is to detect that a signature is a forgery.

Depending on which type of forgery is used, the signature verification process can take different forms.

1.1.3 Feature Extraction of the Signature

In the signature verification problem the features can be extracted in different ways. An **engineered feature** is some measurement conceived and made by a human. For example, the first approach to solve the signature verification problem in this project was made by an engineered feature, see Section 7.4. The engineered feature was to compere the difference between the coordinates of two signatures. This is in contrast to the other models, which used **machine learning features**, where the features not were made by the human hand. Instead learning algorithms were developed and the machine found the appropriate features on its own.

1.2 Related Work

This section discusses related work and the differences between the related work and the project presented in this report. This is an interesting section, because it can give valuable information about how to analyze the approach carried out in this project and also give hints of future improvements that can be made.

1.2.1 Master Thesis - On-line Signature Verification using a Multi-judge Strategy

The work in *On-line Signature Verification using a Multi-judge Strategy*[42] is also about on-line signature verification, but the approach differs from the the techniques used in this project. The equipment used in [42] was a digital tablet (a Wacom pad), compared to ordinary paper which is used in this project. The measured information about the signatures, where only the coordinates and their time stamp, and not as in this project where also information about angles and pressure are available.

In [42], the verification was made by multiple engineered features and not by machine learning techniques.

There is also a difference in how the features were applied to the signatures. In [42] each feature was used as a gate, and if the signature was accepted as genuine by the first feature-gate, the next feature-gate was check, and so on. For a signature to be accepted as genuine it should be accepted as genuine by all the engineered feature-gates. In other words, the features were applied to

the signature in sequential order, rather than in parallel. Logically this looks like this,

$$\text{System judgement} = feature_1 \,\&feature_2\, \& \,\cdots\, \&\, feature_n, \qquad (1)$$

where *system judgment* is the finally decision, and feature *1* to n are all the engineered features that were checked and together with a threshold (as in Equation 2 later in Section 2.1) formed a gate. In this project, different features will be applied to the signature in parallel, and each feature will be assigned a weight, compared to the method in [42].

The collected database in [42], contained individual engineered templates for each genuine signature and some effort was made to calculate individual thresholds for all the feature gates belonging to each signature and this process was not automatic. In the project presented in this report, a real signature can be compared with an arbitrary signature, that is, the signature that should be checked. In this way there are no individual, engineered thresholds for the different persons' signatures.

The engineered features that were used in [42] were, for example:

- The number of strokes, if the number of strokes exactly matches the number of strokes in the template in the database (in this project it has been seen that the number of strokes often are constant, but there are exceptions).
- The time. A forgery had usually a very different time profile.
- Stretching. In [42] it was observed that if the signature was scaled in only one dimension, it often was a sign of being a forgery.

In [42] it was discussed that there are two kind of variations, namely inter-class and intra-class variation. Where **inter-class** variation is the differences between different classes of genuine signatures and **intra-class** variation is the variance within a persons signature, it can for example be described with scaling, rotation and shearing. In [42] **stability** was discussed, which is when the intra-class variation is small, and it has been observed that the stability varies a lot between different writers. This means that some writers are more suited for signature verification than others.

In the project presented in this report the features will be found during training, by the artificial neural network and all the features are handled as a black box. In this project the different features are not investigated or analyzed further. In general it is difficult for a human to understand the reason for the output of a neural network, a machine learning technique that is easier for a human to understand the reason of its outputs, is *decision trees*, which is another common machine learning technique.

There are different classes of forgeries, see Section 1.1. In this project the forgeries could be said to be in class 1, because here is only genuine signatures distinguished from each other. However, in [42] forgeries of class 3 and 4 were used.

To summarize, some of the positive and negative aspects of the two different methods can be:

- The final machine learning technique used in this project will find more features than in [42]. It will also find features on different global levels, see Section 3.4.4.
- The features in this project are applied in parallel and weighted, which may be better, due to the structure of the problem.
- The verification process is more automated in this project compared to the method used in [42], where each personal template were engineered.
- One advantage in [42] is that the outcome from the algorithm is easier for a human to understand, compared to the output from a neural network. This can give an insight of further development of the final algorithm in this project.
- The equipment used in this project gives the writer of the signature, a more genuine signature signing experience. Since the signature is written on an ordinary piece of paper and not on a digital tablet. Additionally there is more information (pressure and pen-orientation) recorded about the signature.
- It is not possible to compare the result of those two methods, because the database was based on different classes of forgeries.

1.2.2 Other Master Theses

There has also been some more work done on on-line handwriting recognition (HWR) by [44], [24], [19], [49] and [8]. But HWR has a completely different goal than signature verification has. In HWR the goal is to recognize the different characters of the handwritten words and not to verify who has written it. In [44] the work was focused on recognizing cursive handwriting. Where the whole word was written in one stroke, and the challenge was to separate the characters from each other. In [19] Cyrillic handwriting recognition was carried out with the help of support vector machines. The Cyrillic alphabet includes the Russian, Serbian, Ukrainian and Bulgarian alphabets. In [49] edit distance and linguistic knowledge were used to improve the HWR and in [8] a database for the Arabic language was created.

1.2.3 International Signature Verification Competition - SVC2004

In 2004 the first international Signature Verification Competition (SVC2004) was organized, and is described in the report *SVC2004: First International Signature Verification Competition* [58]. The objective was to allow researchers to compare the performance of different signature verification systems systematically based on common benchmarks. In this case two different databases and

bench-marking rules were used. The organizer made it clear from the beginning that the event should not be considered as an official certification exercise, since the databases used in the competition were only acquired in a laboratory rather than in a real environment. The performance of the system can vary significantly, dependent on how the forgeries are provided and dependent on the language. For example if the signature is written in an Arabic language it is written from right to left compared to the western approach left to right, this can results in different features [58].

The competition in [58] contained two tasks. The two tasks used different databases and the information available in the databases were a bit different.

- Task 1: only information about the coordinates and the time stamp of each data point were available.
- Task 2: the coordinates, the pen orientation, the pressure and the time stamp of each data point were available.

It is interesting for this project to compare the data collecting procedure with the procedure used in this competition, to be able to analyze and improve this in future work.

In the competition, each task had a database with 100 sets of signature data. Each set contained 20 genuine signatures from one signature contributor and 20 skilled forgeries from at least four other contributors. Of the 100 sets of signature data, the first 40 sets were released (25 October 2003) to the participants for developing and evaluating their systems before submission (31 December 2003). So the **development set**, consisted of 40 sets. The remaining 60 sets, were released on the competition day, and the algorithms were tested on those data sets. This is called **secret test set**, and means that the participants do not have access to the test set before the contest. The set will then consist of unseen samples to the algorithm.

For privacy reasons the contributor was advised not to use their real signature which they use in daily life. Instead they designed a new signature and practiced writing it sufficiently, so that it remained relatively consistent over different signature instances, just like a real signature. The contributors were asked to do the signatures in a way so it should be consistent both in spatial and in dynamic features. This approach to collect "toy" signatures, solved some problem that arises due to ethical and social implications, see Section 2.2. The signature contributor gave first 10 signatures, and controlled that they were satisfied with it. One week later they gave 10 more. For the skilled forgeries, each data contributor got a software, where the viewer could see the genuine signature and even replay the writing sequence of the signature on the screen, the forgeries were in other words of class five, see Section 1.1.2. They were also told to practice writing the signature before they did the forgeries. The signature were collected with a digital tablet (Wacom Intuos tablet). There were 15 teams for Task 1 and 12 teams for Task 2 (from Australia, China, France, Germany, Korea, Singapore, Spain, Turkey and United States). For both the tasks, it was team 6 from Sabanci University of Turkey that got the lowest error

rate, for task 1 they got 2.84% and for task 2, they got 2.89% [58]. Those results are of interest for future work of the project presented in this report.

It is interesting that they got better results on task 1, where there were less information about the signature available.

As implied in the report *SVC2004: First International Signature Verification Competition* [58], the reason for this can be, that the additional dynamic information may not be useful and can instead lead to impaired performance. But the author of this project think there is a possibility that this can depend on the quality of the measured parameters, or the the quality of the database. But this is an interesting subject to analyze further in future work of this project, and in case of impaired performance, optimize the classifier to work better on the additional information.

To see the complete result, see [45].

1.3 Goal of the Project

The main goal in this project is to implementation some form of a *deep learning* algorithm and investigate how well it perform on the signature verification problem.

This can be divided into sub goals.

- This involves investigation into the subject of the signature verification problem and applying different solutions to the problem.
- To apply different solutions a code framework will be developed
- Neural networks with deep architecture, will be analyzed and a practical solution will be developed.

1.4 Overview of Thesis

Firstly the background surrounding the signature verification problem is given. The handwritten signature is explained as a biometric parameter and a discussion about what this means is provided.

Later machine learning will be explained and important concepts that later systematically leads to the explanation of the models and algorithms used in this report. Thereafter as the complexity increases, artificial neural networks theory will also be explained.

The MINIST Classification problem is used to exemplify machine learning. It was the tutorials for this data set that was the starting point for the development of the models used in this project.

Later on, neural networks are explained further with examples, history about deep learning and results from contests on the subject.

After the theoretical part has been described the practical part begins by defining the software and hardware used in the project. Continuing with the methods, results and discussion.

Finally the report ends with suggestions of future work and a conclusion.

2 The Signature as a Biometric Parameter

The ability to recognize a person plays a crucial rule in a number of applications. Some examples are, regulating internal border crossings, restricting physical access to important facilities like nuclear plants or airports, controlling logical access to shared resources and information, performing remote financial transactions or distributing social welfare benefits [33].

A person can be recognized based on three basic methods:

- What he/she knows. The person has exclusive knowledge of some secret information, for example password, personal identification number, or cryptographic key [33], [34].
- What he/she possesses extrinsically. The person has exclusive possession of an extrinsic token, for example identification card, driver's license, passport, physical key or personal device such as a mobile phone [33], [34].
- Who he/she is intrinsically. This is based on the person's identity and his/her inherent physical or behavioral traits. This is known as **biometric recognition** [33], [34].

Formally, biometric recognition can be defined as the science of establishing the identity of an individual based on the physical and/or behavioral characteristics of the person either in a fully automated manner or in a semi-automated manner [33], [34].

Biometric recognition offer a natural and more reliable solution for identity recognition, compared to what knowledge-based and token-based person recognition does. For example a password can be guessed, shared or forgotten by the user and a token can be forged, stolen or lost. But at the same time biometric recognition open up for new questions and problems, that are discussed further in Section 2.2.

To achieve a secure system, it is important not only to use mechanisms like passwords and tokens, but also use biometric recognition and the different methods can form multiple layers of security.

The biometric parameters are categorized as **physical biometrics** and **behavioral biometrics** [34], [33].

The physical biometric parameters are related to the shape of the body, for example DNA, ear, face, fingerprint, hand geometry, hand vein, iris, palm print, retina and odour [34].

The behavioral characteristics are related to the pattern of behavior of a person, for example typing rhythm, gait, handwriting and signature. Then there is voice, which is a combination of both physical and behavioral biometrics [34].

The primary advantages signature verification has over other types of biometric technologies is that the handwritten signature is already the most widely accepted biometric for identity verification in daily use. It is the most common used in connection with contracts and agreements. If it was possible to get a good signature verification system, with a low error rate, this should open up new possibilities to verify the correctness of the daily use verification procedure.

2.1 Biometric Recognition Systems in General

A biometric recognition system is essentially a pattern recognition system that operates by acquiring biometric data from an individual, extracting a feature set from the acquired data and comparing this feature set against a set of templates in the database. The biometric system can operate in either **verification** mode or **identificarion** mode [34].

- Verification: the system validates a person's identity by comparing the captured biometric data with a template (or templates), that has earlier been stored for this person. The template usually is found in the database by a PIN (Personal Identification Number). Verification typically answer the question *"Does this biometric data belong to the person that this person claims he/she is?* [34].
- Identification: the system recognize a person by searching through the database of templates. Identification typically answer the question *Whose biometric data is this?* [34].

The generic term **recognition** will in this context stand for both verification and identification [34], and is useful to use when it is not necessary to separate the two concepts apart.

A **biometric recognition system** consist of both an **enrollment** phase and a recognition phase [34], see Figure 1.

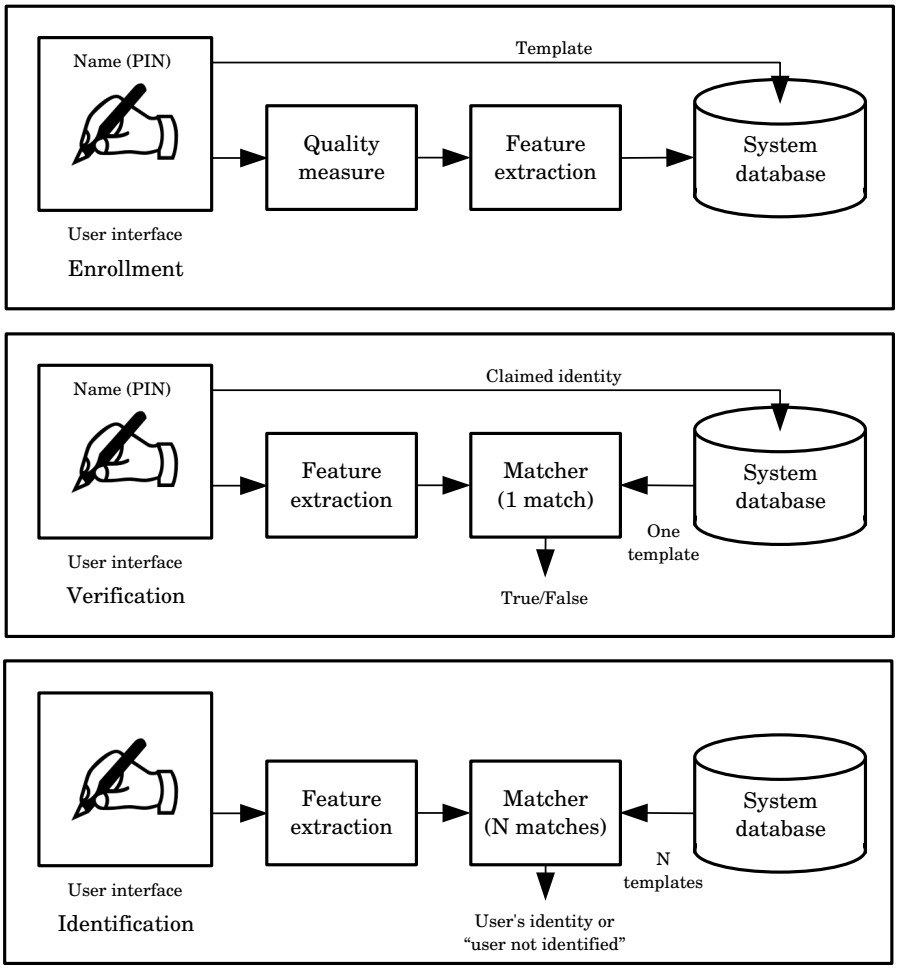


Figure 1: Block diagrams over how an usual biometric recognition system works. The system consist of tree phases and they are enrollment, verification and identification. The picture is based on [34].

The enrollment phase is used when collecting data to the database. The biometric data is collected and checked by a **quality measure**, to ensure that the data is good enough. Then features are extracted from the data and stored in the database.

In the project presented in this report, the *raw-data* were saved without first performing the feature extraction. The **raw-data** is the data that are collected before any pre-processing or feature extraction are done, see more of those steps in Section 7. In the project presented in this report, the pre-procession and feature extraction are a step in the development of the models, so it was more practical to save the original raw-data in the data base. The reason to why the feature extraction usually is performed in the enrolment phase, is to minimize the storage requirements. But in the development phase it is more convenient to save the original raw-data.

The verification function f takes an input feature vector, X_Q, and the claimed identity I, and determines if $f(I, X_Q)$ belongs to class c_1 or c_2, where c_1 indicates *true* and c_2 indicates *false*. Typically, X_Q is matched against X_I, which is the biometric template corresponding to user I. This can be written as

$$f(I, X_Q) = \begin{cases} c_1 & \text{if } S(X_Q, X_I) \geq t, \\ c_2 & \text{otherwise,} \end{cases} \tag{2}$$

where S is a function that measure the similarity between X_Q and X_I, and t is a predefined *threshold*. The value of $S(X_Q, X_I)$ is called a **similarity** or **matching score** between the biometric measurements of the user and the claimed identity. Later in Section 3.3.4 will a *soft* threshold be discussed, which is dependent on the similarity score.

The identification function g instead takes the maximum of the templates that is true under the condition if $S(X_Q, X_I) \geq t$. This can be written as

$$g(I, X_Q) = \begin{cases} I_k & \text{if } \max_k\{S(X_Q, X_{I_k})\} \geq t,\ k \in \{1, \ldots, N\}, \\ \text{user not identified} & \text{otherwise,} \end{cases} \tag{3}$$

where X_{I_k} is the biometric template corresponding to identity I_k in the database and t is the predefined threshold.

Especially for the behavior based biometric parameters, the data in the database has to be collected in intervals, because the parameters can change over time. [7]

2.2 Ethical and Social Implications

This section will discuss the ethical and social implications that arises when the signature verification problem is handled.

When it comes to capturing and storing signatures as a biometric parameter, this will of course raise some questions regarding the ethical and social implications. Since a biometric parameter is a personal quality, in this case a behavioral biometric, the questions that arises are concerning personal and

individual rights. The signature can be considered as personal data that only the person in question should have access to. Looking back at Figure 1 there is a system database which contain all the signatures. In this case the signature can be considered as a sophisticated password and something you would like to protect. In general the principle *"the less you know, the harder it gets"* applies.

When looking at an article published by the *Data Protection Working Party of the European Union* an interesting argument arises. The article refers to a directive which states that "... personal data must be collected for specified, explicit and legitimate purposes and not further processed in a way incompatible with those purposes.", [20]. There are two main fears expressed in this directive.

Firstly that personal data might be collected for unspecific, implicit and non-legitimate reasons. An example of this would be to collect a lot of signatures without a specific motive. Then after the data was collected decide what to do with it. An example of a non-legitimate reason would be to collect signatures in the objective of committing crimes. Some examples of possible legal violations would be forging of signatures in the intent of identity theft and financial or insurance fraud.

Secondly the data might be collected in a way that is compatible to the purpose but later on be used in a completely different way. This second fear is known as **function creep** where the continued development of the technology stretches beyond its original intent. [22] Especially when the technology starts to invade the privacy of individuals.

2.2.1 Important Security Measures for Biometric Systems

The European Data Protection Working Party has identified some important factors that acquires consideration [20] regarding security measures. Whenever biometric data is processed (i.e stored, transferred, features are extracted or compared) there is a risk that the personal privacy is breached. This could be done by the data being destroyed, viewed by a third party or altered.

In a social point of view it is important to understand where the security weaknesses of a biometric system lies. First of all, during the enrollment phase when the biometric data is collected. The data should be saved with sufficient encryption in a database. If somehow a signature which was unauthorized was associated with a person who is authorized then the unauthorized person was just able to steal the authorized identity. So during the enrollment phase checking the identity of the person is highly important.

When considering that errors can occur in the algorithm (i.e. someone is falsely rejected or falsely accepted) this can be of real problems for the individual at hand. For example if the system falsely reject a person based on his/her signature the trust in the technology could be so high that the rejection is counted as "indisputable" evidence. Since all biometrical systems are designed to reduce the risk of errors this creates the illusion that the system is always correct.

3 Theoretical Background and Introduction Regarding Methods used in this Thesis

In this section the theoretical background to the techniques that are used in this project will be described. It will start with an introduction to the field of machine learning and then systematically lead to the techniques that are used in the developed models of this project.

3.1 Introduction to Machine Learning

A computer program is considered to be learning if it improve its performance on future tasks after making observations about the world. Why would we want our program to learn? There are different reasons, for example,

- The programmer can not anticipate all possible situations the program (agent) might find itself in [48].
- The programmer can not anticipate all changes over time [48].
- Sometimes the programmer has no idea of how to program a solution by themselves [48]. For example a human is great at recognizing faces of family members, but it is considered highly difficult to construct a computer program to do the equivalent, without using learning algorithms.[2]

Machine learning can be categorized into three broad classes, depending on the type of **feedback** the system has.

- **Supervised Learning**:
  The program is given example input-output pairs and learn a function that maps input to output [48]. This is the technique used in this project, see more on this in Section 3.2.
- **Unsupervised Learning**:
  The program has to find a structure in the input on its own. The most common unsupervised learning task is **clustering** [48].
- **Reinforcement Learning**:
  The computer interacts with a dynamic environment in which it must perform a certain goal, and a teacher telling it, if it has come close to the goal or not. An example is learning to play a game by playing against an opponent. The program learns from a series of reinforcements, rewards or punishments [48].

Between supervised and unsupervised learning is **semi-supervised** learning, which uses a combination of the two methods [48].

[2] The work in this project is based on algorithms used in image recognition. Then translated and modified for the signature verification problem.

3.2 Learning From Input-output Pairs - Supervised Learning

Supervised learning is about learning a function from a collection of input-output pairs, this is also called **inductive learning** [48].

Mathematically the input-output pairs can be described as

$$\mathcal{D} = (\mathbf{x_1}, y_1), (\mathbf{x_2}, y_2), \cdots, (\mathbf{x_N}, y_N), \tag{4}$$

where $\mathbf{x_i}$, $i \in (1, N)$, is some data belonging to each example and y_i, $i \in (1, N)$ is generated by some unknown function $y_i = f(\mathbf{x_i})$. The supervised learning is about discovering a function $h(\mathbf{x}) = \hat{y}$ that approximates $f(\mathbf{x}) = y$.

To measure the accuracy of the function $h(\mathbf{x})$, a **test set** is used, which is separated from the **training set**. The function $h(\mathbf{x})$ is said to **generalizes** well if it correctly predict the output, that is if $\hat{y} = y$ is true, for the new examples in the test set. If the output y is a finite set of values, the learning problem is called **classification** (see Section 3.3) and if the output only has two classes it is a boolean or **binary classification problem**. If the output is a continuous number the learning problem is called **regression**.

The **error rate** is then calculated as,

$$\text{error rate } = \frac{\sum_{i=1}^{N} g(y_i, \hat{y}_i)}{N}, \tag{5}$$

where,

$$g(y_i, \hat{y}_i) = \begin{cases} 1 & \text{if } y_i \neq \hat{y}_i \\ 0 & \text{otherwise.} \end{cases}$$

To have two separate sets, a training set and a test set, is the simplest way to measure the error rate in an accurate way. However, in some cases this method can give an inaccurate measurement. This may happen if there is not enough samples available in the database or if there is not a good distribution and spread of the samples when it was partitioned into the two separate sets. But there are other ways to predict the performance, for example **k-fold cross-validation**, where the whole data set of examples is divided into k complementary subsets. The error rate is then measured as in Algorithm 1, where in each round the model is trained on all but one of the sets. Then the trained model is tested on the subset that was left out during training. The error rate is then given by taking the *root mean squared* error rate on all the separate measurements. The problem described with the conventional method there only two separate sets are used, can be avoided with cross validation, due to that the error rate is measured as a mean value.

In this project is the conventional method used and not cross-validation. This is because the final model in this project took very long time to train. But to avoid the problems described above the samples in the total database were shuffled randomly before the set was divided into the separate sets, see more on this in Section 7.3.

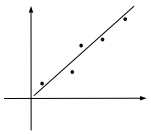
Algorithm 1 Calculate error rate with help of k-fold cross-validation

for each $subset_i \in subsets$ **do**
  $h_i \leftarrow$ train model on set, $(subset_i^C \cap subsets)$
  $error_i \leftarrow$ measure error rate on $subset_i$ with model, h_i
end for
error rate $\leftarrow \frac{\sum_{i=1}^{N} error_i}{N}$

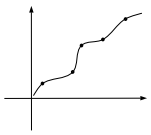
The reason to keep the test and training set separated is that it is not desirable to let the test set influence the training. If it happen anyway, it is called **peeking** and it can invalidate the model, because it is not certain the model then will generalize well to unseen examples.

But it is possible to let the performance on unseen data guide the training, and help to select a good model, but in this case a third separate set has to be used, a **validation set**.

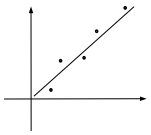
The validation set can be used to ensure that the model does not **overfit** or is **over trained**. A model is over trained if it has been trained to even detect patterns in the noise of the training data.



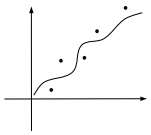
(a) Data set A, model C.



(b) Data set A, model D.



(c) Data set B, model C.



(d) Data set B, model D.

Figure 2: Data set A is the training set, which model C and D are trained on. Data set B is a test set with unseen examples. As can be seen in the figures is that model C generalizes better to the unseen examples in data set B. Model D has higher complexity than moden C, because the model is based on a higher degree polynomial than model C. In this case is the model D over-fit.

For example, assume some input data, which output are linearly dependent on the input, but due to the noise it does not fit a straight line exactly, see Figure 2a. Then there probably is a high degree polynomial that fit just this input data much better, see Figure 2b. But this high degree polynomial will probably not generalize well to unseen examples, this is shown in Figure 2d, compared to the linear model, shown in Figure 2c.

To get a good model is both about **model selection** also called **hyper-parameter optimization** and **optimization**. In the example with the polynomial, the model selection is about which degree of the polynomial to choose

and the optimization is about minimize the error from the polynomial to the input data, that is, choose the coefficients of the polynomial.

In machine learning problems one often talks about **loss function** and not error function. This is because there are different types of errors. For example, a classifier that classifies email as spam or non-spam, it is considered worse to classify a non-spam as spam. Because this could result in a user missing an important email. In the signature verification problem it could be worse to classify a forged signature as a genuine, because this might imply that a forger has succeeded. So in machine learning, the utility of the output is often used instead. The **empirical loss** is then the average over all examples.

In many cases it is the **cost function** that are minimized instead (see Equation 15), this function measure both the empirical loss and the complexity of the model. In the example with the polynomial above, which was shown in Figure 2, the complexity is the degree of the polynomial. Model C in Figure 2, has low complexity, compared to model D in the same Figure, which has higher complexity.

3.3 Classification

In this section the theoretical background behind the classifier used in this project is given. The section starts with some easier case in lower dimensions which are easier to understand, this will serve as a motivation to the reader and will then be generalized to the more advanced techniques used in this project.

3.3.1 Linear Regression in One Dimension

The linear function that is dependent on one variable, a straight line, with input x and output y, has the form $y = b + w_1 \cdot x$. The value of y will depend on the value of b and w_1, and they will be called weights of the function. The vector $\mathbf{\Theta} = [b, w_1]$ is then the parameter-vector or weight-vector to the model.

If some input data x is given, one can form a **hypothesis** that this data follow a linear function. The hypothesis can be defined as, $h_{\mathbf{\Theta}}(x) = b + w_1 \cdot x = \hat{y}$ (where $\hat{y}$ is the prediction of y). The hypothesis can also be viewed as a **hypothesis space**, when different values of the parameters will give a space of different hypothesizes.

The task to find the values of the parameter-vector, when $h_{\mathbf{\Theta}}(x)$ is the linear function, that fits the input data x best is called **linear regression**. This can be done by finding the weights that minimize the empirical loss function. In this problem it is traditional to use the squared loss function, L_2. The function to be minimized will then be

$$Loss(h_{\mathbf{\Theta}}) = \sum_{j=1}^{N} (y_j - (b + w_1 \cdot x_j))^2, \tag{6}$$

where a summation over all training examples N are done. The optimal weight

vector is then found by

$$\mathbf{\Theta}^* = argmin_{\mathbf{\Theta}} Loss(h_{\mathbf{\Theta}}) \tag{7}$$

In this example the **weight space**, is defined by b and w_1, that span a plane in two dimension. The loss function can then be graphically shown in a space, where $Loss(\Theta)$, b and w_1 span a room in three dimensions. This function is **convex**, which means that there is only one minimum and that is the global minimum. This applies for all linear regression problem with a L_2 loss function. It is easy to find the solution of Equation 7, it is just to set the partial derivatives of $Loss(mathbf\Theta)$ to zero and solve the equation, that is,

$$\begin{cases} \frac{\partial}{\partial b} \sum_{j=1}^{N} (y_j - (b + w_1 \cdot x_j))^2 = 0 \\ \frac{\partial}{\partial w_1} \sum_{j=1}^{N} (y_j - (b + w_1 \cdot x_j))^2 = 0. \end{cases} \tag{8}$$

However, for the general problem the loss function will be minimized by **gradient decent** (where the loss function not necessarily is convex). This procedure is described by Algorithm 2, where α in the algorithm is the **learning rate**. The learning rate can be a fixed number or decay over time. In the later

Algorithm 2 Update rule for gradient decent

```
Θ ← any point in the weight space
loop until convergence:
    for each θ_i ∈ Θ do
        update rule of Θ // for example the one in Equation 9
    end for
end loop
```

case convergence is guaranteed, compared to the first case where the solution can oscillate around the minimum. If α is a fix, small value, the algorithm will come closer to the minimum, but it will take longer time.

The procedure in Algorithm 2, that cover **one training example** has the **update rule**,

$$\theta_i \leftarrow \theta_i - \alpha \frac{\partial}{\partial \theta_i} Loss(\mathbf{\Theta}). \tag{9}$$

The simple update rule in Equation 9, will be reused in several more complicated models in later sections and is worth to remember.

With the linear function dependent on one variable and the loss function, L_2, the partial derivatives in Equation 9 can be written as

$$\frac{\partial}{\partial \theta_i} Loss(\mathbf{\Theta}) = 2(y - h_{\mathbf{\Theta}}(x)) \frac{\partial}{\partial \theta_i} (y - h_{\mathbf{\Theta}}(x)),$$

and with $h_{\mathbf{\Theta}}(x) = b + w_1 x$ this gives

$$\begin{cases} \frac{\partial}{\partial b} Loss(\mathbf{\Theta}) = -2(y - h_{\mathbf{\Theta}}(x)) \\ \frac{\partial}{\partial w_1} Loss(\mathbf{\Theta}) = -2x(y - h_{\mathbf{\Theta}}(x)). \end{cases}$$

The update rule in this case is then given by

$$\begin{cases} b \leftarrow b + \alpha(y - h_{\mathbf{\Theta}}(x)) \\ w_1 \leftarrow w_1 + \alpha x(y - h_{\mathbf{\Theta}}(x)), \end{cases} \tag{10}$$

where the constant "2" has been included in α.

For N training examples it is the sum of each individual loss that should be minimized. The update rule for **N training examples** is given by,

$$\theta_i \leftarrow \theta_i - \alpha \sum_{\text{all examples}} \frac{\partial}{\partial \theta_i} Loss(\mathbf{\Theta}). \tag{11}$$

With loss function, L_2, and $h_{\mathbf{\Theta}}(x) = b + w_1 x$ gives,

$$\begin{cases} b \leftarrow b + \alpha \sum_{j=1}^{N} (y_j - h_{\mathbf{\Theta}}(x_j)) \\ w_1 \leftarrow w_1 + \alpha \sum_{j=1}^{N} x_j (y_j - h_{\mathbf{\Theta}}(x_j)). \end{cases} \tag{12}$$

This update rule uses a summation over all training examples and is called **batch gradient descent**. This may be very slow, because for each training step (i.e. each weight update), it has to loop through all training examples and it may be many steps until convergence.

But there is an alternative method called **stochastic gradient decent**, where one training example is chosen randomly in each step. The update rule for stochastic gradient decent in Algorithm 2 is then identical to Equation 9. With loss function L_2 and $h_{\mathbf{\Theta}}(x) = b + w_1 x$, this gives

$$\begin{cases} b \leftarrow b + \alpha(y_s - h_{\mathbf{\Theta}}(x_s)) \\ w_1 \leftarrow w_1 + \alpha x_s (y_s - h_{\mathbf{\Theta}}(x_s)), \end{cases} \tag{13}$$

where s = index of one randomly chosen training example.

A combination of batch gradient descent and stochastic gradient decent, is **mini-batch stochastic gradient descent (MSGD)**, which is like stochastic gradient decent, but instead of taking just one example, it take some more, a batch of examples.

3.3.2 Linear Regression in Higher Dimensions

The theory behind linear regression in one dimension can quite easy be extended to linear regression in higher dimensions. The hypothesis space will now look like,

$$h_{\mathbf{\Theta}}(\mathbf{x}) = b + \sum_i w_i x_i = b + \mathbf{w}^T \cdot \mathbf{x}. \tag{14}$$

For the linear regression in one dimension over-fitting was no problem, but in high dimensional space there is a possibility that some dimensions appear to be useful, when they actually are not. This will result in over-fitting. So in higher dimensions it is common to use **regularization**, which mean that it is

the cost function that should be minimized and not the loss function. The cost function measures both the loss and the complexity of the model, and it can be written as

$$cost(h_{\boldsymbol{\Theta}}) = \text{Empirical Loss}(h_{\boldsymbol{\Theta}}) + \lambda \text{ Complexity}(h_{\boldsymbol{\Theta}}). \tag{15}$$

For linear functions the complexity can be specified as a function of the weights and it is often defined by the norm of the weights. The complexity function is then given by

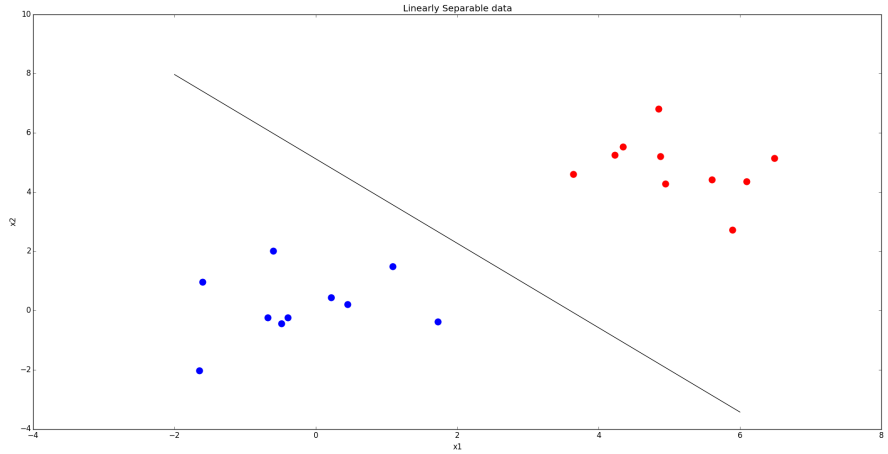
$$Complexity(h_{\boldsymbol{\Theta}}) = L_q(\boldsymbol{\Theta}) = \sum_i |\theta_i|^q. \tag{16}$$

Which regularization to use is up to the problem. In this project the L_1 norm is used for the multi-layer perceptron model and for the convolutional neural network, see more of those models in the Sections 7.6 and 3.4.4.

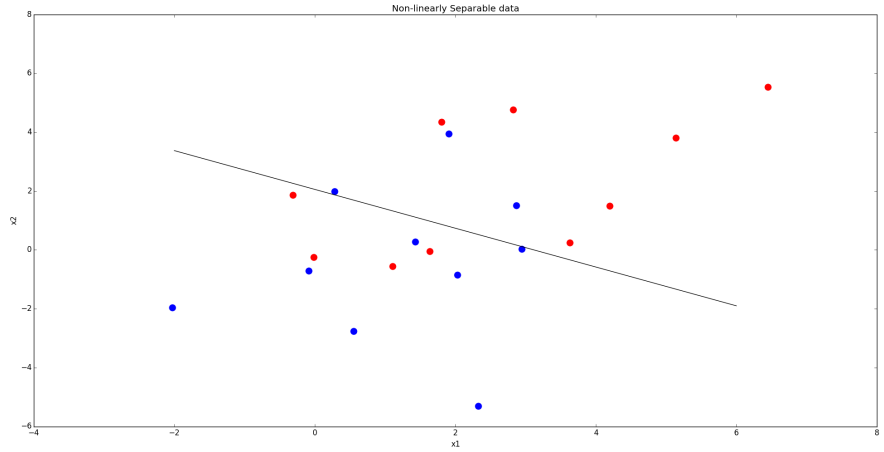
3.3.3 Linear Classifiers with a Hard Threshold

Recall from Section 3.2, the difference between regression and classification. The theory that was explained in Section 3.3.1 and 3.3.2 about regression, will now be modified to fit a binary classification problem.

In the classification problem, the linear function instead will be used as a **decision boundary**. In one dimension it is simply a line, and in higher dimensions it is a hyper plane. A linear decision boundary is called **linear separator** and if the input data can be separated by a linear separator, the data is called **linearly separable**. Figure 3 shows this.



(a) Example of linearly separable data, where the red and blue dots are data in two separate classes.



(b) Example of non-linearly separable data, where the red and blue dots are data in two separate classes.

Figure 3: The red and blue dots in the figures are data belonging to different classes. Picture (a) shows an example then the two data sets are linerly separable, compared to in picture (b).

In the binary classification problem one separator will be used to separate the two classes apart. The hypothesis space for the classification will then be,

$$h_{\mathbf{\Theta}}(\mathbf{x}) = \begin{cases} 1 & \text{if } b + \mathbf{w}^T \cdot \mathbf{x} \geq 0, \\ 0 & \text{otherwise.} \end{cases} \tag{17}$$

The hypothesis space can also be written as a **threshold function**, that is,

$$\begin{aligned} &h_{\mathbf{\Theta}}(\mathbf{x}) = Threshold(b + \mathbf{w}^T \cdot \mathbf{x}), \\ &\text{where } Threshold(z) = \begin{cases} 1 & \text{if } z \geq 0, \\ 0 & \text{otherwise.} \end{cases} \end{aligned} \tag{18}$$

The update rule (with loss function L_2) for one training example j, that converge to a linear classifier is given by,

$$\begin{cases} b \leftarrow b + \alpha(y_j - h_{\mathbf{\Theta}}(\mathbf{x})), \\ w_i \leftarrow w_i + \alpha x_{i,j}(y_j - h_{\mathbf{\Theta}}(\mathbf{x})). \end{cases} \tag{19}$$

The update rule is exactly the same as for linear regression, but the behavior is somewhat different.

The interpretation of Equation 19 is seen below.
Both y and $h_{\mathbf{\Theta}}$ are either 1 or 0, this gives three possibilities:

- The output is correct, that is $y = h_{\mathbf{\Theta}} \implies$ the weights are not changed.
- Prediction not correct, with $y = 1$ and $h_{\mathbf{\Theta}} = 0 \implies$ the weights are changed according to,
  if $x_i > 0$ **then**
  increase θ_i
  else
  decrease θ_i
  end if
- Prediction not correct, with $y = 0$ and $h_{\mathbf{\Theta}} = 1 \implies$ the weights are changed according to,
  if $x_i > 0$ **then**
  decrease θ_i
  else
  increase θ_i
  end if

3.3.4 Linear Classifiers with a Soft Threshold

There are some problems with a hard threshold. First thing is that the hypothesis function is discontinuous, and thereby not differentiable. The second

thing is that it does not say anything about how far away the input is from the boundary.

A solution to this is to instead use a soft threshold, that approximate the hard threshold with a continuous, differentiable function. A good function for this is the **logistic function**, also called the **sigmoid function**, which looks like this,

$$Sigmoid(z) = \frac{1}{1 + e^{-z}}. \tag{20}$$

The hypothesis will now look like

$$h_{\mathbf{\Theta}}(\mathbf{x}) = Sigmoid(b + \mathbf{w}^T \cdot \mathbf{x}) = \frac{1}{1 + e^{-(b+\mathbf{w}^T \cdot \mathbf{x})}}. \tag{21}$$

The logistic function gives an output value between 0 and 1, and can be interpreted as a *probability*.

The hypothesis will then form a soft boundary in the input space, see Figure 4. For input space at the center of the boundary region, the hypothesis function

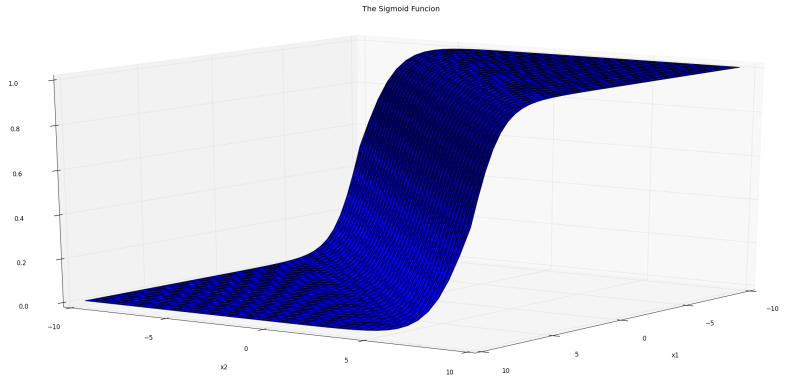


Figure 4: The sigmoid function, $sigmoid(x) = 1/(1 + e^x)$, is plotted on the interval (-10,10) and with argument $f(\mathbf{\Theta}) = x_1 - x_2$, that is $b = 0$, $w_1 = 1$ and $w_2 = -1$.

will give a value of 0.5, and far away from the boundary it approaches 0 or 1.

The process of fitting the weights to a binary classification problem with help of the logistic/sigmoid function is called **logistic regression**. Stochastic gradient decent can be used to find the weights and the update rule in Equation 9 can be used in Algorithm 2. The L_2-function can be used as the loss function. However, the derivative of the loss function in Equation 9, looks a little bit different and the *chain rule* has to be used more than once.

The derivation of the update rule is as follows:

$$\begin{aligned}\frac{\partial}{\partial\theta_i}Loss(\boldsymbol{\Theta}) &= \frac{\partial}{\partial\theta_i}(y - h_{\boldsymbol{\Theta}}(\mathbf{x}))^2 \\ &= 2(y - h_{\boldsymbol{\Theta}}(\mathbf{x})) \cdot \frac{\partial}{\partial\theta_i}(y - h_{\boldsymbol{\Theta}}(\mathbf{x})),\end{aligned}$$

with $h_{\boldsymbol{\Theta}}(\mathbf{x})) = sigmoid(b + \mathbf{w}^T \cdot \mathbf{x})$, this gives,

$$\frac{\partial}{\partial\theta_i}Loss(\boldsymbol{\Theta}) = -2(y - h_{\boldsymbol{\Theta}}(\mathbf{x})) \cdot sigmoid'(b + \mathbf{w}^T \cdot \mathbf{x}) \cdot \frac{\partial}{\partial\theta_i}(b + \mathbf{w}^T \cdot \mathbf{x}),$$

and $sigmoid'(x) = sigmoid(x)(1 - sigmoid(x))$, this gives,

$$\frac{\partial}{\partial\theta_i}Loss(\boldsymbol{\Theta}) = -2(y - h_{\boldsymbol{\Theta}}(\mathbf{x})) \cdot \underbrace{h_{\boldsymbol{\Theta}}(\mathbf{x})(1 - h_{\boldsymbol{\Theta}}(\mathbf{x}))}_{\text{derivate of sigmoid}} \cdot \underbrace{\frac{\partial}{\partial\theta_i}(b + \mathbf{w}^T \cdot \mathbf{x})}_{\text{inner derivate}}, \tag{22}$$

The update rules for b and w_i are then given by,

$$\begin{cases} b \leftarrow b + \alpha(y - h_{\boldsymbol{\Theta}}(\mathbf{x})) \cdot h_{\boldsymbol{\Theta}}(\mathbf{x})(1 - h_{\boldsymbol{\Theta}}(\mathbf{x})), \\ w_i \leftarrow w_i + \alpha(y - h_{\boldsymbol{\Theta}}(\mathbf{x})) \cdot h_{\boldsymbol{\Theta}}(\mathbf{x})(1 - h_{\boldsymbol{\Theta}}(\mathbf{x})) \cdot x_i. \end{cases} \tag{23}$$

3.3.5 Classifiers for Multiclass Classification Problems

Above in Section 3.3.4, a binary classification with a soft threshold was described. Logistic regression for binary problems can be generalized into multiclass classification problems, that is when the output has K possible outcomes rather than just two. In this case the **softmax** (also called normalized exponential) function will serve as an equivalent to the logistic function in the binary logistic regression.

The softmax function is a generalization of the logistic function, which operates on a K-dimensional vector $\mathbf{x}$ of arbitrary real values to a K-dimensional vector $softmax(\mathbf{x})$ of real values in the range $(0, 1)$. This is called **multi-class linear regression** [3]. The function is given by

$$softmax(\mathbf{x_j}) = \frac{e^{\mathbf{x_j}}}{\sum_{k=1}^{K} e^{\mathbf{x_k}}}, \tag{24}$$

where $j = 1, \cdots, K$.

The parameters in the multiclass classification problem will now look a little bit different from the parameters in Sections 3.3.1 - 3.3.4, where the weights were a vector, $\mathbf{w}$, and the offset, b, was a scalar. In this case will the weights be a matrix, $\mathbf{W}$, where the kth column represents the separation hyper plane for class k and the the offset is now a vector, $\mathbf{b}$, where element k represent the offset

[3] Multi-class linear regression is also known by other names as softmax regression, multinomial logistic regression, polytomous logistic regression, multinomial logit, maximum entropy (MaxEnt) classifier and conditional maximum entropy model [11].

parameter of hyper-plane k. The input, $\mathbf{x}$, is a matrix where row j represent input training example j.

The probability of the ith class is given by

$$\begin{aligned} P(y = i|\mathbf{x}, \mathbf{W}, \mathbf{b}) &= softmax(\mathbf{b} + \mathbf{W} \cdot \mathbf{x}) \\ &= \frac{e^{b_i + \mathbf{W}_i \cdot \mathbf{x}}}{\sum_{k=1}^{K} e^{b_k + \mathbf{W}_k \cdot \mathbf{x}}}, \end{aligned} \tag{25}$$

where $\mathbf{W}_k$ is the kth column in $\mathbf{W}$.

The output of the model or prediction is given by taking the argmax of the vector whose jth element is $P(Y = j|\mathbf{x})$. This takes the class with the highest probability.

$$\hat{y} = argmax_j P(Y = j|\mathbf{x}, \mathbf{W}, \mathbf{b}). \tag{26}$$

In multi-class logistic regression, it is common to use the negative log-likelihood (see Section 3.3.6) as the loss function, compared to the L_2 function that was used in the earlier sections.

3.3.6 Maximum-likelihood Estimation (MLE)

Maximum-likelihood estimation (MLE) is a method of estimating the parameters, Θ, of a statistical model. Given a data set and a model, for example a data set $\mathcal{D}$ and a model $\mathcal{M}$, then assume the data set is normally distributed, but the the mean, μ, and variance, σ, is unknown. Maximum likelihood estimation will then give an estimation of μ and σ, which make the observation on the data set $\mathcal{D}$ the most probable, given the model $\mathcal{M}$. In this case $\Theta = [\mu, \sigma]$ are the parameters of the model.

In general, for a fixed data set and an underlying model, the method of maximum likelihood is based on selections of the parameters that maximizes the **likelihood function**. The likelihood function is given by

$$\mathcal{L}(\Theta|\mathcal{D}) = P(\mathcal{D}|\Theta), \tag{27}$$

where Θ is the set of parameter values of the model and $\mathcal{D}$ is the observed values of the data set. The likelihood gives the probability of the observed values given those parameter values.

Often in practice the logarithm of the likelihood function is used instead, the function is then called the **log-likelihood**. The logarithm is a strictly monotonically increasing function, which mean that the logarithm of a function has the same maximum as the original function and thereby can the log-likelihood be maximized instead of the likelihood. It is often more convenient to work with the log-likelihood function, because the likelihood function often consists of a product, the log-likelihood will then be a sum, which is easier to derivative.

In this project a loss function will be minimized and not maximized as the (log-)likelihood function. The **negative log-likelihood** will then be used instead, which is the negated log-likelihood function and is then given by

$$\text{Negative log-likelihood}(\Theta|\mathcal{D}) = -log(\mathcal{L}(\Theta|\mathcal{D})). \tag{28}$$

3.4 Artificial Neural Networks

Artificial neural networks have been inspired by findings in neuroscience, especially from the hypothesis that mental activity primarily consist of electrochemical signals in a network, called **neurons**.

The neurons in the biological network are "fired" when a linear combination of its inputs exceeds a certain hard or soft threshold.

The neurons are connected by **synapses** and the synapses are the ones transporting the electrochemical signals. In the artificial neural network the synapses can be considered as weights and the neurons as activation functions.

An interesting thing about this, is that this structure can be implemented with help of the theory described in the Sections 3.3.3 and 3.3.4. The neural network consist of a collection of nodes that are connected together into a network and the properties of the network are determined by its topology and the properties of each node in the network.

3.4.1 The Structure of The Neural Network

The neural network consist of **nodes** or **units** (like neurons), connected by directed **links** (like synapses). A link from node i to node j serves to propagate the **activation** a_i from i to j. There is also a **weight**, $w_{i,j}$, associated with each link. The weight determines the strength and sign of the connection. Each node j computes a weighted sum of its inputs by

$$input_j = b_j + \sum_{i \in I} w_{i,j} a_i, \tag{29}$$

where I contains all the links directed into node j. Inside the node is an **activation function**, g, applied and the output from the node is then given by

$$a_j = g(input_j) = g(b_j + \sum_{i \in I} w_{i,j} a_i). \tag{30}$$

Either a non-linear function or a hard threshold can be used as an activation function. An important property when a non-linear function is used, is that the entire network is represented as a non-linear function. Actually any non-linear function can be used as an activation function, but usually it is the **sigmoid** or the **hyperbolic tangent** used.

The nodes inside a network can be connected in two fundamentally different ways.

- **Feed-forward networks**, have connections only in one direction and they form a directed, acyclic graph. Each node receive its input from *upstream* nodes and delivers output to *downstream* nodes, and there are no loops. A feed-forward network is a function of its current input, and there is no internal state other than its weights. [48]
- **Recurrent networks**, feeds its output back into its own input. The network form a dynamic system that may reach a stable state, oscillate

or exhibit chaotic behavior. This form of network can support short-term memory. This has made them more interesting as models of the brain, but the behavior of the network is also more difficult to understand. [48]

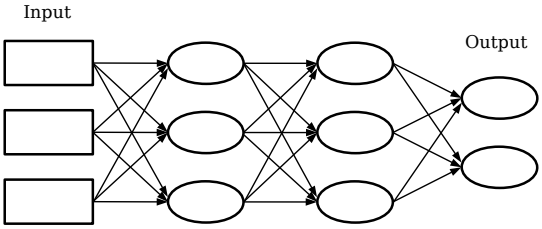


Figure 5: Schematic picture over an example of a feed-forward neural network. The oval nodes will symbolize the parametrisized nodes, compared to the input nodes that are not parametrisized.

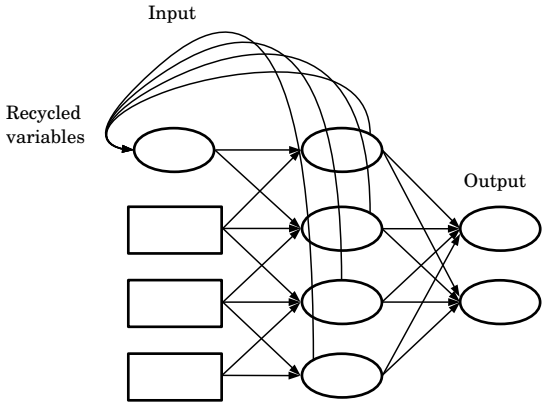


Figure 6: Schematic picture over an example of a recurrent neural network. The oval nodes will symbolize the parametrisized nodes, compared to the input nodes that are not parametrisized.

In this project a feed-forward neural network has been developed. Feed-forward networks are usually arranged in **layers**, where each node receives its input from the nodes in the neighboring preceding layer.

As descried previously in this paper the output from the classifier has been a scalar variable. But to generalize this to fit other neural networks the output is represented as a vector $\hat{\mathbf{y}}$. This generalization will simplify the further explanation when the classification includes more than two classes.

In the rest of this report only feed-forward networks will be described, if not something else is written.

3.4.2 Single-layer Neural Networks

In a single-layer network the output is directly connected to the input, see Figure 7. This network can only represent functions that are linearly separable (both the input and the output should be in a space that are linearly separable). For example, the *AND*-function is linearly separable and can then be represented by a single-layer network, but the *XOR*-function is not, so it can not be represented by a single-layer network.

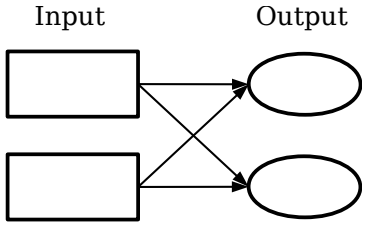


Figure 7: Schematic picture over an example of a small single-layer network. The output is directly connected to the input. The oval nodes will symbolize the parametrisized nodes, compared to the input nodes that are not parametrisized.

3.4.3 Multi-layer Neural Networks

In a multi-layer network there is one or more **hidden nodes**. The hidden nodes are characterized as be between the input and output layer. So all nodes that are not included in the input or output layer is considered hidden, see Figure 8.

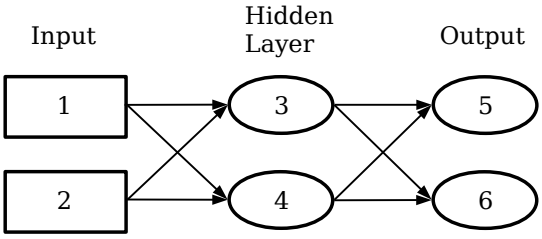


Figure 8: Schematic picture over an example of multi-layer perceptron. It consists of a input layer, a output layer and one hidden layer, with two nodes in each layer. The arrow will symbolize the links between the nodes. Each arrow has a weight, which is not shown in the picture. The weight between node 1 and 4 is for example represented by $w_{1,4}$. The oval nodes will symbolize the parametrisized nodes, compared to the input nodes that are not parametrisized.

The network can be represented as a function, $\mathbf{h}_{\boldsymbol{\Theta}}(\mathbf{x}) = \hat{\mathbf{y}}$, parameterized by its weights, $\mathbf{w}$ and bias, $\mathbf{b}$. In a multi-layer network it is easier to think of the hypothesis as a vector function, $\mathbf{h}_{\boldsymbol{\Theta}}$, rather than a scalar function, $h_{\boldsymbol{\Theta}}$. The output of the hypothesis, will be an expression of its input and its weights. The

gradient decent for loss-minimization can be used to train the network, as long as the expression is differentiable. So, a soft, differentiable threshold, g, has to be used in each node and not a hard, discontinuous threshold.

Lets look at Figure 8, it has two input nodes, two hidden nodes and two output nodes. Let $\mathbf{x} = (x_1, x_2)$ represent the input and the output from each nodes are called **activations** and will be represented by a_i. For the input nodes the activation functions (a_1, a_2) are equal to the input values (x_1, x_2), that is $((a_1, a_2) = (x_1, x_2))$. The expression for output node 5 is then given by

$$\begin{aligned} a_5 &= g(b_5 + w_{3,5}a_3 + w_{4,5}a_4) \\ &= g(b_5 + w_{3,5}g(b_3 + w_{1,3}a_1 + w_{2,3}a_2) + w_{4,5}g(b_4 + w_{1,4}a_1 + w_{2,4}a_2)) \\ &= g(b_5 + w_{3,5}g(b_3 + w_{1,3}x_1 + w_{2,3}x_2) + w_{4,5}g(b_4 + w_{1,4}x_1 + w_{2,4}x_2)), \end{aligned} \tag{31}$$

where g is the non-linear activation function, $w_{i,j}$ the weight of the link between node i and node j, b_j is the offset for node j and a_i the activation for node i. The output for node 6 is given by a similar expression.

As can be seen in Equation 31, the expression for the output has many nestled non-linear functions. A consequence of this is that the entire network will behave as a non-linear function and this will enable **non-linear regression**.

Lets look at the final step in Equation 31 again,

$$a_5 = \mathbf{g}(b_5 + w_{3,5}\mathbf{g}(b_3 + w_{1,3}\mathbf{x_1} + w_{2,3}\mathbf{x_2}) + w_{4,5}\mathbf{g}(b_4 + w_{1,4}\mathbf{x_1} + w_{2,4}\mathbf{x_2})), \tag{32}$$

where the input is written in *green*, the first non-linear transformation is marked with *magenta* and the second non-linear transformation is marked with *blue*. The chain of transformations from input to output is called a **credit assignment path** (**CAP**), and in this example is $CAP = 2$, due to the two non-linear transformations of the input. The credit assignment path is a measure of the depth of the network, and describes potentially causal connections between inputs and outputs. In general for a feed-forward network the credit assignment path is equal to the number of hidden layers plus one, because the output layer also is parametrized. For a recurrent neural network, where the input may propagate through a layer more than once, the credit assignment path is potentially unlimited. In the Figures, in this report, the parametrisized nodes are drawn with ovals compared to the input nodes, which are not parametrisized and drawn with a rectangles.

It has been shown that, with a single sufficiently large hidden layer, it is possible to represent any continuous function of its input, with arbitrary accuracy. And with two layers, even discontinuous functions can be represented [48].

To train a multi-layer network is not so different from the classifiers in Section 3.3 and the update rule in Equation 9, can be used even in this case. To be able to use this update rule a method called **back-propagation** is applied, and a motivation of how it works will now be given.

The error at the output nodes can be calculated by the true values, $\mathbf{y}$, given by the training examples, and by the activations from the output nodes (in Figure 8 the activations of the output nodes were given by a_5 and a_6). The error vector is then given by $Err = \mathbf{y} - \mathbf{h_\Theta}$.

It is interesting to see how this errors will be represented when dealing with the loss function. Equation 9, says that the partial of the loss function has to be calculated.

If L_2 is used as the loss function, the partial derivative of the loss will be

$$\frac{\partial}{\partial\theta}Loss(\mathbf{\Theta}) = \frac{\partial}{\partial\theta}|\mathbf{y} - \mathbf{h}_{\mathbf{\Theta}}|^2 = \frac{\partial}{\partial\theta}\sum_{k\in output}(y_k - a_k)^2 = \sum_{k\in output}\frac{\partial}{\partial\theta}(y_k - a_k)^2, \tag{33}$$

where k spans over all nodes in the output layer. In Equation 33 it is shown that the loss can be decomposed into a separate loss for each output node. This loss function is additive across the components of the error vector. In general if the loss function has this property the learning problem can be handled as separate learning problems for each output node, if they in the end are summarized.

At the output nodes the the update rule is identical to Equation 23. But at the hidden nodes the error will be back-propagated from the output nodes. Let the error at each output node, k, be Err_k. It will be convenient to define $\Delta_k = Err_k \cdot g'(input_k)$, where $g'(input_k)$ is defined as in Equation 30. If this is compared to the Equation 22,

$$-2 \cdot \overbrace{\underbrace{(y - h_{\mathbf{\Theta}}(\mathbf{x}))}_{Err_k} \cdot \underbrace{h_{\mathbf{\Theta}}(\mathbf{x})(1 - h_{\mathbf{\Theta}}(\mathbf{x})) \cdot \frac{\partial}{\partial\theta_i}(b + \mathbf{w}^T \cdot \mathbf{x})}_{g'(input_k)}}^{\Delta_k},$$

it can be seen that Δ_k will represent the partial derivative of the loss function. The update rule for the output layer is then given by

$$w_{j,k} \leftarrow w_{j,k} + \alpha \cdot a_j \cdot \Delta_k \tag{34}$$

The idea of back-propagation is then that hidden node j is responsible for a fraction of the Δ_k of the output nodes which it connects to. The Δ_k values are then divided according to the strength of the connection between the hidden node and the output node, and are then back-propagated back to provide the Δ_j values for the hidden layers. The **propagation rule** for the Δ values is given by

$$\Delta_j = g'(input_j)\sum_{k\in out}\theta_{j,k}\Delta_k, \tag{35}$$

where k spans over all nodes which node j connects to. The update rule for the weights between the input layer and hidden layer is then

$$w_{i,j} \leftarrow w_{i,j} + \alpha \cdot a_i \cdot \Delta_j, \tag{36}$$

which now is identical to the one in Equation 34.

The back-propagation process is summarized in Algorithm 3. [4]

[4]For the complete derivation of the back-propagation algorithm, consult reference [48].

Algorithm 3 Back-propagation

$\mathbf{\Delta} \leftarrow$ Compute the Δ values for all the output nodes,
using the observed error.
Start with the output layer, **then:**
repeat
Propagate the Δ values back to previous layer.
Update the weights between the two layers.
until the first hidden layer is reached.

3.4.4 Convolutional Neural Networks

Convolutional neural networks (CNNs) are specialized neural networks for processing data that has a known grid-like topology. Examples can be time-series data (voice recognition or signature verification as in this project), which can be thought of as a one dimensional grid taking samples at regular time intervals, or image data, which can be thought of as a two dimensional grid of pixels.

Convolutional neural networks are distinguished from ordinary neural networks by using a mathematical operation called **convolution**, which is a special kind of linear operation. Convolution is an operation of two functions of real valued arguments.

For example let $x(t)$ be a function which gives a measurement of the x-coordinate of an object as a function of time. Then let $w(a)$ be a weight function, which does a some kind of weighted average over time and a is the age of the measurement. If $w(a)$ is applied on $x(t)$ at every moment of time, that gives a new function $s(t)$, which is given by

$$s(t) = x(t) \cdot w(a) = \int_{a=-\inf}^{\inf} x(a) \cdot w(t-a)da. \tag{37}$$

This is the **definition of convolution** for a continuous one-dimensional signal. When the time is discrete as in numerical applications, convolution is instead given by

$$s(t) = x[t] \cdot w[a] = \sum_{a=-\inf}^{\inf} x[a] \cdot w[t-a]. \tag{38}$$

The convolutional operation is often denoted as

$$s(t) = (x * w)(t), \tag{39}$$

both for the continuous and the discrete case. A neural network that uses convolution in at least one layer is called convolutional neural network (CNN).

In convolutional neural network terminology, the first argument (in the example, $x(t)$) is referred to as the **input** and the second argument (in the example, $w(a)$) as the **kernel**. The output is often called **feature map**. In machine learning applications is the input often a multidimensional array of data and the

kernel often a multidimensional array of learn-able parameters, usually called weights.

Discrete convolution can be implemented with matrix multiplication. For convolution in one dimension this is done with the **toeplitz matrix**, which is given by

$$Toeplitz = \begin{pmatrix} t_0 & t_{-1} & \cdots & t_{-n} \\ t_1 & t_0 & \cdots & t_{-n+1} \\ \vdots & \vdots & \ddots & \vdots \\ t_n & t_{n-1} & \cdots & t_0 \end{pmatrix}, \tag{40}$$

where t_i is a scalar element in the matrix. Two dimensions convolution can be implemented with a **doubly block circulant matrix**, which is given by

$$\text{Doubly block circulant matrix} = \begin{pmatrix} C_0 & C_1 & \cdots & C_n \\ C_n & C_0 & \cdots & C_{n-1} \\ \vdots & \vdots & \ddots & \vdots \\ C_1 & C_2 & \cdots & C_0 \end{pmatrix}, \tag{41}$$

where C_i is a circulant matrix, which is defined by

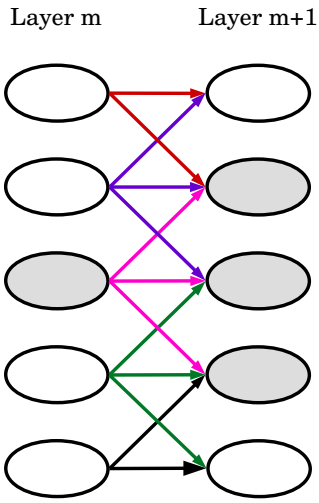
$$\text{Circulant matrix} = \begin{pmatrix} c_0 & c_1 & \cdots & c_n \\ c_n & c_0 & \cdots & c_{n-1} \\ \vdots & \vdots & \ddots & \vdots \\ c_1 & c_2 & \cdots & c_0 \end{pmatrix}, \tag{42}$$

where c_i is a scalar element in the matrix.

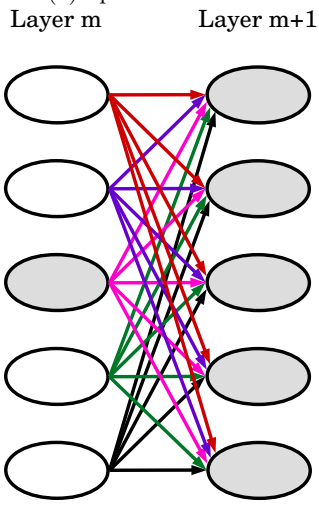
For example to numerically calculate the derivative of a vector $\mathbf{x}$, this can be done by multiply $\mathbf{x}$ with a matrix, A, and then add boundary conditions. The matrix A is a tridiagonal matrix and is given by

$$A = \frac{1}{h^2} tridiag(1, -2, 1) = \frac{1}{h^2} \begin{pmatrix} -2 & 1 & 0 & \cdots & 0 \\ 1 & -2 & 1 & \ddots & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & \ddots & 1 & -2 & 1 \\ 0 & \cdots & 0 & 1 & -2 \end{pmatrix}, \tag{43}$$

where h is some small value. This can be compared with convolution in one dimension, by convolve the one-dimensional input vector, $\mathbf{x}$, with the kernel, $W = \frac{1}{h^2}[1, -2, 1]$, which is of size [1x3]. As seen in Equation 43, the matrix contains many zeros, the matrix is then called **sparse**. A common property of convolution is sparse matrices, because the kernel usually is much smaller than the input. This is called **sparse connectivity** (also referred to as *sparse interactions* or *sparse weights*). Figure 9a shows a sparse connected layer. In a **fully connected** layer every output node is connected to every input node, this could be compared by multiplying the input nodes with a dense matrix (the most elements in the matrix are non-zero). Figure 9b shows a fully connected layer.



(a) Sparse connected.



(b) Fully connected.

Figure 9: In picture (a) and (b) the difference between a sparse connected and a fully connected network is shown. One node in each picture, in layer m, is marked with gray, and the nodes it affect in layer $m+1$, are also marked with gray. The directed links are marked in different color depending on which node it starts from. In picture (a) is a kernel of width 3 used, which lead to that each node in layer m will affect three nodes in layer $m+1$ (except maybe in the boundary of the layer). In picture (b) on the other hand, a node in layer m will affect all of the nodes in layer $m+1$.

A benefit of sparse connectivity is that it needs to store fewer parameters, which reduce the memory requirements of the model and it will require fewer operation to compute its output. It can also improve its statistical efficiency, because it reduce the complexity of the model and the probability to over-train the model is less.

The **receptive field** of a node is the field within stimuli will affect the node. Even if a network has sparse interactions, and the receptive field of a node is small in the direct preceding layer, the node may indirectly interact with a larger part of the input in the earlier layers, as can be seen in Figure 10. This

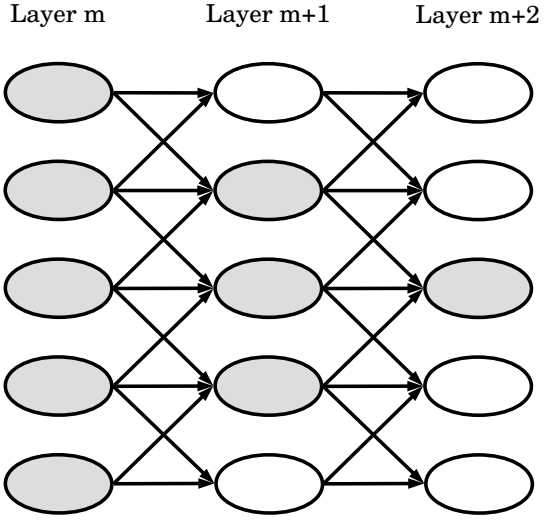


Figure 10: The picture shows the receptive field of a node in layer *m+2*, when a kernel of width 3 is used. The receptive field in deeper layers of a sparse connected neural network, is larger than in more shallow layers. The node in layer *m+2* that is marked with gray, gets information from three nodes in layer *m+1*, and it get information from all the nodes in layer *m*. This shows that even if the direct connections are very sparse, deeper layers can be indirectly connected to most of or all of the nodes in the input layer.

results in that the network will operate on different global levels. In the earlier layers, the convolution will find features on a local level, those feature maps are then input to the deeper layers, which then will find features from those feature maps. The result is that the deeper layers will find features on a more global level on the input data. This allows the network to efficiently describe complicated interactions between many variables.

Convolutional layers also use **parameter sharing** (also known as the network has *tied weights*), which means that the model uses the same parameter for more than one function in the model. In a fully connected layer each element in the weight matrix is used exactly once when computing the output of the layer. Each element in the kernel is used at every element in the input (except perhaps

the boundary elements, depending on the design of the convolution, see Figure 13). Parameter sharing means that instead of learning different set of parameters for every element in the input, its learn just one set. Figure 11 shows this principle. This will also reduce the storage requirements of the model. Another

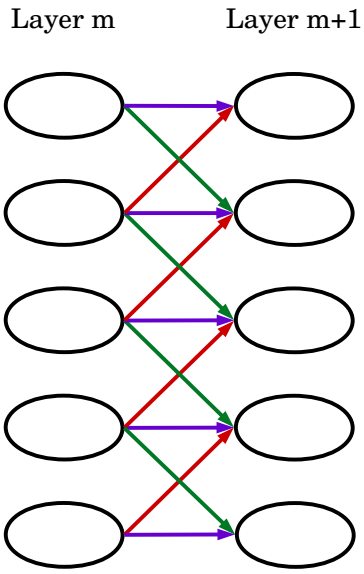


Figure 11: The picture shows the principle of parameter sharing. The links of same color represent weights of the same value. Here a kernel of width 3 is used.

property of convolution is that it is **equivariant to translation**. If the input changes, and then the output changes in the same way, the function is said to be equivariant, mathematically it can be written as $f(g(x)) = g(f(x))$. In the case of processing time series data, convolution produce a sort of time-line that shows when different features appears in the input. So if an event moves later in time, the same representation of it will appear later in time in the output.

Convolution is **not equivariant** to some other transformations, such as changes in the *scale* and *rotation* of the input.

A typical convolutional layer consists of three stages. In the *first stage* it performs **several convolutions in parallel**, this is because each convolution produce one feature map which can detect one type of feature at different locations of the input. If many convolutions are applied in parallel, the model will be able to detect many features at this global level.

In the *second stage*, each feature map is run through a non-linear activation function. This stage is sometimes called **detector stage**.

In the *third stage* a **pooling function** is used, to modify the output of each layer. A pooling function replaces the output of the layer at a certain region by a summery of the nearby outputs. **Max-pooling** takes for example the

maximum value of a rectangular region. [5]

The three stages are shown in Figure 12.

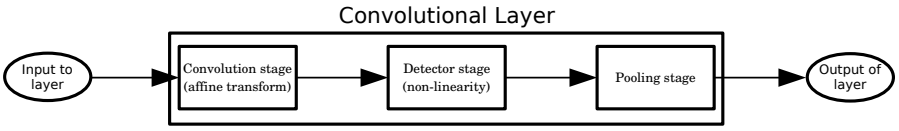


Figure 12: The picture shows the typical components of a convolutional layer.

The pooling function makes the representation become **invariant to small translations** of the input. Invariance to small local translations is useful if it is more important that a feature is present than exactly where it is present.

Pooling over spatial regions produces invariance to translation, but if the pooling is applied over the output of separately parametrized convolutions, the features can learn which transformations to become invariant. However, in this project is a fix pooling factor used.

The pooling function also reduce the size of the layer, which improve computational efficiency of the network, because the next layer will operate on a smaller input size. It will also reduce memory requirements of storing the parameters.

The pooling function can also be used to handle inputs of varying size. This can be done by varying the size of offset between pooling regions, so the classification layer always receive the same number of summary statistics regardless of the input size. This is not used in this project, but could perhaps be used in future development of the model.

Convolution and pooling can cause under-fitting, when pooling is used on all features. A solution could be to use different pooling for different features.

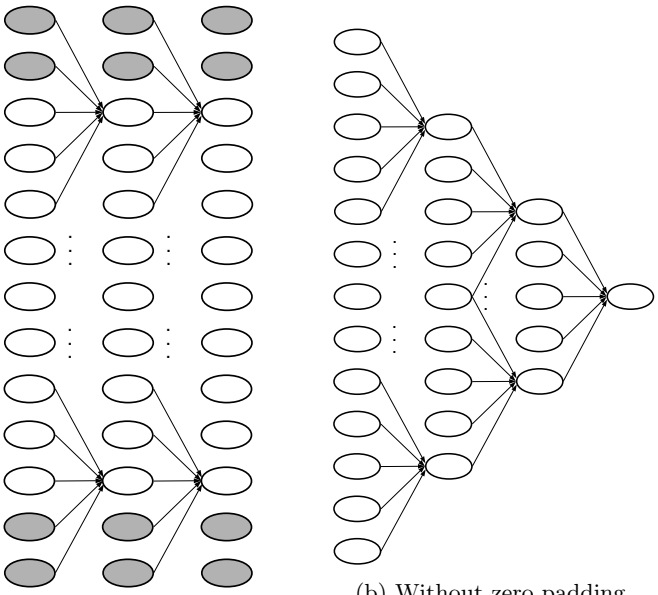
The convolution can be designed in different ways, depending on how the kernel should be used in the boundary of the input data. The alternatives are to add a number of zeros at the boundary of the data, called *zero padding*. The two main methods are:

- **Valid convolution**: no zero padding is used. In this case the width of the representation will shrink by the kernel width minus one in each layer, see Figure 13b. This causes some limitations on how to design the model, because the width of the data shrinks after each convolutional layers and a trade-off between kernel size and number of layers has to be made. This can limit the expressive power of the networks. But a positive aspect is that the output nodes are functions of the same number of input nodes. This makes the behavior more regular, compared to when zero padding is used. [10] This is the design used in this project.

- **Same convolution**: zeros are added at the boundary in order to keep the size of the input equal to the size of the output of the layer, see

[5] A convolutional neural network which uses max-pooling is often denoted MPCNN.

Figure 13a. In this case input nodes close to the boundary will influence fewer output nodes, compared to input nodes in the center. However, the positive aspect is that the model can be design to contain an arbitrary number of convolutional layers, with arbitrary size of the kernel. [10]



(a) With zero padding.

(b) Without zero padding.

Figure 13: In picture (a) and (b) the effect of using zero padding is shown. In this example a kernel of width 5 is used and pooling is not used. In picture (a) four zeros is added in each layer at the boundary (shown in gray), this prevent the layers to shrink in size in deeper layers, compared to in picture (b), where no zero padding is used. In picture (b) each layer will shrink by 4 due to the convolutional operator.

3.4.5 Ensemble Learning

The learning methods described in Sections 3.4.1 - 3.4.3 selected a single hypothesis in a space of hypotheses. The idea of **ensemble learning** is to select a collection, or ensemble, of hypotheses from the hypothesis space. For example if 5 hypotheses are chosen from the hypothesis space, then their predictions can be combined. This can be made by the simple majority voting. So if a new example should be miss-classified, at least 3 of the hypotheses should give this prediction. The ensemble of hypotheses will then form a new hypothesis. Ensemble learning have showed to be very useful and can reduce the error rate further, compared to when a single hypothesis is used.

3.4.6 Neural Networks in Practical Applications

So far in this report the theoretical description about the network and how to train it has been presented. But in a practical application there are more considerations to be made.

The dimension of the input layer (i.e. the number of nods in the first layer) is given by the dimension of the input data. But the dimension of the hidden layers (how many hidden nodes in the hidden layer), and how many hidden layer to use is still not determined. Unfortunately there is no good theory to answer those questions. The standard approach is to use cross-validation and try different structures. A certain amount of work is needed to get the structure right and also to achieve convergence to something close to the global optimum in the weight space.

Another decision concerning the practical application is to choose appropriate input data. The collected input data set is often large, multi-dimensional and may consist of redundant data, and it is an engineering problem to decide what data is necessary for the task at hand.

Another thing to keep in mind is that the size of the collected data set will also affect the performance of the algorithm. In general when it comes to neural networks, a small data set will not provide the sufficient amount of information required for a accurate result.

4 The MNIST Classification Problem

The MNIST digit classification problem is one of the most famous benchmarks in machine learning. This problem is of a certain importance for this project, because tutorials for the MNIST digit classification problem were a starting point in the development of the models for the signature verification problem treated in this project.

4.1 The MNIST Dataset

The MNIST dataset consists of handwritten digits of the numbers zero to nine. All images have been normalized and centered to a fixed size image of 28 x 28 pixels. In the original dataset each pixel of the image is represented by a value between 0 and 255, where 0 is black and 255 is white and anything between is different shades of grey. Some examples can be seen in Figure 14.

Figure 14: Examples of images from the MNIST dataset of handwritten digits.

The set consists of 70 000 examples, and are divided so that 50 000 are in the training set and 10 000 each are in the test and validation set.

The MNIST digit classification problem is a multi-class classification problem, because it is about classify pictures into 10 classes, depending on which

number the picture represents. This can be compared with the signature verification problem in this project, which is a binary classification problem with two output classes (match and non-match).

5 Neural Networks with Deep Architecture

The word **deep learning** refers a to multi-layer network, where many hidden layers are used. Common for the deep learning models are that they attempt to model high-level abstraction in the input data by using a deep structure of composed, non-linear transformations.

What distinguish shallow learning from deep learning is their credit assignment path (CAP) [50]. Most researchers in the field agrees on that deep learning has multiple non-linear layers with CAP > 2. Schmidhuber [50], considers CAP > 10 to be very deep learning.

Deep learning has become a part of many state-of-the-art systems in different disciplines, particular in computer vision and automatic speech recognition [50]. Currently, it has been shown that deep learning architectures in the form of convolutional neural networks (CNNs) have been nearly best performing [36], [54].

5.1 Hardware and Deep Learning Networks

Deep learning systems often requires a lot of computing power, memory and time for training. Often it is needed to pass a certain amount of those ingredients, to achieve good results. In today's society computing power and computer memory are relatively cheap, but this was not the case ten years ago. This may be one of the reasons to why the field is so new.

In the 2000s deep learning had a breakthrough in form of cheap, multiprocessor graphics card (graphic processing units, GPUs). GPUs excel at the fast matrix and vector multiplications, which can speed up neural network training by a factor of 50 and more. This is one thing that has contributed to the success in contests for pattern recognition, image segmentation and object detection [50]. The final model developed in this project used a convolutional neural network, trained on GPU.

5.2 The History of Deep Neural Networks

Shallow neural network-like methods have been around for many decades, if not centuries. However, the early neural network architectures, 1943 [43], did not learn. The first ideas about unsupervised learning were published a few years later, 1949. But in a sense neural network ideas have been around even longer, because early neural networks were essentially variants of linear regression methods going back at least to the early 1800s [28], [29], [41]. Those early neural networks had a CAP depth of 1.

The inspiration of deep neural networks came in the 1960s when simple cells and complex cells were found in the cat's visual cortex [30], [31]. These cells fire in response to certain properties of visual sensory inputs, such as the orientation of edges (compere with the theory described in Section 3.4.1). This inspired the later development of deep neural network architectures.

In 1979, Professor Fukushima proposed the Neocognitron model, [26], [25], [27], which introduced the convolutional neural networks, see Section 3.4.4, which were perhaps the first deep network. The Neocognitron model is very similar to today's contest-winning methods. However, Fukushima did not set the weights by supervised back-propagation, but instead by an unsupervised learning method, and he used spatial averaging instead of max-pooling for the sub-sampling.

Back-propagation was developed in the 1960s and 1970s, and applied to neural networks in 1981. But back-propagation-based training of deep neural networks with many layers, was found to be difficult in practice in the late 1980s, so it become an explicit research subject only, in the early 1990s [50].

In 1989 back-propagation was applied to the Neocognition-like model [38], [37], [39]. The same year the MINST data set of handwritten digits was introduced [38], see Section 4.1. Convolutional neural networks with a depth of 5 achieved good performance on MNIST [37]. They also performed well on fingerprint recognition [9]. Similar networks were also used commercially in the 1990s.

In 1992 came a method called Dresception model [57], which uses max-pooling as down-sampling between the convolutional layers.

But it was not until 2006 the expression *Deep Learning* was coined, even if learning networks with many non-linear layers date back to 1965.

5.3 Effects and Results from Contests-winning Methods

Already in the 1990s, certain neural networks had won some controlled pattern recognition contests with secret test sets [50] (secret test sets were explained in Section 1.2.3).

In the decade around 2000, many commercial pattern recognition applications were dominated by non-neural machine learning methods such as Support Vector Machines (SVMs) [56].

Important for many present competition-winning pattern recognizers were developments in the convolutional neural network department. In 2003, a back-propagation-trained convolutional neural network set a new MNIST record of 0,4 % error rate [51]. It used a method called training pattern deformations, but no unsupervised pre-training, compared to many earlier methods. A standard back-propagation network achieved 0,7 % error rate. Those methods had a low CAP.

Then in 2006 a back-propagated convolutional neural network set a new MNIST record again, of 0.39 % error rate [46]. It used pattern deformation, but no unsupervised pre-training, as before. In 2006 came also the first GPU-based, convolutional neural network implementation [13], which was up to 4

times faster than the CPU-based implementation.

In 2007 back-propagation was first applied to a max-pooling, convolutional neural network and the network was topped with a fully connected layer (explained in Section 3.4.4), this is the structure in the final method developed in this project.

Then in 2009 came a new MNIST record of 0.35 % error rate [18]. It used back-propagation and pattern deformation. It did not use unsupervised learning or convolution. But it used GPU which made it up to 50 times faster than standard CPU-versions. This result seemed to suggest that advances in exploiting modern computing hardware were more important than advances in algorithms [50].

In 2011 came a **GPU based, ensemble, max-pooling, convolutional neural network**, see Section 3.4.5. This kind of system was the first system to achieve superhuman visual pattern recognition [14], [15]. The competition was about traffic sign recognition [52], [53], with images of size 48x48 pixels. This is of interest for fully autonomous, self-driving cars in traffic [21]. The GPU based, ensemble, max-pooling, convolutional neural network obtained 0.54 % error rate, which was twice as good as the human test subjects.

Computers are in general not good at visual pattern recognition. For example in 1997 the human chess world champion was beaten by an IBM computer, but at that time computers could not compete with little kids in visual pattern recognition. So it was a big step when a deep neural network was better than human at recognizing traffic signs in 2011.

In 2012, a GPU based, ensemble, max-pooling, convolutional neural network was developed that also performed human-competitive on MNIST with around 0.2 % error rate [16].

In the same year, another GPU based, ensemble, max-pooling, convolutional neural network emerged that achieved the best result on the ImageNet classification benchmark [35], which is very popular in computer vision, and has larger images with a size of 256x256 pixels.

Around 2011/2012 deep learning achieved excellent result in image recognition and classification. Another area that is of interest is object detection, for application as image-based search engines, or for biological diagnosis, where the goal can be to automatically detect tumors etc in images of human tissue [50].

In 2012 came the first deep learning system to win a contest on visual object detection [17]. It used a GPU based, ensemble, max-pooling, convolutional neural network. The contest was on large images with several million pixels [32], [47].

5.4 Today's Successful Techniques

Most of today's contest-winning or benchmark record-setting deep learning system uses one of two supervised techniques: recurrent long short-term memory (LSTM) trained by connection temporal classification (CTC) (not discussed further in this report), or **feed-forward, GPU based, max-pooling, convolutional neural network** (as the final model in this project) [50].

6 Hardware and Software Used in this Project

This section describes the hardware and software used in this project.

6.1 Software

- The entire project is mainly developed in **Python 2.7** and for the **Linux** platform.
- The open source Python library **Theano** [55] has been used. Theano has support for GPU-based computation and has libraries for mathematical functions. Theano has also a ready-to-use implementation of a function for convolution, *theano.tensor.signal.conv2d()*, which is implemented in C, and is optimized for fast evaluation.
- A tool chain called **CUDA** was used. Theano has support for GPU based computation, and can be configured to use CUDA. To be able to do this is a NVIDA graphic card needed (at the moment).
- The open source Python library **numpy** was used.
- In the development of the algorithm in this project, tutorials from **deeplearning.net** [1] has been a starting point, which deals with the MNIST classification problem.
- To be able to read the hid-events from the pen, the open-source program, **hid-recorder.c** was used [6].
- The entire project was stored and version controlled in **Git**.
- The pictures in this report were made with the open-source tools **Libre Office Draw** and **Inkscape**. The report were written in **LaTeX**.

6.2 Hardware

- The pen used in this project is **Anoto's ED prototype-pen**. This pen enables Bluetooth connection to a computer. The pen need special **Anoto-paper**.

The **computers** that were used are,

- **Byron**: this is *Beatrice Drott's* (one of the author's) computer. During the development of the logistic regression model, see Section 7.5, and of the multi-layer perceptron model, see Section 7.6, was partly of the testing done on this computer.

  The models were run during day time, thought ssh and with help of the program **screen**, the terminals could be accessed both from the office and from home.

During development of those models, mainly the multi-layer perceptron model, see Section 7.6, there were many hyper-parameters that could be varied. *Byron* have four cores, but can simulate eight, which means that up to eight models could be run in parallel.

The computer specification is,

- Graphic card: NVIDIA GeForce GTX 760, with 2 GB RAM.
- Size of RAM: 16 GB.
- Processor: Intel Core i7 4770K.

- **Spaceship**: this is *Thomas Hassan-Reza's* (one of the author's) computer, that he brought to work to be able to train the final model. The final model, the max-pooling, convolutional neural network, was trained during some weeks on this computer.

  Thomas' computer also has multi-cores, but due to the model was run on the GPU, only one model could be trained at a time.

  The computer specification is,

  - Graphic card: NVIDIA GeForce GTX 760, with 2 GB RAM.
  - Size of RAM: 8 GB.
  - Processor: Intel Core i5 3570K.

- **Laptop for collecting signatures**: It was practical to have a movable computer to this task. In this way could the signature collectors get to the signature donors, rather than the other way around. The authors of this project urged the employees at Anoto to come and visit their office room, but very few did. However, it turned out to be much more successful to take the signature-collecting-tools to events, like meetings and coffee breaks, especially events were good company and a piece of cake were included.

A problem that occurred when the models were run on many different computers were that the format of the weights were different, for example *float32* and *float64*. And also that Theano had some special format on their weights. This was solved with development of a conversion program.

6.3 Data

- The **digital signature** is the recorded signature with the Anoto pen. The signatures were written on paper with an ink tip, and the recorded information was sent to a computer via Bluetooth, there it then was saved. The pen has a sample rate of 75 Hz and collects the x- and y-coordinates, pressure, x- and y-tilt and twist.

7 Methods

In the following sections the entire practical part of the project is described. A series of **classification algorithms** have been developed, where the final one is built on a GPU, max-pooling, convolutional neural network. No matter how complex the classification algorithm is, the four steps in Figure 15 are needed.

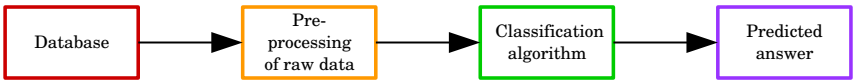


Figure 15: Schematic picture over the code for a signature verification process.

A code frame work for those four steps was developed, and then four different classification algorithms were developed.

First a simple algorithm was developed, which just compared the coordinates of the two signatures, see Section 7.4. This model needed no training since it only used one engineered feature. Then the first model that used machine learning was developed, which was built on logistic regression, see Section 7.5. The next model, the multi-perception model was an extension of the logistic regression model, see Section 7.6. In this model a fully-connected, hidden layer was added before the last, classifying, logistic regression layer. Both the logistic regression and the multi-perception model needed training, but not for so long time as the final model needed. The final model was then based on a convolutional neural network, see Section 7.7, it was also an extension of the earlier multi-perception model. In this model convolutional layers were added before the final two layers, which were a fully connected layer and lastly a classifying, logistic regression layer, as in the multi-perception model. A schematic figure of the last three algorithms, can be seen in Figure 16.

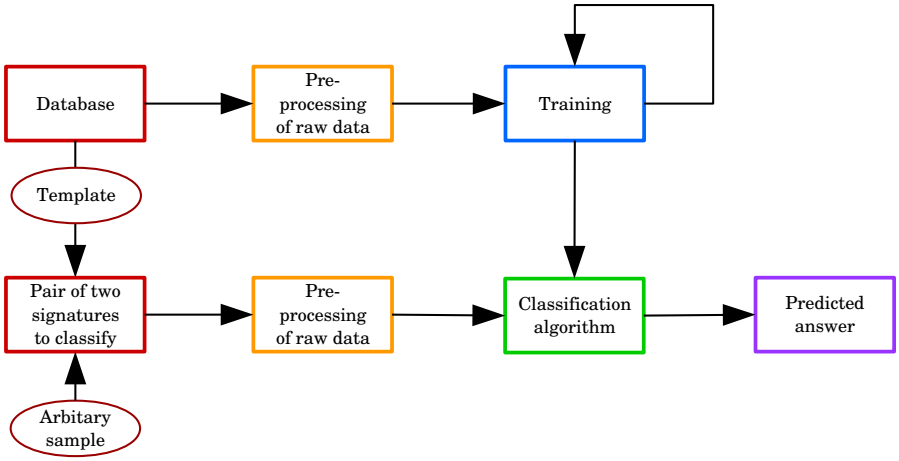


Figure 16: Schematic picture over the code for a signature verification process, where machine learning is used to train the classification algorithm.

7.1 Data Collection

This section will describe the enrollment phase, see Section 2.1, used in this project.

The data collection was done in a standardized way so all signatures were collected under the same conditions. Signatures were gathered from one person at time using equipment from Anoto. The person was given a standard A4-paper to sign 7 authentic samples of their own signature, an example can be seen in Figure 17. The box, in which the signature was written, had the size of

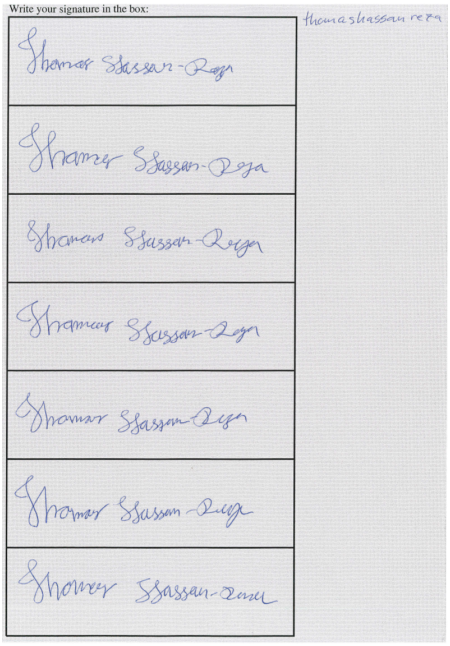


Figure 17: An image of the A4-paper the signature donors were asked to sign. In this case *Thomas Hassan-Reza* has given 7 samples of his signature.

130×40 mm.

7.1.1 Quality Measure

Before saving the biometric data a quality measure of the data is needed, see Section 2.1. The quality check that was made in this project is described in the following section.

After each signature was written, the recorded coordinates were plotted and shown both to the writer and the data collector. In Figure 18 an example of a plotted signature is shown.

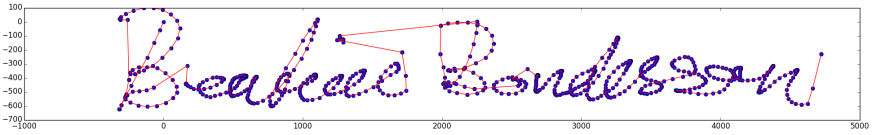


Figure 18: The coordinates, over the recorded signature, were shown in a plot during the data collecting process. The plot was controlled both by the writer, in this case *Beatrice Bondesson* and by the data collector. The blue dots are the recorded coordinates and the red line combines them in the order they were written. Note that from the red line some information about the order of the strokes are given. In this case it can be seen that the dot over the *i* in *Beatrice* is written after the first name is written and not at the end, after the surname.

The quality check that was done on each signature was then:

- The data collector checked that the coordinates were recorded correctly (sometimes some problem occurred and the signature was not recorded correctly).
- Then the writer was asked if he/she was satisfied with the signature, that is, if the writer considered the sample to be representative of his/her signature.

The raw data from the signatures were then saved in a file structure, in a database.

7.2 Pre-processing of Data

To be able to use the saved data from the signatures, some pre-processing was required. The signatures were of different length, and thereby consisted of different number of points. So in order to compare two signatures with each other the *first approach* was to add zeros on the shorter signature until they were the same length. This approach was abandoned when it was discovered that the models needed a standardized signature length to handle the input data. One way of continuing the adding zeros approach would be to find the longest signature in the database and set it's size as the standard length. But this would result in that all signatures, except the longest one, would receive redundant data to fit a model. So the approach of re-sampling the data to a specific length was proposed. This was a trade-off between losing information of the sample and adding redundant data to the sample.

7.2.1 Re-sampling of the Input Data

The first decision that had to be made was the **standard length** of a signature. This was chosen to be 300 points because that was the average of all of the signatures collected at that time. Now all the signatures should be re-sampled

to that standard length. How this re-sampling should be made was not entirely clear from the start.

Some different approaches were tried but it ended up being **linear interpolation** that was implemented. Linear interpolation is when a straight line is drawn between two known points $(x1, y1)$ and $(x2, y2)$, see Figure 19.

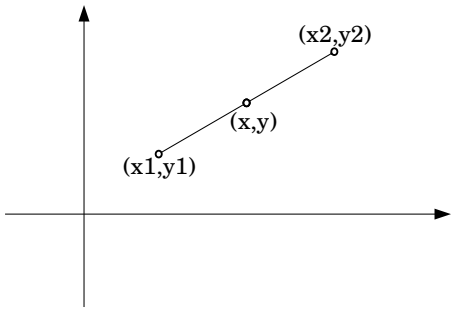


Figure 19: In this figure two points $(x1, y1)$ and $(x2, y2)$ are connected with a straight line and a new point is added (x, y) between them.

This means that all points which are added on a straight line between the two closest original sample points. In order to clarify, all the original points build up a series of connected straight lines which are described by Equation 44.

$$y = y_1 + (y_2 - y_1)\frac{x - x_1}{x_2 - x_1}, \tag{44}$$

where all variables are described in Figure 19.

Then a new one dimensional grid is calculated from the original number of sample points and the standard length of a the signature. With this new one dimensional grid, $\mathbf{x}$, it is easy with the help of Equation 44 to calculate the new values, $\mathbf{y}$. The implemented algorithm which is responsible for the re-sampling task was developed so it was easy to change the standard length of a signature.

When it comes to signatures which are shorter than the standard length the added points are considered to be redundant data as well as when adding zeros. But this was the way signatures of different length were handled in this project.

7.2.2 Normalization of the Input Data

The data was also normalized in a way, so the first point was sat to be the origin for the signature. The normalization was only done on the x- and y-coordinates. The other parameters (pressure and angles) were just keep to be the same. This results in that all signature are invariant to where the first contact of the paper occurred (for the x- and y-parameters). The angles and pressure whoever are measured in absolute numbers and not to their relative origin.

7.3 Train, Test and Validation Sets

The total data set was divided into three separate sets, the training set with a part of $\frac{5}{7}$ of the total data set and test and validation set with a part of $\frac{1}{7}$ each. These three sets were strictly separated at sample-level, but not at person level. That is, the same parson could have given signatures in all three sets, but the signatures in each set were different samples. For example if a person had given 7 signatures, then 5 were in the training set, 1 in the test set and 1 in the validation set. The samples in the total set were randomized and then divided into the three sets, so the distribution were not strictly as described above, but the principle was like that.

The signatures were then paired, with two signatures in a pair. The number of pairs of signatures that were written by the same persons, true examples, was much less then the number of pairs written by different persons, false examples. The sets were then balanced in each training round (Algorithm 5, explains training round) in that way that each set consisted of $\frac{1}{2}$ of true examples and $\frac{1}{2}$ of false examples. The false examples were chosen randomly from the sets total set of non-equal pairs and the new balanced set were then shuffled before a new training round began. Figure 20 shows this process.

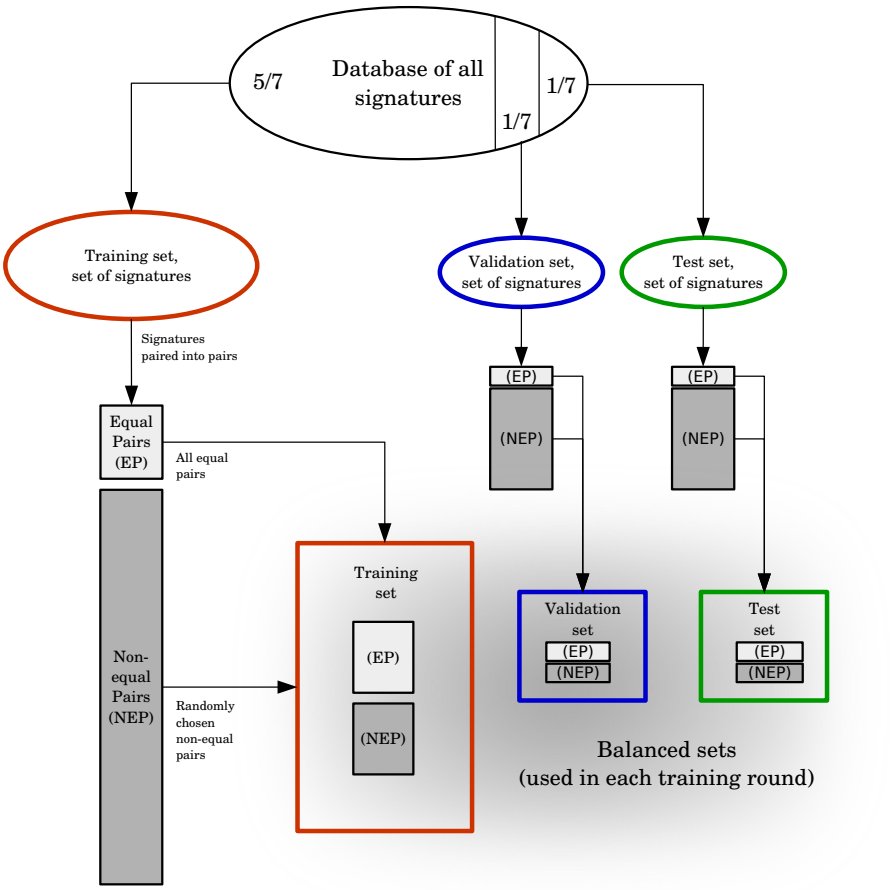


Figure 20: This schematic picture shows how the training, test and validation sets were selected. The total database of signatures was divided into three sets, were $\frac{5}{7}$ of the signatures was in the training set and a part of $\frac{1}{7}$ was in each of the test and validation set. After the division was made, the signatures were paired into pairs, where the number of equal pairs (match) were much less than the number of non-equal pairs (non-match). Therefor, a part of the non-equal pairs was chosen randomly from each set, in each training round, so the number of non-equal pairs were equal to the number of equal pairs. In this way balanced sets were used in each training round. The reason why the non-equal pairs were chosen randomly within the set was that in this way was the original, total set maximally utilized. Before the training in each round started, were all the pairs in the set shuffled, in order to variate the shape of the cost function.

7.4 First Attempt of an Easy Compare Algorithm

The first attempt to compare two signature was made by the Easy Compare algorithm. This was done by just simply engineer a feature called *diff*,

$$diff = \sum_{i=x,y} \sum_{j}^{N} |SignatureA_{i,j} - SignatureB_{i,j}|, \tag{45}$$

where *SignatureA* and *SignatureB* are the data collected from signature A and B (two signatures in a pair). The index i is at first the parameter x then y and the index j is the specified value for that parameter at sample point j.

To summarize, the feature *diff* is the sum of absolute differences between each sample point from two signatures over the x- and y-coordinates.

The reason why the absolute value difference was chosen is that it is easy to generalize if more parameters (for example the pressure) should be added to the difference (i = x,y,p). This engineered feature was the first naive approach in solving the signature verification problem.

This was also a quick way of developing a code framework of collecting signature data, processing the data and evaluating results. The results from this easy method is also used as a benchmark for the more advanced methods. The interesting part of this model is that it is easy to construct and understand. The question that is asked is if increased complexity in more advanced algorithms will result in improved performance and will be answered later in this report.

7.5 Second Attempt with Logistic Regression

The second attempt was made with logistic regression, which is a linear classifier, see Section 3.3.4. This was the first machine learning algorithm, that was developed.

7.5.1 Implementation

The development of this model started from a tutorial [4], which is about classifying the MNIST database, see Section 4.1. The MNIST digit classification problem is a multi-class classification problem, so the tutorial used the theory described in Section 3.3.5.

The linear classifier is parametrized by a weight matrix $\mathbf{W}$ and a bias vector $\mathbf{b}$, as in Section 3.3.5. Recall Equation 25 in Section 3.3.5, the probability an input matrix $\mathbf{x}$ is a member of class i is given by,

$$\begin{aligned} P(y = j|\mathbf{x}, \mathbf{W}, \mathbf{b}) &= softmax(\mathbf{b} + \mathbf{W} \cdot \mathbf{x}) \\ &= \frac{e^{b_i + W_i \cdot \mathbf{x}}}{\sum_{k=1}^{K} e^{b_k + W_k \cdot \mathbf{x}}}, \end{aligned} \tag{46}$$

The prediction is then given by (recall Equation 26),

$$\hat{y} = argmax_i P(y = i|\mathbf{x}, \mathbf{W}, \mathbf{b}). \tag{47}$$

For the MNIST problem there are 10 classes, but in the signature verification problem there are just 2, which means that the signature verification problem could be implemented with the theory described in Section 3.3.4 and the logistic (sigmoid) function for binary classification problems. But since the MINST tutorial was followed, the classification was implemented with the *softmax* function instead, but using two output classes.

The dimension of $\mathbf{W}$ is $[\mathbf{W}] = [16 \cdot 300, 2]$, where $16 \cdot 300$ is the number of data points in one training example, 8 parameters [6] from each signature in the pair, each parameter was re-sampled to 300 points, see Section 7.2.1. The 2 in the shape of $\mathbf{W}$, stands for the two output classes. The dimension of $\mathbf{b}$ is $[\mathbf{b}] = [2, 1]$, and are the free parameters for the two classes. The weights $\mathbf{W}$ and $\mathbf{b}$ were initialized with zeros, before the training started and were then changed during training.

The loss function that was used was the negative log-likelihood function, as described in Section 3.3.5. The cost function was sat to be equal to the loss function, that is, no regularization (Section 3.3.2) was used in this model. The loss/cost function was then minimized by mini-batch stochastic gradient decent (MSGD) (Section 3.3.1).

The loss function is given by

$$\begin{aligned} Loss(\mathbf{W}, \mathbf{b}, \mathcal{D}) &= -\frac{1}{|\mathcal{D}|}\mathcal{L}(\Theta = \{\mathbf{W}, \mathbf{b}\}, \mathcal{D}) \\ &= -\frac{1}{|\mathcal{D}|}\sum_{i=0}^{|\mathcal{D}|}\log(P(Y = y^{(i)}|x^{(i)}, \mathbf{W}, \mathbf{b})), \end{aligned} \tag{48}$$

where $\mathcal{D} = (\mathbf{x}, \mathbf{y})$ are the training examples in the mini-batch, and $x^{(i)}$ is the ith training example in the mini-batch, and $y^{(i)}$ is the key corresponding to this example, $\Theta = \{\mathbf{W}, \mathbf{b}\}$ is the parametrization of the model and $\mathcal{L}$ is the log-likelihood function. [7]

The model was then trained according to the theory described in Section 3.3, and the weights were updated according to Algorithm 2 and the update rule in Equation 9.

The gradients of the loss function in Equation 9, $\frac{\partial Loss}{\partial \mathbf{W}}$ and $\frac{\partial Loss}{\partial \mathbf{b}}$, can be fairly complex, especially when taking problems of numerical stability into account. But with the Python library Theano, this is quite simple, because Theano performs automatic differentiation and applies certain maths transformations to improve numerical stability.

The entire data set was divided into tree sets, for training, validation and testing, see Section 7.3 and Figure 20.

[6] In the other sections of this report only 6 parameters are mentioned exclusive time. In this phase of the development, another parameter also was used, *pen up/down*. This was later removed, due to the pressure also gave the same information.

[7] Formally the negative log-likelihood is defined as the sum, see Section 3.3.5, and not the mean of the data set. But in practice, to take the mean instead of the sum, will result in that the choice of the learning rate is less dependent of the size of the mini-batch.

The entire learning algorithm was made by looping over all batches in the training set, compute the minibatch cost, taking one step of MSGD. The training was validated with the validation set. If the improvement of the validation error rate was smaller then a certain threshold (called patience) then the training was interrupted.

Algorithm 4 Learning algorithm for logistic regression

```
function LEARNING(epoch_max, minibatches_train_set)
    W ← 0 or earlier saved W
    b ← 0 or earlier saved b
    Θ ← [W, b]
    Θ = TRAINING_ROUND(epoch_max, minibatches_train_set, Θ)
    return Θ
end function
```

Algorithm 5 One training round

```
function TRAINING_ROUND(epoch_max, minibatches_train_set, Θ)
    done ← false
    loop epoch < epoch_max and not done
        epoch ← epoch + 1
        for each minibatch ∈ minibatches_train_set do
            Calculate the cost for this minibatch
            for each θ ∈ Θ do
                θ ← θ − α ∂/∂θ cost(θ)
            end for
            if time to check validation set then
                Calculate error rate over validation set
                if performance improvement < patience then
                    done ← true
                end if
                Calculate error rate over test set,
                // just for evaluation
            end if
        end for
    end loop
    return Θ
end function
```

In Algorithm 4, epoch_max is the maximum number of epochs the algorithm will run (user defined) and minibatches_train_set is a list of all minibatches in the training set. The test set has no algorithmic meaning accept inform about the models performance. In this project **one training round** is defined as when Algorithm 5 returns, i.e. after the loop stops, due to *epoch_max* is reached or if the validation error has not improved enough. Because the cost function for the

logistic regression is convex, the Algorithm 4, converged nicely in one training round. Later in Section 7.6 and 7.7, the algorithms did run for more than one training round.

7.5.2 Problems Encountered on The Way

In this section the problems encountered during the development phase are described and the process that lead to the final choice of this model.

Some problems that were encountered on the way are:

- When the model finally compiled, the error rate was suspectly low, below 3 %. But it was soon realized that this low error rate was due to that the sets were not balanced. It was just a small fraction, some percent of the sets that had true samples. This was fixed, and the sets were then balanced, see Section 7.3.
- Then an error rate of 15% was reached, which was also beyond expectation. But the bug was now that the three sets, the training, test and validation sets, were not strictly separated. That means, that the algorithm was trained on the same samples as it was tested and validated on, which leads to *peeking*, see Section 3.2, and invalidated the model. This was fixed, and the sets were then strictly separated on samples level, see Section 7.3.
- Another bug that was found when developing this model was that the samples were not randomly distributed in the batch, and this affected the shape of the cost function and then the optimization. When the samples were randomized another bug was found, the python's *random.shuffle()* function did not worked as expected on *numpy.array-objects*, as first were used, but the function worked fine on ordinary list-objects. Both those problems were solved.

After the problems were solved the error rate reached around 30%.

7.6 Third Attempt with a Multi-layer Perceptron

The third attempt was made with a small multi-layer model, which is an extended version of the logistic regression model. This model i called Multi-Layer Perceptron (MLP). In this model the input is first transformed by a learned non-linear, fully-connected hidden layer. Fully connected means that the output from the hidden layer is a linear-combination of it's input, see Section 3.4.4. The linear regression model will only succeed on linearly separable input data, but the hidden layer in this model will project the input data into a space where it becomes linearly separable. But compared to the linear regression model, which had a convex cost function, the cost function in this model is not convex. This makes the training harder, because the optimization method can not guarantee to find the global minimum, it may get stuck in local minima on the way.

This model also had many hyper-parameters that could be varied and it is not trivial to find a good choice of those parameters, because there is no optimization procedure for such work. The only option is trial, error and experience, see more about this in Sections 7.6.3 and 7.6.2.

7.6.1 Implementation

The development of this model started from a tutorial [5]. The tutorial is about classifying the MNIST database, see Section 4.1.

The expression for the output from the model is given by

$$\mathbf{f}(\mathbf{x}) = softmax(\mathbf{b_2} + \mathbf{W_2} \cdot h(\mathbf{x})), \tag{49}$$

this is quite close to Equation 25, but here is the input given by $h(\mathbf{x})$ instead of $\mathbf{x}$. $h(\mathbf{x})$ is the hidden layer, and is given by

$$h(\mathbf{x}) = tanh(\mathbf{b_1} + \mathbf{W_1} \cdot \mathbf{x}), \tag{50}$$

where hyperbolic tangent is used as activation function. The dimension of $\mathbf{W_1}$, $\mathbf{b_1}$, $\mathbf{W_2}$ and $\mathbf{b_2}$ are
$[\mathbf{W_1}] = [16 \cdot 300, \text{number of hidden nodes}]$,
$[\mathbf{b_1}] = [\text{number of hidden nodes}, 1]$,
$[\mathbf{W_2}] = [\text{number of hidden nodes}, 2]$ and
$[\mathbf{b_2}] = [2, 1]$.

In Algorithm 6, the weights in the hidden layer are initialized with random

Algorithm 6 Learning algorithm for the multi-layer perceptron model

```
function LEARNING(epoch_max, minibatches_train_set)
    W1 ← rand or earlier saved W1
    b1 ← 0 or earlier saved b1
    W2 ← 0 or earlier saved W2
    b2 ← 0 or earlier saved b2
    Θ ← [W1, b1, W2, b2]
    loop until user stops training
        minibatches_train_set = shuffle examples and replace
                non-matching pairs, see Figure 20.
        Θ = TRAINING_ROUND(epoch_max, minibatches_train_set, Θ)
    end loop
    return Θ
end function
```

numbers from the uniform distribution, on the interval

$$[-\sqrt{\frac{6}{nodes_{in} + nodes_{out}}}, \sqrt{\frac{6}{nodes_{in} + nodes_{out}}}]. \tag{51}$$

This interval is dependent on the activation function, and should be taken in the linear interval of the activation function and Equation 51, is the linear regime of the hyperbolic tangent.

In Algorithm 6, epoch_max is the maximal number of epochs the algorithm will run (user defined) and minibatches_train_set is a list of all minibatches in the training set. Compared to the logistic regression, this model improved if it was run during several training rounds. The reason is that the cost function is not convex for this model, in contrast to the logistic regression model, which means that the function can reach a local minimum. When a new round then starts, the cost function has another shape, because the examples has been shuffled and the non-equal pairs are replaced to other pairs in the set. So if the function was stuck in a local minimum, there is a possibility it is not a local minimum in the new cost function and the training can continue to improve. The training took some days for the multi-layer perceptron model.

In this model a regularization (see Section 3.3.2) was used, compared to the logistic regression model.

7.6.2 Problems Encountered on The Way

In this section the process, which result in the final model and final choice of hyper-parameters for this model will be described. The hyper-parameter optimization part, played a mayor rule in the development of the multi-layer perceptron model.

A model was built, which was an extended version of the logistic regression model, but now with an extra fully-connected layer before the classifying layer. With different options to the program different hyper-parameters and the activation functions could easily be varied. Both the hyperbolic tangent and the sigmoid function were tried.

Some problems that were encountered on the way are:

- The development was first carried out on the computers at the office. First there was a problem with the memory capacity, when the model was tried to run. The program got out of memory when it read a lot of data into the RAM. But this was solved with optimizing the code framework, see Figure 16, and the code was rewritten to used less RAM during run-time. After this it was possible to run the program.

- During the development of the program, the computers at the office were used. But during run-time, the program exceeded the capacity of the CPUs. This result longer run-time, but also that it was not possible to continue the development of the project, at the same time the model was training. This was solved by partly of the training was carried out at the Byron computer, see Section 6.2, and the programs were run through *ssh*.

- Then when the model was tried for the first time, it did not perform very well. The error rate was around 50%. Sometimes the error rate started around 47 %, and increased to 50 %. At this time the learning rate was around 0.1-0.8. With some consultancy from theory [8], the learning rate

[8] If the step size is too large, larger than twice the largest eigenvalue of the second derivative matrix (Hessian) of the cost function, then gradient steps will go upward instead of downward

was then varied in log space, $\{\cdots, 10^{-3}, 10^{-2}, 10^{-1}\}$. It was then found the bad performance was due to the big learning rate. When the model was run with a learning rate of 10^{-7}, improvements were seen. After this an error rate around 30% was reached when hyperbolic tangent was used as activation function.

- The sigmoid function was also tried as the activation function, but without any promising results.
- The number of hidden nodes were also varied. It was a little bit hard to see if it gave good result to increase the number of hidden nodes, because the time for the training increased significantly. With a big number of hidden nodes one training round could take over 13 hours, compared to some seconds for a low number of hidden nodes. But the error rate seems to reach around 30% for all the models, where the hidden nodes were varied.
- If a to small batch size were used, like 5 samples, bad performance was also noticed. With this small batch size, the error rate was increased to 50%. For the other models a batch size of 100 samples were used.

The best result for this model was around 30% error rate.

7.6.3 Model Selection

After all testing described in Section 7.6.2, a good choice of parameters seems to be:

- A batch size of about 100.
- Learning rate around 10^{-6}.
- Hyperbolic tangent as activation function.
- A number of hidden nodes of 1000.

The other parameters that were used but not varied and tested further were:

- Regularization with the L_1-norm, $L_1 = 0.01$.

7.7 Final Model with Convolution and Deep Architecture

This model was an extended version of the multi-layer perceptron model. The two last layers were as in the multi-layer perceptron model, a fully-connected layer and a classifying, logistic regression layer. But before those layers, convolutional layers were added. The convolutional layers had the structure described in Section 3.4.4 and in Figure 12. The training was carried out on the computer *Spaceship*, where the model was run on the GPU rather than on the CPU as in the earlier models. The training also took longer time, about some weeks/month, compared to some days for and the multi-layer perceptron model.

[40].

7.7.1 Implementation

The development of this model started from a tutorial [3], which was based on the LeNet-5 architecture [39] and is designed to classifying the MNIST database, see Section 4.1.

The model is build on the theory presented in Section 3.4.4. Recall the structure of the convolutional layer that was described in Section 3.4.4 and shown in Figure 12.

- **Convolutional stage**: a set of sub kernels, W^k, were applied in parallel, which gives a set of different feature maps.
- **Pooling stage**: max-pooling was used on the rectangular interval [1x2] on the output from the convolutional stage. This reduced the size by a factor 2. The pooling was kept small in order to avoid under-fitting.
- **Detector stage**: each feature map from the pooling stage were then run through the activation function, which was the hyperbolic tangent. [9]

The output from the layer is given by

$$\text{Output of convulotional layer} = tanh(\text{Max-pooling}((\mathbf{W} * \mathbf{x})) + \mathbf{b}), \qquad (52)$$

where *tanh* is the hyperbolic tangent function, which is applied element-wise on its input matrix. The max-pooling function is described in Section 3.4.4 and $\mathbf{x}$ is the input to the layer and $\mathbf{W}$ and $\mathbf{b}$ are the parameters to the layer. The convolution ($\mathbf{W}*\mathbf{x}$) was made by Theano's library function *theano.tensor.signal.conv2d()*, Figure 21 shows this function applied on an image.

In this section $\mathbf{W}$ will also be refereed to as the kernel, and is a 4D matrix. The dimension of $\mathbf{W}$ is $[\mathbf{W}] = [dim1, dim2, dim3, dim4]$, where *dim3* and *dim4* gives the dimension of each *sub kernel*, which will be convolved over each channel, *dim2* gives the channel. The dimension of the vector $\mathbf{b}$ is $[\mathbf{b}] = [dim1, 1]$. In the first layer of the model the channel represents which parameter to be convolved. In image recognition this typically is the color (red, yellow and blue) channels. The dimension, *dim1*, gives the set of sub kernels, recall Section 3.4.4, that in each convolutional layer many convolutions are made in parallel, in order to be able to detect more than one feature at each global level.

The input is also a 4D matrix, where the first index gives the batch index, the second the channel index and the third and fourth gives the dimension of the input data for each channel.

The model consists of three convolutional layers, where each layer has 20, 40, respective 60 different feature maps. The dimension of the input and the kernel, will change between each convolutinal layer, because no zero padding was used, see Section 3.4.4. The dimensions of the input to each layer and the kernel in each layer are given below:

[9] As can be seen here, the order of stages is a little bit different form the general structure, presented in Section 3.4.4. The detector stage is here used as the last step, this is due to the LeNet architecture.

(a) Original picture.

(b) Transformed with kernel of size [15x15] and one feature map.

Figure 21: The figure represents a transformation with the operator *theano.tensor.nnet.conv2d()*. The sub kernel is chosen to be of size [3x3] and only one feature maps are used. The original image is of size [726, 492] and consists of the channels (red, yellow and blue). In those pictures are only the gray-scale of the input image and the output image shown, in order to show the behavior of the convolution better. The weights are initialized by a uniform random distribution, on the linear interval of the hyperbolic tangent. The transformation is given by $f(\mathbf{x}) = tanh(\mathbf{b} + theano.tensor.nnet.conv2d(\mathbf{x}, \mathbf{W}))$, where $\mathbf{x}$ is the picture in (a) and $f(\mathbf{x})$ is the picture in (b). In picture (b), it can be seen that a randomly chosen filter will act as a edge detector.

- **First convolutional layer:** The dimension of the input to the first layer was given by $[\mathbf{x_1}] = [100, 7, 1, 600]$, where the first is due to the batch size, the second is how many channels there are, which is one for each parameter, the third is the first dimension of the input data, which is 1 and the fourth is the second dimension of the input data, which is 600. All signatures were re-sampled to 300 points, see Section 7.2.1. Then the input consists of data from two signatures, which gives 600 points.

  The dimension of the kernel was then $[\mathbf{W_1}] = [20, 7, 1, 10]$, where the first gives how many feature maps used in the layer, the second is how many channels there are, which is one for each parameter, the two last are the shape of the sub kernel, which was convolved over each channel.

  This layer had $20 \cdot 7 \cdot 1 \cdot 10 + 20 = 1\ 420$ parameters.

- **Second convolutional layer:** The dimension of the input to the second layer (and output from the first layer) was $[\mathbf{x_2}] = [100, 20, 1, 295]$. The number of channels is now 20, because 20 feature maps were used in the first layer. The number of data points in each channel is given by $\lfloor (600 - 10 + 1)/2 \rfloor$, where 600 was the number of data points in the first layer and 10 was the width of the sub kernel in the first layer. The division by two is due to max-pooling.

  The dimension of the kernel was then $[\mathbf{W_2}] = [40, 20, 1, 10]$. The first dimension gives the number of feature maps in this layer. The second dimension is the number of channels, which are 20, of the same reason as for the input, $\mathbf{x_2}$. The other dimensions are the same as in the first layer.

This layer had $40 \cdot 20 \cdot 1 \cdot 10 + 40 = 8\ 040$ parameters.

- **Third convolutional layer:** The dimension of the input to the third layer (and output from the second layer) was $[\mathbf{x_3}] = [100, 40, 1, 143]$ and the dimension of the kernel was $[\mathbf{W_3}] = [60, 40, 1, 10]$. Those dimensions are given in a similar way as for the second layer.

  This layer had $60 \cdot 40 \cdot 1 \cdot 10 + 60 = 24\ 060$ parameters.

The output from the last convolutional layer had the shape of $[100, 60, 1, 67]$. The **fourth fully connected layer** operates on a two dimensional input, so before passing the output from the third layer it is reshaped, and the dimension of the input to the fourth layer was then $[\mathbf{x_4}] = [100, 60 \cdot 1 \cdot 67] = [100, 4020]$. The number of hidden nodes was sat to 1000 in this model, which gives an input size of 1000 to the last classifying layer. The size of the weight matrix, W_4, for the fully connected layer was $[\mathbf{W_4}] = [4020, 1000]$ and for the **last classifying layer** $[\mathbf{W_5}] = [1000, 2]$.

The fully connected layer had $4020 \cdot 1000 + 1000 = 4\ 021\ 000$ parameters and the last classifying layer had $1000 \cdot 2 + 2 = 2002$ parameters. As can be seen, the fully connected layer is the heaviest layer, with 99.1 % of all the parameters in the model. The total model had 4 0560 522 parameters.

Before the model described above was made, another design was carried out, which had four convolutional layers, with ten feature maps in each layer and only one channel for all the parameters. This model perform as well as the one earlier in this section. But the number of parameters in the convolutional layers was smaller, only 440 (compared to 33 520 in the other model). The fully-connected layer had 291 000 parameters, and the last classifying layer had as before 2002 parameters. The total number of parameters was then 293 442. Even for this model was the fully connected layer the heaviest layer with 99.1 % of all the parameters.

The training was then done by the procedure shown in Algorithm 7, which is similar to the one for the multi-perception model. The weights are initialized with random values, initialized from the uniform distribution, on the interval shown in Equation 51.

Algorithm 7 Learning algorithm for the convolutional neural network

function LEARNING(epoch_max, minibatches_train_set)
  $\mathbf{W_1} \leftarrow$ **rand** or earlier saved $\mathbf{W_1}$
  $\mathbf{b_1} \leftarrow \mathbf{0}$ or earlier saved $\mathbf{b_1}$
  $\mathbf{W_2} \leftarrow$ **rand** or earlier saved $\mathbf{W_2}$
  $\mathbf{b_2} \leftarrow \mathbf{0}$ or earlier saved $\mathbf{b_2}$
  $\mathbf{W_3} \leftarrow$ **rand** or earlier saved $\mathbf{W_3}$
  $\mathbf{b_3} \leftarrow \mathbf{0}$ or earlier saved $\mathbf{b_3}$
  $\mathbf{W_4} \leftarrow$ **rand** or earlier saved $\mathbf{W_4}$
  $\mathbf{b_4} \leftarrow \mathbf{0}$ or earlier saved $\mathbf{b_4}$
  $\mathbf{W_5} \leftarrow \mathbf{0}$ or earlier saved $\mathbf{W_5}$
  $\mathbf{b_5} \leftarrow \mathbf{0}$ or earlier saved $\mathbf{b_5}$
  $\mathbf{\Theta} \leftarrow [\mathbf{W_1}, \mathbf{b_1}, \mathbf{W_2}, \mathbf{b_2}, \mathbf{W_3}, \mathbf{b_3}, \mathbf{W_4}, \mathbf{b_4}, \mathbf{W_5}, \mathbf{b_5}]$
  loop until user stops training
    minibatches_train_set = shuffle examples and replace
        non-matching pairs, see Figure 20.
    $\mathbf{\Theta}$ = TRAINING_ROUND(epoch_max, minibatches_train_set, $\mathbf{\Theta}$)
  end loop
  return $\mathbf{\Theta}$
end function

7.7.2 Problems Encountered on The Way

The main problem was the long running time, which limit the possibility to try a different number of designs of the network.

7.7.3 Model Selection

Since the size of the input to deeper layers is smaller than the input to earlier layers, fewer feature maps are chosen in the earlier layers, in order to keep the product of the number of features and the number of data points roughly constant across layers. Earlier work on the subject, [40], has shown that this in general is a good strategy. This was the reason why this choice was made.

When choosing number of layers and shape of sub kernels, a trade off was made between running time and performance. But worth mention is that there was not enough time to do any deeper investigation of this, and this values were more or less picked randomly.

Many of the other hyper-parameters were chosen in the same way as for the multi-perceptron model.

8 Results

This section will describe the results form the different models and how the performance was evaluated.

8.1 Performance Evaluation of Different Methods

To be able to compare the different results from each other a standardized evaluation method is required. The evaluation should also be independent on the size of the database especially since the data collection was a continuous process throughout the project. One way of doing this is to look at the percentage of correct predictions.

For the binary classification problem there are two different ways of both doing a correct and an incorrect prediction. This results in four different types of predictions.

Recall from Section 3.2 that the true value, y, is given by the training example and the prediction, $\hat{y}$, is given by the algorithm. In this report a match will be denoted as the number *one*, for being *true* and a non-match as the number *zero*, for being *false*. The four different types of the prediction are then:

- **True Positive** (TP) is when the prediction of the algorithm is a match and the true value also is a match, i.e. $\hat{y} = y = 1$.
- **False Positive** (FP) is when the prediction of the algorithm is a match but the true value is a non-match, i.e. $\hat{y} = 1$ and $y = 0$.
- **True Negative** (TN) is when the prediction of the algorithm is a non-match and the true value also is a non-match, i.e. $\hat{y} = y = 0$.

- **False Negative** (FN) is when the prediction of the algorithm is a non-match but the true value is a match, i.e. $\hat{y} = 0$ and $y = 1$.

These different scenarios are summarized in Table 1.

Type of scenario	$\hat{y}$	y
True Positive (TP)	1	1
True Negative (TN)	0	0
False Positive (FP)	1	0
False Negative (FN)	0	1

Table 1: The prediction from the algorithm is denoted by $\hat{y}$, and the true value from the training pair is denoted by y. If the two signatures in the pair are matching, this is denoted as *true*, and by the number *one* in the table, else the pair is non-matching, which is denoted as *false*, and by *zero* in the table.

When these outcomes are summarized the entire test set would result in a number of *TPs*, *TNs*, *FPs* or *FNs*. Ideally it would only be *TPs* and *TNs*, but since the classifier can do miss-classifications, *FPs* and *FNs* sometimes are bound to happen. For example, a balanced test set of 200 samples (100 matches and 100 non-matches) could result in 85 *TPs*, 15 *FNs*, 79 *TNs* and 21 *FPs*. If only the error rate was measured, it would be given by

$$\text{Error rate} = \frac{TPs + TNs}{TPs + FPs + TNs + FNs}, \tag{53}$$

which gives the result, $\frac{85+79}{200} = 0.82$.

Now in order to take consideration to the four different scenarios of miss-classifications, the **Sensitivity** or also called **True Positive Rate (TPR)** is given by

$$TPR = \frac{TPs}{TPs + FNs}. \tag{54}$$

This value gives the percentage of the positive examples (matches, $y = 1$) the algorithm succeed to predict. With the numbers from the above example the *TPR* would be $\frac{85}{85+15} = 0.85$.

The other measure is the **Fall-Out** or also called **False Positive Rate (FPR)**. False positive rate is given by

$$FPR = \frac{FPs}{FPs + TNs}. \tag{55}$$

This value measure the percentage of the negative examples (non-matches, $y = 0$) the algorithm fail to predict. In the example above the *FPR* would be $\frac{21}{21+71} = 0.21$.

Instead of measure the performance by the error rate, it is now measured by the *TPR* and the *FPR*. In this report the *TPR* versus *FPR* are plotted for the different models, which are shown in the Figures 24, 25, 26 and 27. The plots

are given by sweeping a threshold on where to separate the predicted matching samples from the predicted non-matching samples. Typically this threshold is varied from accepting all samples to rejecting all samples.

The **Optimal Threshold** for the different algorithms was chosen to be the point $(x = FPR, y = TPR)$, closest to the ideal point $(0 = FPR, 1 = TPR)$, where FPR is shown on the *x-axes* and TPR is the *y-axes*. The ideal point is given when all predictions are correct (no FP or FN are found).

This is made under the assumption that a FP is equally as incorrect as a FN. If this relation is not desired the optimal threshold could be picked to emphasize that a FP is worse than a FN or vice versa.

If a completely random classifier is used, it would make predictions only based on taking a random number between zero and one. Then the performance plot with TPR versus FPR, would look like a straight line going from point [0,0] to [1,1]. This hypothetical plot is shown in Figure 22.

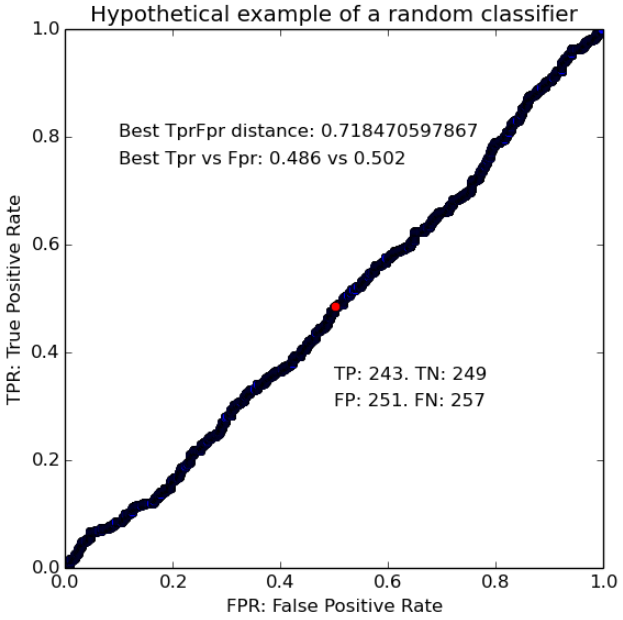


Figure 22: This figure displays the result of a random classifier. In this example a straight line is seen between the points [0,0] and [1,1]. The optimal threshold point is shown in red (0.502, 0.486).

Note that when the algorithm improves the line would constantly move closer to the upper left corner of the plot. This means that a perfect classifier would be following the *y-axis* up until the optimal threshold point *(1,0)*, and then continue parallel with the x-axis to point *(1,1)* as seen in Figure 23.

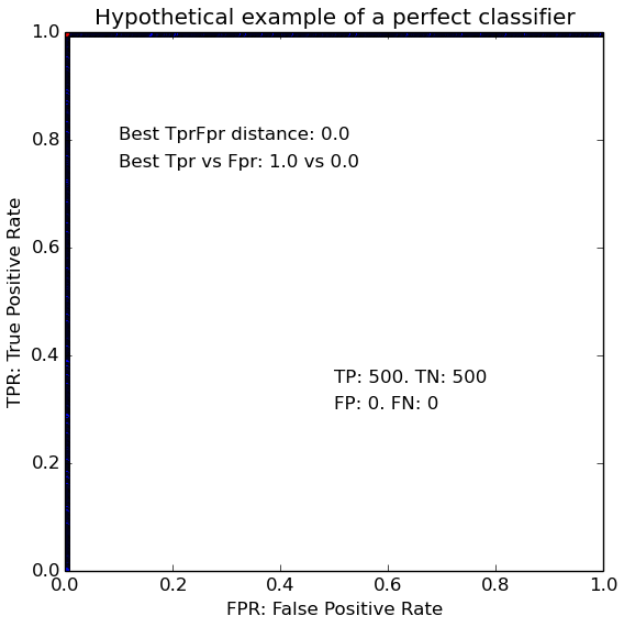


Figure 23: A hypothetical results plots with a theoretical perrfect classifier.

8.2 First Attempt of an Easy Compare Algorithm

The result from the first model, which was the Easy Compare Algorithm, is shown in Figure 24.

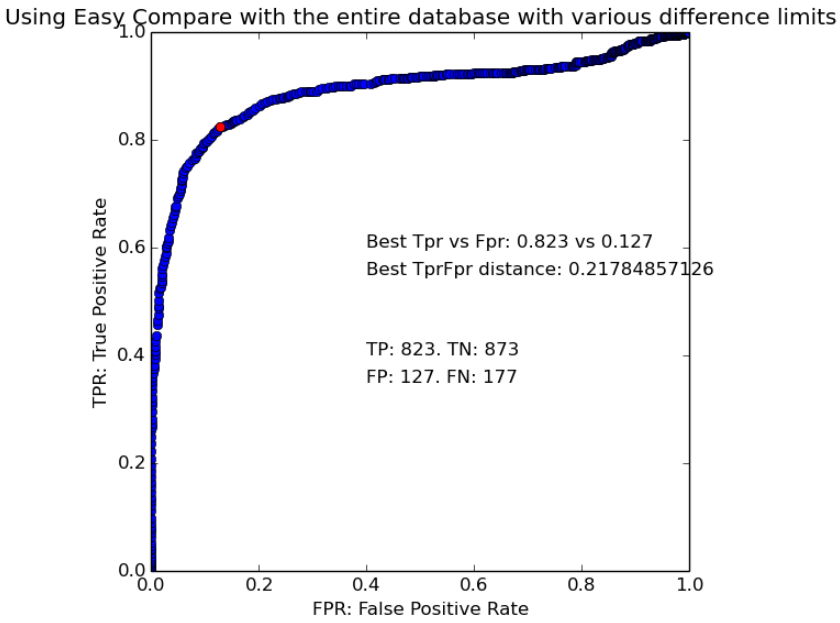


Figure 24: True/false-positive rate for *Easy Compare*-algorithm. The optimal value of the threshold, t, is denoted with a red dot (0.127,0.823).

Recall Equation 45 in Section 7.4, where the difference between the coordinates of signature A and B were measured. Then recall Equation 2 in Section 2.1. For the Easy Compare Algorithm, the *matching score* or *simularity* is measured as matching score $= -diff$. If those equations are combined, it gives

$$f(SignatureA, SignatureB) = \begin{cases} 0 & \text{if } diff \geq t \\ 1 & \text{otherwise.} \end{cases} \tag{56}$$

This means that the two signatures are predicted as a match or non-match depending on if their *diff* is smaller or greater than a certain threshold, t. Then the plot in Figure 24 was given by sweeping the threshold, t, over the interval $[0, \text{diff max}]$.

If a *false positive* is regarded as more incorrect than a *false negative*, the threshold, t, could be picked so the *false positive rate* is close to zero. In Figure 24 is the point $\approx [0, 0.4]$, on the curve, which means that a threshold could be chosen so the model never failed to predict a non-matching sample, but on the other hand it should fail to predict around 60 % of the matching samples.

8.3 Second Attempt with Logistic Regression

The second attempt with Logistic Regression can be seen in Figure 25. When this algorithm was run the amount of people in the database was 46. These people had contributed with *324* different signatures. A balanced set of pairs were then made, by the process described in Section 7.3 and in Figure 20.

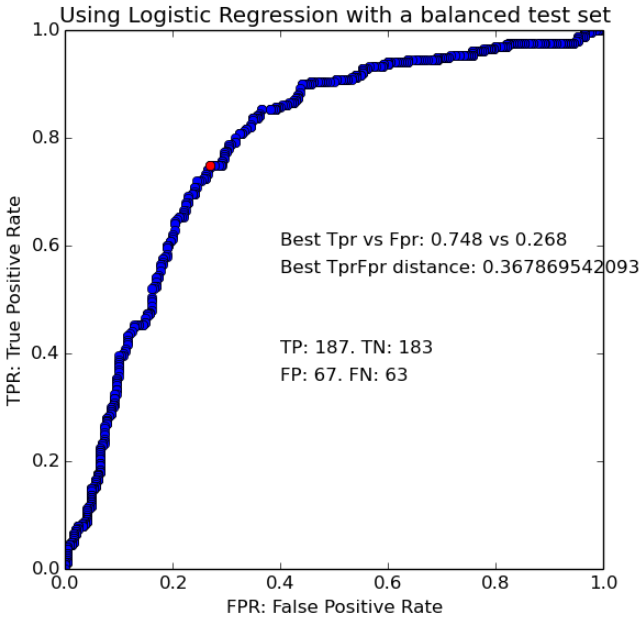


Figure 25: True/false-positive rate for classifier with logistic regression. The optimal value of the threshold, t, is displayed with a red dot (0.268,0.748).

For the multi-class classification problem the *argmax* is usually taken, see Equation 26. But for the signature verification problem, there is only two classes, and then just one class can be picked. If for example *class 0* is chosen, Equation 25 gives the probability of the sample belongs to *class 0*. The same applies for *class 1*. Because the probabilities in the binary case is given by $probability(\text{class } 0) = 1 - probability(\text{class } 1)$, it is sufficient to just look at one of the classes.

The threshold was then varied for *class 1*, and the classification was then given by

$$f(\mathbf{x}) = f(SignatureA, SignatureB) = \begin{cases} 1 & \text{if } P(y = 1|\mathbf{x}, \mathbf{W}, \mathbf{b}) \geq t \\ 0 & \text{otherwise,} \end{cases} \quad (57)$$

where the trained weights are **[W,b]**, and the input $\mathbf{x}$ consists of the two signatures, A and B, and the threshold, t, which can be varied on the interval $[0, 1]$.

The *matching score* is here given by $P(y = 1|\mathbf{x}, \mathbf{W}, \mathbf{b})$.

The Figure 25, belonging to the *Logistic Regression*, is actually worse then the curve in Figure 24, belonging to the *Easy Compare Algorithm (ECA)*. In the beginning, the curve does not follow the *y-axes* as well as the curve for the ECA and the optimal point is also worse in comparison. One positive thing to be said is that the classifier is at least better than a completely random algorithm (which would be a straight line from the point (0,0) to (1,1)), which can be seen in Figure 22.

8.4 Third Attempt with Multi-layer Perceptron

The results from the third attempt with the Multi-Layer Perceptron (MLP) is displayed in Figure 26.

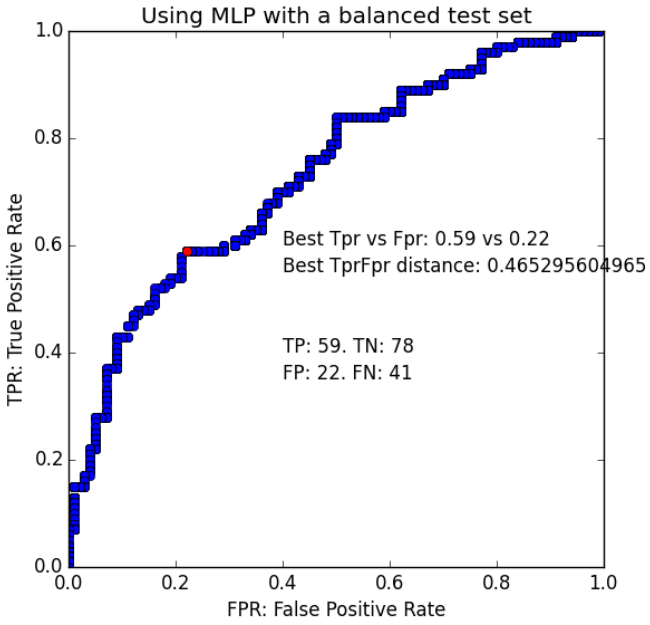


Figure 26: True/false-positive rate for the multi-layer perceptron. The optimal value of the threshold, t, is displayed in a red dot (0.336, 0.602).

This classification is done in the same way as the for the Logistic Regression, and is given by,

$$f(\mathbf{x}) = f(SignatureA, SignatureB) = \begin{cases} 1 & \text{if } P(y = 1|\mathbf{x}, \mathbf{W}_1, \mathbf{b}_1, \mathbf{W}_2, \mathbf{b}_2) \geq t \\ 0 & \text{otherwise.} \end{cases} \tag{58}$$

The only difference is that the *matching score*, $P(y = 1|\mathbf{x}, \mathbf{W}_1, \mathbf{b}_1, \mathbf{W}_2, \mathbf{b}_2)$, now depends on more learn-able parameters.

The plot in Figure 26 was made in a similar way as the one for Logistic Regression.

This model had a best error rate of 32 %.

An interesting thing to note form Figure 26, is that the result is worse than the results for the logistic regression and the easy compare algorithms.

8.5 Final Model with Convolution and Deep Architecture

The result from the final model with Convolution and a Deep Architecture can be seen in Figure 27. This algorithm was run with balanced sets, see Section 7.3 and Figure 20 on how the sets were build.

The same procedure of evaluating the trained model was made for this model, as for the logistic regression and the multi-layer perceptron, see Sections 8.4 and 8.3. The only difference here was that the *matching score* contained even more learn-able parameters. Sweep the threshold, t, over the interval [0,1]. Calculate the TPR and FPR values for each threshold and plot these values.

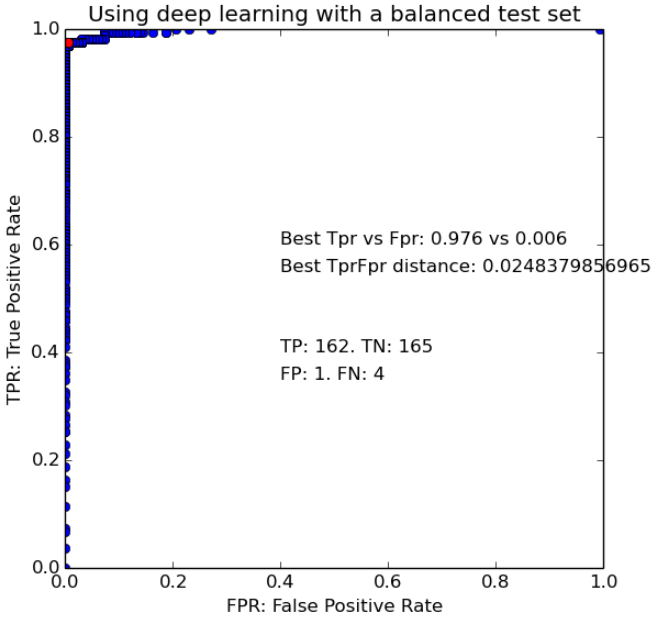


Figure 27: True/false-positive rate for convolutional neural network. The optimal value of the threshold, t, is displayed in a red dot (0.006,0.967)

Observing the results now in Figure 27, it is evident that this result is of much higher standard compared to the previous plots. The optimal value had

4 *FNs*, 162 *TPs*, 1 *FP* and 165 *TN* from the test set of size 322. This might suggest that the algorithm is better at distinguishing non-matches than it is at recognizing matching samples. In order to predict all matching samples, *TPs*, correctly with no *FNs* a high error rate must be accepted i.e. a total error rate of around 25 %. Compared to predicting all *TNs* correctly (with no *FPs*) it would result in a *TPR* of 0.97 which means a total error rate of around 3 %.

8.5.1 Time Comparison CPU vs GPU

A comparison between how long time the training took on the computer *Spaceship*, when training on *CPU versus GPU* were made, and the result can be seen in Table 2.

	times slower on CPU
CNN model 1	11.17
CNN model 2	6.9

Table 2: Model 1 was a convolutional neural network with 3 convolutional layers and with [20, 40, 60] feature maps in each layer and one channel for each parameter, while model 2 had 4 convolutional layers with [10, 10, 10, 10] feature maps in each layer and one channel for all the parameters.

8.6 Summary of All Models

In Figure 28 all the different results are shown in one picture. This is displayed to give an overview on how well the different algorithms performed compared to each other.

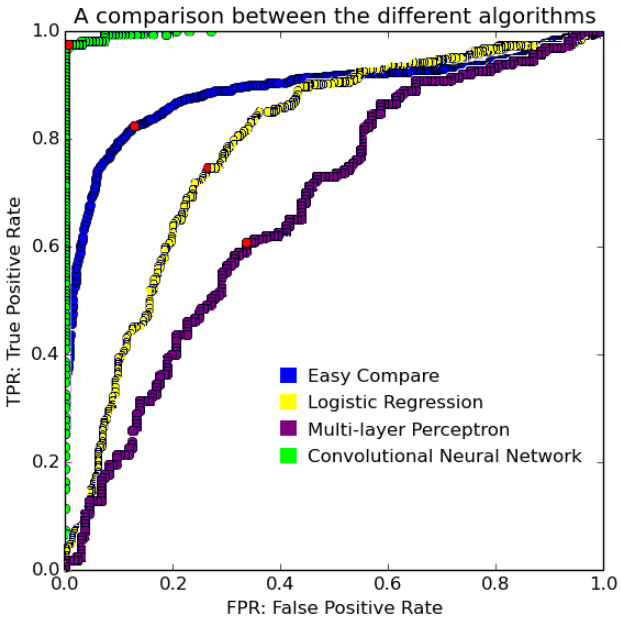


Figure 28: True/false-positive rate for all the different models.

9 Discussion

This section is used to discuss the results and other concepts regarding this project.

9.1 Data Collecting

An important subject to discuss is the data collecting process since it may have an impact on the results. When it comes to behavioural biometric parameters, they can of course be dependent on many different things. Some circumstances that can affect the intra-class variations of the signature sample are,

- the position of the signature contributor, whether he/she is standing up or sitting down,
- the mood of the signature contributor, for example if he/she is stressed, calm or nervous,
- the focus from the signature contributor on the task, whether he/she is focused or unfocused on writing his/her signature.

Not only how, but when the signatures are collected can affect the result. For example, in this project the seven genuine signatures were collected at the same time. In Section 1.2.3, the international Signature Verification Competition (SVC2004) was described. In this competition the genuine signatures were collected at two different occasions, first ten genuine signatures from one contributor were given, and one week later, ten more genuine signatures were collected. To collect signatures at different occasions can have some advantages, compared to when they are collected at the same occasion. The reason is that the circumstances described above may be a little bit different at the different occasions and it will probably increase the intra-class variations. If the algorithms are trained on this data set instead, it may lead to that the algorithms will be more invariant to the intra-class variations, which is an desirable property.

Another problem that occurred during the data collection phase was that some people was doubtful about giving away their signature, because of the ethical and social implications described in Section 2.2. In the SVC2004, this problem was solved by using "toy" signatures, i.e. the contributor design a new signature which they did not used in their daily use.

The quality of the data is also dependent on the equipment used during the enrollment phase. Some circumstances that can affect the quality of the data are,

- The type of pen, it could be ink, pencil or a quill pen.
- The type of substrate, it could be soft, hard or a digital tablet.
- The space that is available to write the signature on.

- The accuracy of the measurements.

In this project ordinary paper was used compared to digital tablets in previous work. The signature contributors often thought this gave a more authentic feeling when writing the signature, and may impact the quality of the signature in a good way. The pen also used an ink tip, which made the feeling even more authentic. The substrate under the paper was not standardized, and could vary between being directly on the table or on a heap of paper. This will probably also affect the features of the signature a little bit. In this project the signatures were given in a standardized box. The orientation and size of the signatures were never normalized and were a part of the features of the signature. Depending on the application this can be both good or bad. If the application has not got a standardized space for the signature, it could be a good idea to normalize those properties.

9.2 Pre-processing

As mentioned in Section 7.2, there were a few ways of standardizing the size of the input data so it could be processed by the learning algorithms. The method chosen was to linearly interpolate all data points in the signature to a standard length of the mean signature length. The problem is that all signatures that are longer than the mean length, will lose information. Similarly for shorter signatures, redundant data will be added.

One way to never lose data would be to re-sample all the signature to the max length instead. But this would mean that all (expect one) signatures receives redundant data that has to be processed and computed by the GPU/CPU. Considering that the training time is dependent on the number of float operations, this should increase the time for training which is not favorable. Redundant data can also impair the performance of the algorithms. The best solution of this problem would be to never add redundant data or remove any data. A more advanced method that was mentioned in [10] was to use the structure of the network and use different pooling-factors in the convolutional layers, so the network could handle input of different sizes. In this way no redundant data would be added and no information would be lost to the sample (except for the deduction of data during training, due to pooling).

The normalization of the x- and y-coordinates were made as described in Section 7.2. This could have been made in other ways, as described by [40], and may affect the performance of the training. But this was the first approach to solve the problem.

The normalization could have included normalization of the size and orientation of the hole signature, but if this is desirably is up to the application at hand.

How the training set was chosen for the final model, could have been done in a slightly different way. Instead of taking all equal pairs, a subset of those could have been taken randomly. Because the training was run over several training rounds, all equal pairs should eventually be used. If a smaller set was used, it

had probably decreased the training time.

9.3 Easy Compare Algorithm

First of all an important point to be made about the Easy Compare Algorithm (ECA) is that it was only intended to build a code framework which could later be adjusted for learning algorithms. ECA is not a learning algorithm so it might not be fair to compare this result to the other algorithms. Mostly because this algorithm considers the entire database as a test set since the prediction is made by an engineered feature *diff* (recall Equation 45). Putting this aside it is interesting to see that the first approach of a naive examination of the differences (defined by *diff*) between two signatures gave considerably satisfying results. It was not until the final model these results were surpassed.

9.4 Logistic Regression

When looking at the results from the logistic regression it is presumably possible that the input data is not linearly separable. Looking back at Figure 3, it can be explained that the linear separator only separated the output data with an accuracy of 70 %. The complexity of the problem and the way the input parameters are presented to the logistic regression results in an non-favorable algorithm.

9.5 Multi-layer Perceptron

There were some problems when trying to get the multi-layer perceptron working. Recalling the problems during training (see Section 7.6.2), it was all about varying the hyper-parameters to achieve a satisfying result. This resulted in the parameters listed in Section 7.6.3.

There could most likely be another set of hyper-parameters that would give a better result than the one attained. Another thing that was never tested was to let the algorithm train for more than a couple of days. This could have been made, but more emphasize was to develop the deep learning architecture which had proven great results on similar problems. So when the multi-layer perceptron gave an error rate that decreased during training, the development of the final, convolutional model started.

To summarize, when taking the complexity of the model and its performance into consideration, this model was not favorable for this application. But it was an important step on the way to develop the final model.

The increased complexity for both the logistic regression and the multi-layer perceptron has shown impaired performance, compared to both each other and the easy compare algorithm. So increased complexity is not always a good thing, and this is an important thing to keep in mind to future work.

9.6 Convolution Neural Network with Deep Architecture

As can be seen in Figure 27, this model performed very well on this database.

An interesting aspect of the result shown in Figure 27, is that the algorithm seems to be better at recognizing a non-match than it was at recognize a match, i.e. the model with the optimal threshold had one False Positive and four False Negatives. This means that the algorithm fail to predict four samples that in reality were a match, but only fail to predict one sample that in reality was a non-match. A potential reason could be that the non-matching samples contain a bigger variation of samples, see Section 7.3 and Figure 20, due to the samples were replaced in each training round. But the behavior could also be a part of the structure of the signature verification problem.

When looking back at the result for the logistic regression and the multi-layer perceptron, it is shown that the structure of the convolutional layers were very successful for this application. An interesting thing to think about is that the convolutinal layers only contained less than one percentage of the total set of parameters. This may suggest that it is worth to keep down the complexity of the model, and exploring the structure of the network instead.

Another question that arise is if the heavy, fully-connected layer (which contained 99.1 % of all the parameters in the convolutional neural network) really is needed. But the structure of using a fully-connected layer before the classifying layer is the common structure of convolutional neural networks. This has historically shown good results and was the reason this was used also in the project.

One important discussion to be made about the convolution neural network is that it is equivariant to translation as mentioned in Section 3.4.4, but is not equivariant to scaling and rotation, as described in Section 3.4.4. This results in that rotating the paper or writing the same signature but with a different size would result in different features for the network i.e. not the same representation of the signature. Considering that the enrollment phase was standardized, asking people to sign seven signatures in a box of the same size and in a row, could potentially lead to that the scaling and rotation of the signature was a valuable feature for the signature. For example some people often write their signature a bit skew, and in that case it can be an advantage if the network can detect this feature. But if this is not desirable, the signatures should probably be normalized regarding rotation and scaling before passing into the network, as discussed in Section 9.2.

The main problem when developing this model was the time of the training. This limited the possibility to try different selections of hyper-parameters and different structures of the layers. To make the training faster, a faster GPU could have been used or a reduction of the number of trainable parameters could have been made.

To summarize, this model showed a great result. However, when putting the good result aside, some weaknesses of this model is the long training time and the complication in understanding the output from the model, compared to the engineered features as the easy compare algorithm and the methods in

Mattisson's work, [42].

9.7 Performance Evaluation

As mentioned several times earlier in this report the evaluation of which threshold to chose for the different algorithms has been non-biased when it comes to whether a False Positive or False Negative is worse. This results in that those two types of errors have been treated the same, but it can be discussed if this property is desired. If the systems main objective is to detect fraud then it might be considered better to reject a true signature (False Negative) then accept a fraud (False Positive). Another example is used in the justice system where it is better to let one hundred guilty men go free then to sentence one innocent man to life in prison.

9.8 Comparison between the Different Algorithms

The first thing to be said is that the easy compare algorithm provides the best results up until the deep learning algorithm, which has been trained on a GPU for one and a half week. This is really interesting since it means that the naive approach in how to compare two signatures with an engineered feature was strikingly good. Showing that the human mind is capable of constructing an acceptable guess of what is important to look at for a solution.

When it comes to the logistic regression (LR) and the multi-layer perceptron (MLP) algorithms they were mainly used to understand the signature verification problem and gather an understanding of learning algorithms. The focus on these algorithms (LR and MLP) was never to achieve top performance but rather to obtain the knowledge required for a deep learning algorithm.

When it comes to understanding how the prediction of the different algorithms works, it is much easier to understand the engineered features, compared to the weights of the trained model, which becomes increasingly more complex as the model increases in size.

10 Future Work

This section is intended to provide some suggestions of further work and studies, which could be a continuation of this project.

10.1 Data Collecting

The very first step on the signature verification problem is to be able to separate genuine signatures form each other. The next step will be to move on to other types of forgeries and see how well the final model in this project perform on those data sets.

If it is possible to get an algorithm which will work on all the other types of forgeries and if the accuracy of the algorithm is good enough (i.e. close to ideal),

the data collection could be linked to general identification procedures such as for passport controls and the data could be saved on a chip in the passport. Because the signature is a behavioral biometric parameter, the samples should be replaced after X number of years, which is the general procedure for passports and other identification cards.

In future work it would also be interesting to study how the signatures changes over time, for example a couple of weeks or even years.

In the future development of the algorithm it is then important to mimic the real world as much as possible and collect samples at different occasions.

10.2 Pre-processing

The normalization could be made in different ways. If the input data is normalized to be symmetric around zero, the gradient in the training step will variate less and this will probably lead to improved training [40]. In this project one type of normalization was used, but other types of normalization are also worth trying in future development.

10.3 Convolution Neural Network with Deep Architecture

It is an engineering task to find out which model selection that gives the best results. In this project there was not enough time to variate the structure of the final model, and analyze this further.

To improve the developed model further, the structure of the network could be utilized better, where a trade off between needed complexity is considered, i.e. the minimum number of parameters that can be used without losing accuracy. The network could perhaps be implemented to work on an adaptable input length and/or be implemented on sub problems.

In Section 1.2.3, the results from the competition showed that the models performed better on the data set, that only contain the coordinates and not the pen orientation and pressure. It would be interesting to customize the final model developed in this project to only work on the coordinates and see how well it performed.

The GPU based, max-pooling, convolutional neural networks has shown great results on image recognition through history, but even better results has been shown by GPU based, ensemble, max-pooling, convolutional neural networks. To improve this model even more, the model could also use ensemble learning in future development.

11 Conclusion

In this report a *deep learning* algorithm has been developed to deal with the signature verification problem. In order to solve the problem a code framework was built with a database of signatures, a classifier and an evaluation method. Several different classifying algorithms were tested before the final convolutional neural network was developed. The result from this final model was a true positive rate (sensitivity) of 96.7 % and a false positive rate (fall-out) of 0.6 %. The suggested solution for the problem shows promise, but future studies are recommended.

References

[1] Deep learning comminity. http://deeplearning.net/tutorial/. Accessed: 2015-07-28.

[2] historyofinformation.com. http://www.historyofinformation.com/expanded.php?id=2614. Accessed: 2015-09-05.

[3] Tutorial for classification with convolutional neural network on the mnist data set. http://deeplearning.net/tutorial/logreg.html#lenet. Accessed: 2015-07-28.

[4] Tutorial for classification with logistic regression on the mnist data set. http://deeplearning.net/tutorial/logreg.html#logreg. Accessed: 2015-07-28.

[5] Tutorial for classification with multi-layer perception on the mnist data set. http://deeplearning.net/tutorial/logreg.html#mlp. Accessed: 2015-07-28.

[6] Where to find the package, where hid-recorder.c is included. http://bentiss.github.io/hid-replay-docs/, note = Accessed: 2015-08-16.

[7] Dilnya Mahmood Jafar Agnieszka Nabrdalik. Kan biometri öka säkerhetsnivån på svenska sjukhus? Master's thesis, Blekinge Tekniska Högskola, Avdelning för Elektroteknik, 2012.

[8] Cyrus Bakhtiari-Haftlan. Arabic online handwritten recognition. Master's thesis, Lund Institute of Technology, Centre for Mathematical Sciences, 2008.

[9] P. Baldi and Y. Chauvin. Neural networks for fingerprint recognition. *Neural Computation*, 5(3), 1993.

[10] Yoshua Bengio, Ian J. Goodfellow, and Aaron Courville. Deep learning. Book in preparation for MIT Press, 2015.

[11] Christopher Michael Bishop. Pattern recognition and machine learning. springer, 2006.

[12] Sandra Blakeslee. Behind the veil of thought: Advances in brain research; in brain's early growth,timetable may be crucial. *The New York Times*, 1995.

[13] Kumar Chellapilla, Sidd Puri, and Patrice Simard. High performance convolutional neural networks for document processing. In *International Workshop on Frontiers in Handwriting Recognition*, 2006.

[14] D. C. Ciresan, U. Meier, J. Masci, and J. Schmidhuber. A committee of neural networks for traffic sign classification. In *International Joint Conference on Neural Networks (IJCNN)*, pages 1918–1921, 2011.

[15] D. C. Ciresan, U. Meier, J. Masci, and J. Schmidhuber. Multi-column deep neural network for traffic sign classification. *Neural Networks*, 32:333–338, 2012.

[16] D. C. Ciresan, U. Meier, and J. Schmidhuber. Multi-column deep neural networks for image classification. In *IEEE Conference on Computer Vision and Pattern Recognition CVPR 2012*, 2012. Long preprint arXiv:1202.2745v1 [cs.CV].

[17] Dan Claudiu Ciresan, Alessandro Giusti, Luca Maria Gambardella, and Jürgen Schmidhuber. Mitosis detection in breast cancer histology images with deep neural networks. In *Proc. MICCAI*, volume 2, pages 411–418, 2013.

[18] Dan Claudiu Ciresan, Ueli Meier, Luca Maria Gambardella, and Jürgen Schmidhuber. Deep big simple neural nets excel on handwritten digit recognition. *CoRR*, abs/1003.0358, 2010.

[19] David Danowsky. Cyrillic handwriting recognition using support vector machines. Master's thesis, Lund Institute of Technology, Centre for Mathematical Sciences, 2006.

[20] Stefano Rodoà Data Protection Working Party of the European commission. Working document on biometrics. *WP80*, 2003.

[21] E. D. Dickmanns, R. Behringer, D. Dickmanns, T. Hildebrandt, M. Maurer, F. Thomanek, and J. Schiehlen. The seeing passenger car 'VaMoRs-P'. In *Proc. Int. Symp. on Intelligent Vehicles '94, Paris*, pages 68–73, 1994.

[22] C. Petrinin E. Mordini. Ethical and social implications of biometric identification technology. *Annali Dell'Instituto Superiore di Sanita*, 43(1):5-11, 2007.

[23] J.L. Elman. *Rethinking Innateness: A Connectionist Perspective on Development*. A Bradford book. Kluwer, 1998.

[24] Jimmy Engström. On-line handwriting recognition. Master's thesis, Lund Institute of Technology, Centre for Mathematical Sciences, 2003.

[25] K. Fukushima. Neural network model for a mechanism of pattern recognition unaffected by shift in position - Neocognitron. *Trans. IECE*, J62-A(10):658–665, 1979.

[26] K. Fukushima. Neocognitron: A self-organizing neural network for a mechanism of pattern recognition unaffected by shift in position. *Biological Cybernetics*, 36(4):193–202, 1980.

[27] Kunihiko Fukushima. Artificial vision by multi-layered neural networks: Neocognitron and its advances. *Neural Networks*, 37:103–119, 2013.

[28] C. F. Gauss. Theoria combinationis observationum erroribus minimis obnoxiae (theory of the combination of observations least subject to error), 1821.

[29] Carl Friedrich Gauss. Theoria motus corporum coelestium in sectionibus conicis solem ambientium, 1809.

[30] D. H. Hubel and T.N. Wiesel. Receptive fields, binocular interaction, and functional architecture in the cat's visual cortex. 160, 1962.

[31] David H Hubel and Torsten N Wiesel. Receptive fields and functional architecture of monkey striate cortex. 195, 1968.

[32] ICPR 2012 Contest on Mitosis Detection in Breast Cancer Histological Images. IPAL Laboratory and TRIBVN Company and Pitie-Salpetriere Hospital and CIALAB of Ohio State Univ., http://ipal.cnrs.fr/ICPR2012/, 2012.

[33] A.K. Jain, A.A. Ross, and K. Nandakumar. *Introduction to Biometrics*. SpringerLink : Bücher. Springer, 2011.

[34] Anil K. Jain, Arun Ross, and Salil Prabhakar. An introduction to biometric recognition. *IEEE Trans. on Circuits and Systems for Video Technology*, 14:4–20, 2004.

[35] Alex Krizhevsky, I Sutskever, and G. E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems (NIPS 2012)*, page 4, 2012.

[36] O. Abdel-Hamid L. Deng and D. Yu. A deep convolutional neural network using heterogeneous pooling for trading acoustic invariance with phonetic confusion. Technical report, Microsoft Research, One Microsoft Way, Redmond, WA, USA York University, Toronto, ON Canada, 2013.

[37] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. Handwritten digit recognition with a back-propagation network. In D. S. Touretzky, editor, *Advances in Neural Information Processing Systems*.

[38] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. Back-propagation applied to handwritten zip code recognition. 1(4), 1989.

[39] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, November 1998.

[40] Yann Lecun, Leon Bottou, Genevieve B. Orr, and Klaus-Robert Müller. Efficient backprop, 1998.

[41] Adrien Marie Legendre. *Nouvelles méthodes pour la détermination des orbites des cometes*. F. Didot, 1805.

[42] Fredrik Mattisson. On-line signature verification using a multi-judge strategy. Master's thesis, Lund Institute of Technology, Centre for Mathematical Sciences, 2001.

[43] W. McCulloch and W. Pitts. A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, 7:115–133, 1943.

[44] Jonas Morwing. Recognition of cursive handwriting. Master's thesis, Lund Institute of Technology, Centre for Mathematical Sciences, 2001.

[45] Committee of SVC 2004. The results of the svc 2004. http://www.cse.ust.hk/svc2004/results.html, 2004. Accessed: 2015-07-28.

[46] M. Ranzato, C. Poultney, S. Chopra, and Y. Lecun. Efficient learning of sparse representations with an energy-based model. In *Advances in Neural Information Processing Systems (NIPS 2006)*, pages 1137–1144, 2006.

[47] L. Roux, D. Racoceanu, N. Lomenie, M. Kulikova, H. Irshad, J. Klossa, F. Capron, C. Genestie, G. Le Naour, and M. N. Gurcan. Mitosis detection in breast cancer histological images - an ICPR 2012 contest. *J. Pathol. Inform.*, 4:8, 2013.

[48] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Education, 3 edition, 2009.

[49] Jorge Saint-Aubyn. Using linguistic knowledge and edit distance to improve recognition or fuzzy cursive handwriting. Master's thesis, Lund University, Faculty of Engineering, Centre for Mathematical Sciences, 2008.

[50] Jurgen Schmidhuber. Deep learning in neural networks: An overview. 2014.

[51] Patrice Y. Simard, Dave Steinkraus, and John C. Platt. J.c.: Best practices for convolutional neural networks applied to visual document analysis. In *In: Int'l Conference on Document Analysis and Recognition*, pages 958–963, 2003.

[52] J. Stallkamp, M. Schlipsing, J. Salmen, and C. Igel. The German traffic sign recognition benchmark: A multi-class classification competition. In *International Joint Conference on Neural Networks (IJCNN 2011)*, pages 1453–1460. IEEE Press, 2011.

[53] J. Stallkamp, M. Schlipsing, J. Salmen, and C. Igel. Man vs. computer: Benchmarking machine learning algorithms for traffic sign recognition. *Neural Networks*, 32:323–332, 2012.

[54] Brian Kingsbury Tara N. Sainath, Abdel-rahman Mohamed and Bhuvana Ramabhadran. Deep convolutional neural networks for lvcsr. 2013.

[55] Theano. Software. http://deeplearning.net/software/theano/. Version: 0.6.0, Accessed: 2015-07-28.

[56] V. Vapnik. *The Nature of Statistical Learning Theory*. Springer, New York, 1995.

[57] Juyang Weng, Narendra Ahuja, and Thomas S Huang. Cresceptron: a self-organizing neural network which grows adaptively. In *International Joint Conference on Neural Networks (IJCNN)*, volume 1, pages 576–581. IEEE, 1992.

[58] Dit yan Yeung, Hong Chang, Yimin Xiong, Susan George, Ramanujan Kashi, Takashi Matsumoto, and Gerhard Rigoll. Svc2004: First international signature verification competition. In *In Proceedings of the International Conference on Biometric Authentication (ICBA), Hong Kong*, pages 16–22. Springer, 2004.

Master's Theses in Mathematical Sciences 2015:E40
ISSN 1404-6342

LUTFMA-3282-2015

Mathematics
Centre for Mathematical Sciences
Lund University
Box 118, SE-221 00 Lund, Sweden

http://www.maths.lth.se/