VRIJE
UNIVERSITEIT
BRUSSEL

Master thesis submitted in partial fulfillment of the requirements for the degree of
Master of Science in Applied Sciences and Engineering: Computer Science

# CANVAS BASED AUTHENTICATION METHODS WITH DEEP LEARNING

Yannick Merckx

2016 - 2017

Promotor: Ann Nowé

Advisors: Kyriakos Efthymiadis, Florentin Rochet & François Koeune

**Science & Bio-Engineering Sciences: Departement of Computer Science**

VRIJE
UNIVERSITEIT
BRUSSEL

Proefschrift ingediend met het oog op het behalen van de graad van Master in de Ingenieurswetenschappen: Computerwetenschappen

# CANVAS GEBASEERDE AUTHENTICATIEMETHODES MET DEEP LEARNING

Yannick Merckx

2016 - 2017

Promotor: Ann Nowé

Begeleiders: Kyriakos Efthymiadis, Florentin Rochet & François Koeune

**Science & Bio-Engineering Sciences: Departement of Computer Science**

## Abstract

In the last years, canvases have shown promising results with regard to device fingerprinting. A canvas is a HTML5 element, which allows an application or a user to draw an arbitrary drawing. In this thesis, we develop new authentication methods that are seamless and based on canvases and machine learning techniques. Seamless authentication means that a user can authenticate without the need of requesting the credentials. Furthermore, the methods are developed with the intent of weak authentication. This means that the developed authentication methods are intended for non-sensitive services. They can also be part of a larger authentication chain, where they are used as a light authentication test and have the other methods in the chain as fallback for stricter authentication.

The outline of the developed authentication methods is as follows. First, the user is required to register his device to the system by submitting his email and an amount of random canvases. After the registration, the user is able to identify himself to the system with the registered device by sending again his email and an amount of random canvases. The goal of this thesis is to find working methods to decide whether or not a canvas originate from the given user. This is a new problem, where no existing research is available. We approach this problem as a binary classification problem, where we need to decide whether canvases come from the same device or not.

In total, three methods are developed and evaluated. The first method is based on a K-NN classifier and is not suitable for canvas based authentication. It has a bad performance and it does not scale for large datasets. The second method uses convolutional networks and is the best performing method of all three. This method can distinguish canvases with a high accuracy, but has the adverse requirement of training a new convolutional network, every time a new device is added to the authentication system. The last and third method uses an autoencoder to retrieve compressed representations of the canvases and uses these representations to calculate distances between the canvases. The method makes use of a threshold related to the distance, to decide whether or not the canvases are originating from the same device. This method offers the flexibility to the security expert to choose the threshold freely and to define his own authentication policy. Furthermore, it is a generic method. When a new device is added to the system, no new training is needed. In this thesis, the authentication method is evaluated with different threshold configurations and distance measures and shows reasonable results for the optimal configuration.

**Acknowledgements**

The process of making a Masterthesis is something you don't do alone. This is why I like to thank a few people, who supported me during the process. First of all I like to thank my parents and my girlfriend for the unconditional support during the whole process. Also I like to thank my advisor Kyriakos for his support and effort. He was always the whole year available to council me and deliver me fast answers on my questions. Also the people of UCL, Florentin and François, have my gratidude for the pleasant collaboration. At last I like to thank Ann, she kept us on the right track and helped us when necessary.

# Contents

# List of Figures

# Chapter 1

# Introduction

Nowadays we can't think of a world without the internet. Current statistics show that we have more than 3.6 billion internet users today, which is around 40% of the world population (Consortium, 2017). The impact of the internet is something we can't oversee. It influences our social life and how we communicate with each other, companies, the government and much more. Authentication plays a central role in the use of the internet. How do we assure someone's identity via a device and how do we keep it simple for the user? Many methods have been already developed, such as the use of a password or two factor authentication. Despite the already developed methods, authentication is still an ongoing research.

With the introduction of HTML5, many new features were added. One of these features is the canvas element. A canvas is a container on a webpage, which allows the application or the user to make drawings. Canvases are used for visualisation, but also for web tracking. Research has shown that is possible to define a sort of canvas fingerprint, with a high-entropy (Mowery & Shacham, 2012). More over, 5% of the top 100000 websites are using canvases as a tracking method (Acar et al., 2014). This finding inspired us to take the usages of canvases further and use them for an authentication method. There are several reasons, why we would do this. First of all the flexibility of canvases, the amount of possibilities that we can draw on a canvas is endless. Further, it is supported by every modern web browser and it is easy to integrate in applications. Overall is canvas fingerprinting showing a lot of potential. Mainly because the current techniques are relatively simple and are performing well. Also, we see a rather unexplored domain, where we combine machine learning with canvases.

Machine learning techniques learn a concept based on the given data. These concepts can be very complex, which are not possible to code manually. We expect that machine learning will allow us to extract very complex features of a canvas that are usable for authentication. We focus mainly on deep learning, a subdomain of machine learning. Deep learning has the interesting property of learning better with the more data. In the past deep learning has also shown to have very good results in image classification (Krizhevsky et al., 2012) and we believe that the solution towards our learning problem can be found in this subdomain.

In this research, we develop several new authentication methods by bringing canvases and machine learning together.

## 1.1 Problem Statement

The goal of this thesis is to develop a new authentication method, based on canvases. Currently there is no research available towards this subject. Mainly, canvases are used for simple tracking methods, but none of them offer the complexity needed for an authentication method. In this research, we develop and evaluate several authentication methods that combine machine learning techniques with canvases. The methods are developed for weak authentication. This means that they are seamless and intended for non-sensitive services. The developed method could also be part of a larger authentication chain, where other authentication methods activate when our method fails. It could serve as an initial light weight authentication test, while the others serve as a fallback for more stricter authentication.
Basically, we start from scratch with our authentication method. The only two things that are given in this research are the design of the authentication method and the property of GPU leakage in canvases. For the design of the authentication method, we work together with the ICTM team at the UCLouvain. They delivered the outline of the method. Further, we have the findings by Mowery & Shacham (2012). Their findings are the main motivation for this research. They showed that canvas drawings can be separated because of differences in hardware and software. This result motivates us to search for more complex features in the canvases and we hope to find them with machine learning.

This thesis has some important challenges, which requires us to build up the whole method from the ground up. First, we investigate if the canvas drawings are unique enough and contain sufficient information. We are also responsible for the data collecting, since the canvases are not predefined at the start of this thesis. Further, we experiment with different machine learning techniques by starting with rather simple methods and continuing with more advanced methods, such as deep learning. At the end of the experiments, we evaluate the tested methods and investigate which techniques are eligible for authentication.

Our contribution consists of developing and evaluating seamless authentication methods that make use of canvases and machine learning and are intended for weak authentication.

## 1.2 Thesis Structure

This thesis is gradually build up to give the reader a sufficient understanding of what our contribution represents. In chapter 2, we describe authentication and how it is currently done. Next in chapter 3, we provide background on canvases and their possibilities. Then, we continue with machine learning in chapter 4. Chapter 5 is focusing on Artificial network and Chapter 6 is extending the previous chapter by providing background on Deep learning. Eventually in chapter 7, we experiment with and evaluate several authentication methods. At last we draw a conclusion and discuss future work in chapter 8. The contribution of this thesis is mainly in Chapter 7 and 8. They contain all the experiments and conclusions about the authentication methods we have developed and tested.

# Chapter 2

# Authentication

Authentication can be done in many ways. If we want to develop an authentication method, it is important to know what already exists and how authentication is currently done. In the chapter we separate two types of authentication, namely explicit and implicit authentication. For every type we take a closer look at the current authentication methods in use and we discuss on a high level the working of the method, the advantages and the disadvantages.

## 2.1 The Definition

If we speak about Authentication, what do we exactly mean and why do we use it? Well, in this research we define *Authentication*, related to computers, as (Dictionary, 2017):

*The process or action of verifying the identity of a user or process.*

When the authentication happens by human-to-computer interaction, we call it *User Authentication* (Rouse, 2014).
In general we use authentication to allow authorized people or processes to access certain resources and disallow unauthorized people. This can go from logging in on Facebook to paying your coffee with your smartphone. Both examples require authentication. How authentication happens nowadays, will be explained in the following sections.

## 2.2 Explicit Authentication

Explicit authentication is an authentication mechanism, where the user is required to explicitly give some unique identifiable information. Most of our modern day authentication methods fall under this category. There exist several types of identifiable information. We tried to categorize and discuss them in each subsection.

### 2.2.1 Password

Authentication by a password is probably the oldest and popular authentication method we know. It is a simple method, which is easy to verify. The user has to state his identity and the corresponding secret string or so called password to the authentication system. The system checks if the given password matches the password for the given user and allows access if it is a match.

Despite the fact that it is easy to verify and a simple protocol for the user, there are a lot of disadvantages. First of all, the user has a big liability. If the user chooses a weak password, which is easy to guess or brute forcible, the authentication is completely compromised. The authentication system can try to obligate the user to choose a strong password by requiring a certain length or certain type of characters. But all of this reduces the comfort of the user. Secondly, we have also the liability of the authentication system. Nowadays, we see several data breaches a year, where millions of passwords are stolen. As example, the site *haveibeenpwned* [1] keeps track of all the data breaches in the world and has currently the password information of more than 3.7 billion people. At last, we have the fact that the password policy of a user across different password based authentication systems, can compromise the authentication of that user. If the user uses always the same password, only one authentication system needs to be compromised to comprise all authentication systems for that user. There exists variants, who try to solve these issues. One example is the use of an One-time Password Device (OTP-device), which generate every time a different password. This changes the authentication factor from knowledge based to possession based. The authentication does not require the user to have the knowledge of a password, but does require the possession of an OTP device.

### 2.2.2 Inherent Authentication

Inherent authentication is authentication that requires integral features of the system or the user. For user authentication, these are biometric features such as facial recognition, fingerprints and irisscanning. All these features require the user to explicitly submit the identifiable feature. Today, fingerprintscanners are widely integrated in most high-end smartphones and allow user to authenticate to several platforms. Nevertheless have biometric features also drawbacks (Mokrohuz & Kazymyr, 2014). First of all, biometric authentication mechanism are not applicable in all devices. Technical restriction on for example mobile devices and the ethical consideration are here the main reason. Secondly, it is still possible to circumvent the current biometric authentication mechanism of today.

### 2.2.3 Two-factor Authentication

An upgrade of the two previous authentication methods is two-factor authentication. This method has two layers of authentication instead of one and requires the user to explicitly identify two times. This happens in two phases. In the first phase, the user identifies by giving a password or a integral feature such as a fingerprint. In the second phase, the authentication system generates another password and delivers it to the user via a device that is expected to be in the possession by the user. In most cases, this is done by sending the user a text message to the associated phone number.

In summary, we have now seen several authentication methods, which require explicit participation of the user, whether it is submitting a password or scanning a fingerprint. In the next section, we will discuss implicit authentication, where a user is not required to explicitly identify himself.

## 2.3 Implicit Authentication

Implicit authentication is mostly applied as a tracking method. On most websites nowadays, the user gets assigned a cookie, which is used to track the behaviour of the user on that website. We

[1] `https://haveibeenpwned.com/`

call this implicit authentication, since the user does not need to actively identify himself. The authentication happens seamless. In section 2.3.2, we discuss several tracking methods. Although, implicit authentication can also be used for access control, by looking at the behaviour. Behaviour based authentication will be discussed in the next section.

### 2.3.1 Behaviour based Authentication

Behaviour based authentication is in an uplift, since the development of mobile devices. The behaviour of the user can be considered as an unique fingerprint and the current mobile devices allow us to track this. The generation of the behavioral fingerprints happens by the use of machine learning. In 2008, Saevanee & Bhatarakosol (2008) suggested a user authentication mechanism based on the touchpad acting of the user by looking at the keystroke dynamics and the finger pressure. He was able to achieve an accuracy of 90 % or more. More recently, another company called *UnifyID* announced on Techcrunch Disrupt 2016 to have developed a behavioral fingerprint based on basically all sensor data of a mobile device, such as GPS, the accelerometer and much more[2]. They claim to achieve a 99,9 % True rejection rate on these fingerprints. Google is also doing actively research on this domain with projects as *Project Abacus* and the *Trust API*. These projects involve the use of behavioral data to generate a trust score that can be used by Android application to allow or restrict access or actions.[3]
Using behavioral data has two big advantages. To begin with, it is very difficult to recreate, which makes it almost impossible to imitate a user. Secondly it is very user friendly, since everything can be done automatically.

All these examples show that there is more and more research done towards authentication and machine learning. More over, this indicates that there is a password-free future possible, with machine learning as a key component.

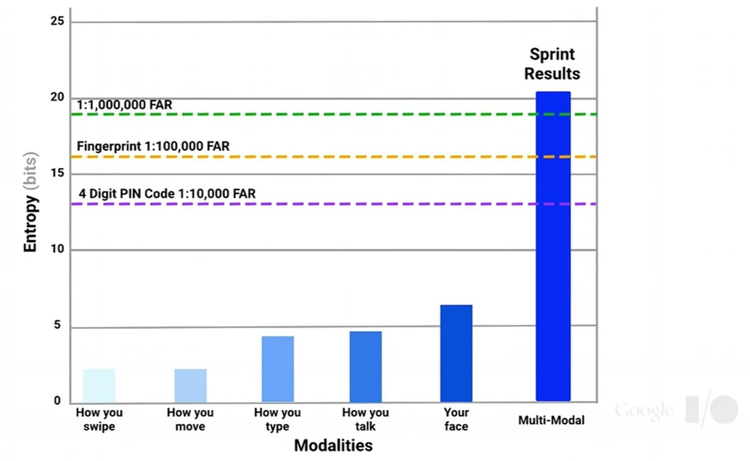


Figure 2.1: A graph about the entropy of the fingerprints by Project Abacus. The multi-model outperforms clearly the other modalities, with a false acceptance rate (FAR) of 1 in a million.

Source: Google I/O 2015

[2]Presentation RSA conference 2017: `https://www.youtube.com/watch?v=2puy3GoXes8`
[3]Google I/O Presentation: `https://www.youtube.com/watch?v=8LO59eN9om4`

### 2.3.2 Tracking

We are all familiar with web tracking. Due the stateless internet, website are obligated to save a sort identifier in your browser to make tracking possible. This all happens in the background, without conscious user actions and can be considered as a sort of implicit user authentication. Obviously, this is a very simple method, since the user can remove the cookies or alter them. But we can draw inspiration from the more advanced tracking systems for the development of our authentication system. In this section, we focus on prominent persistent user tracking systems, since these are more fitting for an authentication system. With persistent, we mean tracking systems, which are able to circumvent the user's tracking preferences. They are also advanced in a sense that they are hard to detect, control, block or remove (Acar et al., 2014).

**Evercookies** is a tracking mechanism, which respawns HTTP cookies by using multiple storage vectors, such as Flash cookies, local storage, session storage and ETtags (Ayenson et al., 2011). This allows trackers to identify users, even if the HTTP cookie is deleted or the user is using a different browser. Evercookies mainly work because the deletion process in browser is often incomplete and never deletes all storages vectors, which enables the trackers to restore the cookies.

**Cookiesyncing** is a tracking method, where third-party domains share pseudo User ID's, stored as a cookie, to keep track of the user's browser history. This happens in the backend, where a graph with connected websites is constructed. Note that, once the users clears its cookie, a new pseudo user ID is generated and a new user history graph is constructed. But if one of the sites is able to respawn its cookies, the old pseudo user ID is sent to the cookie syncing tracker and the old and the new graph are merged. This makes cookie syncing a powerful and persistent tracking mechanism.

**Canvas fingerprinting** is a rather new technique. It was introduced in 2012 by Mowery & Shacham (2012). They showed that is was possible to invisible extract a rather unique fingerprint by drawing text and using WEBGL scene on HTML5 *<canvas>* elements. Later in 2014, Acar et al. (2014) discovered that over 5% of the top 1000000 Alexa sites was using this technology. More about this technique, will be discussed in chapter 3

Overall, we have seen in this chapter different methods of authentication, explicitly and implicitly, which gave us a good overview on what is currently going on in the landscape of authentication. Now, we continue with the next chapter, which is completely focused on canvases, a key component of this research.

# Chapter 3

# Canvases

What are canvases, how are they created and why do we use them? All of these question will be answered in this chapter.

## 3.1 Definition

We define a canvas as a browserdrawing, created with the *<canvas>* element of HTML5. the *<canvas>* element is a container, which can be used for graphical drawings. It is similar to the canvas of a painter artist, but then for your browser. Basically, there are almost no limits to the graphical possibilities of the canvas. It can drawn everything in 2D, from shapes and gradients to text and images. All modern browsers support canvases. The only requirement is the enabling of javascript, since this is needed to draw on the canvases. Note that, the *<canvas>* element does not require to be visible in the browser. A canvas can be drawn by and extracted from a device, while being completely invisible for the user. This feature made canvases a tool of interest for tracking and device fingerprinting.

## 3.2 Canvas fingerprints

The first mentions about canvas fingerprinting where made in 2010, during the discussions on the WebGL mailing list. There were some different opinions on whether or not to make the GPU and driver information available for the javascript WebGL render engine. Steven Baker (Baker, 2010) disputed that he was able to identify the GPU without any extra information. He suggested that it was possible to build up a database with benchmarks, such as the vertex measure performances, and successfully distinguish different hardwares. Later added Benoit Jacob (Jacob (2010)) that GPU rendering analysis is also an option, since most rendering, also WebGL, goes through the GPU and has config-based differences. Eventually, Mowery & Shacham (2012) introduced a system that made canvas fingerprinting possible and experimentally applied the theoretical assumptions of Baker and Jacob. They showed that it was possible to identify hardware and software, due the leakage of GPU-based rendering. More over, they did this with only the use of text rendering and WebGL scenes. The setup of the system was very simple. They rendered a pangram on the canvas in a certain font and compared these canvasdrawings pixel wise. A pangram is a sentence that contains all the letters of the alphabet at least once. Empirical results showed small differences between different operating systems, concerning the text rendering of for example the 30-year old font Arial. As a conclusion, they calculated a distribution entropy of 5.73 bits for the fingerprints generated by their system and believed there was still room for improvement.

Summarized, Canvases are drawings from the <*canvas*> element of HTML5 and can be grossly distinguished by using a few simple tests.

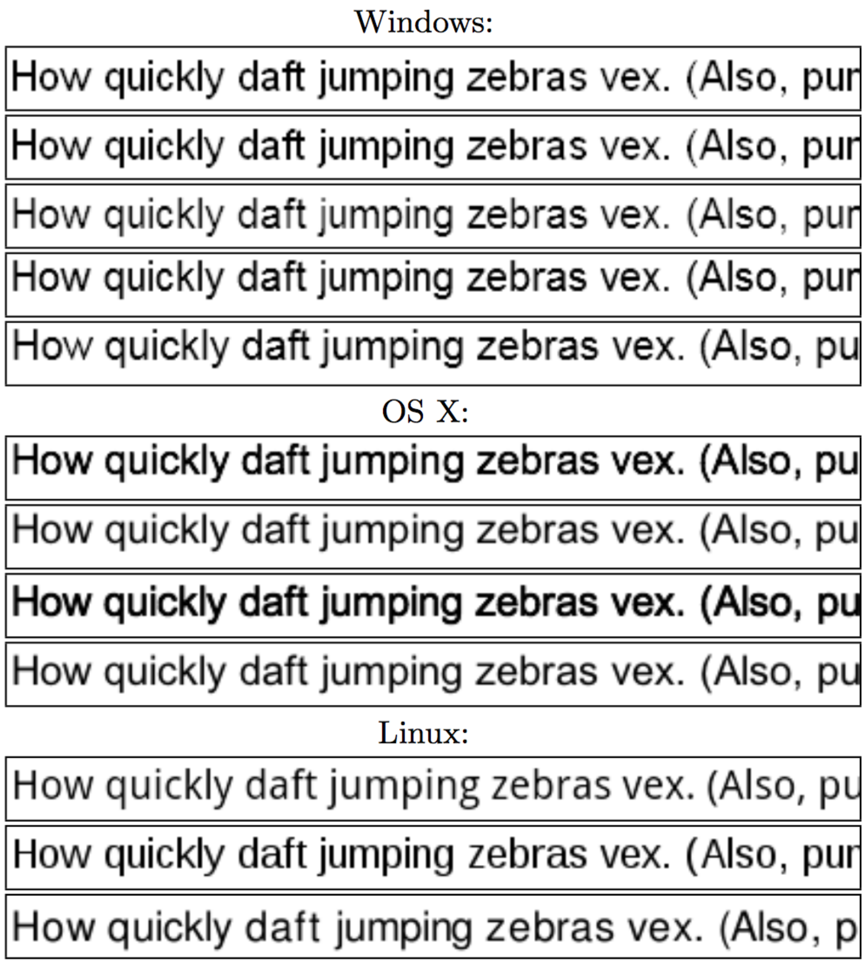

Windows:
How quickly daft jumping zebras vex. (Also, pur
How quickly daft jumping zebras vex. (Also, pur
How quickly daft jumping zebras vex. (Also, pur
How quickly daft jumping zebras vex. (Also, pur
How quickly daft jumping zebras vex. (Also, pu
OS X:
How quickly daft jumping zebras vex. (Also, pu
How quickly daft jumping zebras vex. (Also, pu
How quickly daft jumping zebras vex. (Also, pu
How quickly daft jumping zebras vex. (Also, pu
Linux:
How quickly daft jumping zebras vex. (Also, pu
How quickly daft jumping zebras vex. (Also, pur
How quickly daft jumping zebras vex. (Also, p

Figure 3.1: Mowery & Shacham (2012) discovered that it was possible to see differences in the text rendering. Here do we see how 3 different operating systems render 20pt Arial in 13 different ways.

# Chapter 4

# Machine Learning

Machine learning is a well known field of study in Computer Science. The exact definition, how research is done in this field and several relevant techniques for our research will be discussed in this chapter.

## 4.1 What is Machine Learning

Sometimes we are not able to handwrite a program for a complex problem as for example detecting malicious credit card transaction. There are so many small rules that we need to combine and tune that it is almost impossible to have a complete reliable program. Also do we need to update the program regularly over time, because fraude is a moving target. As a solution people came up with machine learning. Instead of manually write a task-specific program, we try to collect as much concrete examples of our task and let the machine learning algorithm figure out how to execute the required task. In case of our example with the credit card fraud detection, we would collect examples of fraudulent and legitimate transactions and give that to the machine learning algorithm.

If we look at the literature, there exists no clear formal definition about machine learning. Although many people have tried it, for example Samuel (1959) defined machine learning as:

*Field of study that gives computers the ability to learn without being explicitly programmed.*

Later defined Mitchell (1997) a well-posed learning problem as

*A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E.*

Let's explain this with an example, for instance checkers. The experience E can be described as the input, received by the computer program. With checkers the experience is equal to the data of thousands of games, which includes every move of the player and opponent of each game. The task T of the computer program is playing checker. The ability to play checker gets evaluated based on the performance measure P. In our case the goal is to win the checkers game, so the performance P is winning or losing the game. Based on the performance, the computer program can learn from the data and evaluate which moves are good and which are bad during a game. This can be done, based on a probability and a score function. The score function measures the accuracy of the predictions. Every prediction gets rated by the score function. The higher the score, the higher the accuracy of the prediction. During the learning, the computer program will take the prediction with the highest score to improve its chances on winning the game. By taking the prediction with the highest score, the computer program takes the most accurate prediction. Eventually if we update after every move, selecting every time the move with th highest chance on victory, the program will learn itself to play checkers. Mitchell (1997) has defined this example more formal as:

*A computer program that learns to play checkers might improve its performance as measured by its ability to win at the class of tasks involving playing checkers games, through experience obtained by playing games against itself.*

In general we can describe machine learning as the field of study that does research and development towards self-learning algorithms. Overall is research in machine learning done in 3 steps: datacollection, pre-processing and the learning.
Within the field of machine learning, we make in general three distinction in the types of learning tasks (Russell & Norvig, 1995), namely supervised learning, unsupervised learning and reinforcement learning. In this research we will discuss the group of supervised and unsupervised learning. These groups contain algorithms which we will use during our research.

## 4.2 Workflow of a Machine Learning Experiment

Now that we have a better understanding of what machine learning exactly is, we will continue with the workflow of a general machine learning experiment, which is the methodology we followed for our research. A general machine learning experiment exist out of three important phases: the data gathering, the pre-processing and the learning phase.

The first phase is the data gathering. This may seem trivial but this is not the case. Before we even start to collect data we need be conscience on what we want to collect and on what to do with it. For example 'How do we label the data?', 'Are we able to label the data?' and 'Where do we get this data from?' are all important questions we need to ask ourself during this phase. Note that, the collected data is a representation of the problem we want to learn. If the data is collected or labeled bad, it influences the whole experiment. Often it is the case at the start of a machine learning project that the data is already given, but for our research this was not the case. So caution is needed.
Once we have collected the data, we continue with the pre-processing phase. In this phase we try to represent the data as efficient as possible. By doing this, we try to optimise the performance of our self-learning algorithm. Crone et al. (2006) has shown that there is a significant impact of pre-processing on datamining. In section 7.9.2, we will discuss Principle Component Analysis, a technique that allows us to extract the most informative features of a dataset.
At last we have the learning phase, where why try to learn a certain problem, for example playing checkers. In this phase, we feed the pre-processed data to our algorithm and the algorithm tries to learn from this data and solve the problem. Machine learning tasks and there algorithms are divided in three broad categories, based on the type of 'signal' or 'feedback' that is available. As mentioned previously, these categories are reinforcement learning, supervised learning and unsupervised learning and we focus only on the latter two.

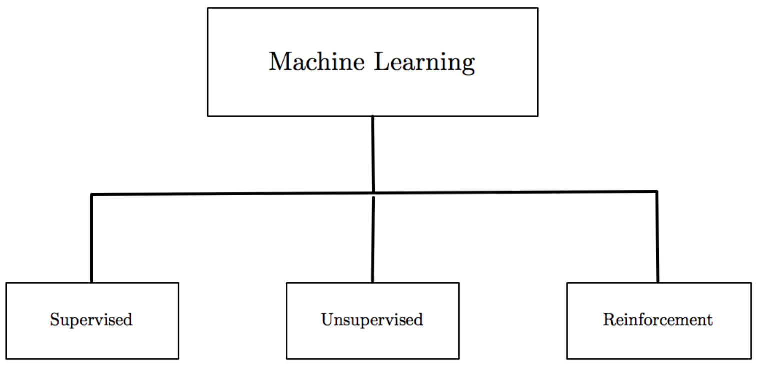


Figure 4.1: Schema of the different categories in the type of tasks in machine learning

## 4.3 Supervised Learning

Most of the time we have a dataset available in machine learning, with examples of the concept we want the algorithm to learn. We speak of supervised learning, when an algorithm is presented with examples of the concept, namely the given inputs and the desired outputs, and it tries to learn a general rule for the mapping of the inputs to the outputs. We also call this 'general rule' the hypothesis of an algorithm. With this hypothesis, for example $y = f(x_1, x_2, ..., x_d)$, the algorithm can predict the output $y$ for the given input features $x_i$, even if the algorithm has never encountered the input. The goal is to achieve the highest possible prediction accuracy. Although we must be careful with this ambition, because we can encounter overfitting or the opposite underfitting. We will discuss this later in more detail.
Under Supervised learning we group several types of problems such as a classification problem and a regression problem. A classification problem is a problem where we try to construct a hypothesis for a small set of possible output values $y$. For example a spamfilter, with spam and non-spam as possible output.
Another type of problem is a regression problem. In contrast to a classification problem, we construct for a regression problem a hypothesis for the output value $y$, where $y$ is part of a very big or infinite set of output values. An example of such type of problem is the prediction of the rent, based on the surface.
In the next section we continue by focussing on the classification problem. This is the problem we need to solve during our research.

### 4.3.1 Classification Problem

A classification problem is one of the well-known type of problem in supervised learning. When we have to construct a hypothesis, where the possible set of output values are rather small, we speak of a classification problem. As an example we have a spamfilter where spam and non-spam are the possible output values. If we try to describe the example by the definition of Mitchell (1997) is experience E the dataset $v$ with labeled emails. The task T of the spamfilter is to decide which email must be labeled as spam and which as non-spam. The performance P will be decided based on the accuracy of the spamfilter. This is the precision of correctly labeled emails.
One of the many supervised learning classification algorithm is K-Nearest Neigbors, which will be discussed in next section.

### 4.3.2 K-Nearest Neighbors

The K-Nearest Neighbors algorithm, also know as the K-NN algorithm, is an algorithm used for classification in supervised learning. It is an algorithm that uses previous trained instances from the memory to decide on new instances. This kind of learning is also called instance-based learning. The K-NN algorithm is one of simplest algorithms in machine learning (Mitchell, 1997) and is very intuitive. The algorithm takes $n$ input values and corresponds these to points in a $n$-dimensional space $\mathbb{R}^n$. It assigns a value to a new instance by looking at the most common value amongst the k nearest neigbors of that new instance. Notice that the K-NN algorithm is a non-parametric algorithm, which means it is making no assumptions about the functional form of the problem being solved.
To find the k nearest neighbors of an instance, a certain distance measure is used. For the distance measure there are a lot of options. The Euclidean distance measure is one of the most popular distance measure. More precisely, let us describe an arbitrary instance $x$ as

$$\langle a_1(x), a_2(x), ..., a_n(x) \rangle$$

with $a_r(x)$ the $r$th attribute of instance $x$.

The Euclidean distance $d(x_j, x_j)$ of the instances $x_i$ and $x_j$ can then be described as

$$d(x_i, x_j) = \sqrt{\sum_{r=1}^{n} (a_r(x_i) - a_r(x_j))^2}$$

Although, the Euclidean distance is the most popular, other distance measure can be used too. Such as the hamming distance, where we calculate the distance between the binary vectors. For example the hamming distance between the binary vectors 1011101 and 1001001 is 2. Another distance measure is the Manhatten distance. This is a distance measure equal to the sum of the absolute differences of their Cartesian coordinates.

$$d(x_i, x_j) = \sum_{r=1}^{n} |a_r(x_i) - a_r(x_j|$$

At last we have also the Minkowski distance, which is a generalisation of the Euclidian distance and the Manhatten distance.

$$d(x_i, x_j) = \left( \sum_{r=1}^{n} |a_r(x_i) - a_r(x_j)|^p \right)^{\frac{1}{p}}$$

Overall the Minowski distance is used in most K-NN algorithms. Note that, if we have $P = 1$ we get the Manhatten distance and with $P = 2$ the Euclidian distance.

## 4.4 Unsupervised learning

Unsupervised learning is another category within the machine learning. With supervised learning in 4.3, we saw that we have labeled data containing examples of the mapping between the input values and the output values. With unsupervised learning this is not the case, where the algorithm has to figure out the mapping from the input values to the output values by its own. The algorithm has no extra information in the training set and has to try to construct a hypothesis in another way. An example of an unsupervised learning technique is clustering, where the algorithm tries to group the data. An example of clustering is shown below.

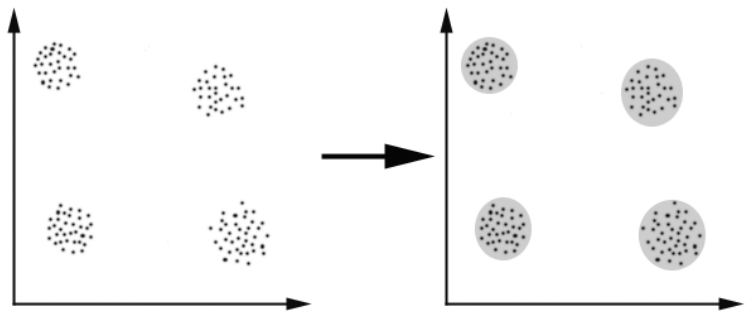


Figure 4.2: Identificatie bij clustering (Source: `http://home.deib.polimi.it`)

One of the unsupervised learning techniques we will use during our research are autoencoders. autoencoders are used for deep learning and are a special configuration of an artificial neural network. More about autoencoders will follow in section 6.3.

## 4.5 Underfitting and Overfitting

In section 4.3, we briefly mentioned something about the fact that during the training of a classifier, we must be careful with the ambition of achieving the lowest training error as possible, since we have the risk of overfitting. Mitchell (1997) defined overfitting as:

*Given a hypothesis space H , a hypothesis h in H is said to overfit the training data if there exists some alternative hypothesis $h^{'}$ in H, such that h has smaller error than $h^{'}$ over the training examples, but $h^{'}$ has a smaller error than h over the entire distribution of instances..*

Summarized, this means that we make the model too specialised on our training data and it is not able to classify other unseen data correctly. Overfitting occurs because of two problems: not enough data and noise. The trainings data can not be considered perfect and is certainly not enough to represent the real world. It is just an approximation. More specifically, if we are training our classifier until zero errors, we are including the noise in our model. As a result will the classifier perform perfectly on the trainings data, but very poor on other non-training data. The opposite of overfitting is underfitting, where our model is too general and has insufficient representational power. It will perform bad on the training- and test data. Note that, an underfitted model still can perform better on unseen data than an overfitted model. As a researcher, it is important to find a balance in the generalisation and specialisation of the model. In practice, we like to define this equilibrium as the point just before the error of the training and test data starts to increase. Figure 4.3 and 4.4 are examples of an overfitted and underfitted model on the same data. The true function is the model we try to obtain. Figure 4.5 shows the equilibrium between generalisation and specialisation of the model. This is an example of a well trained model.

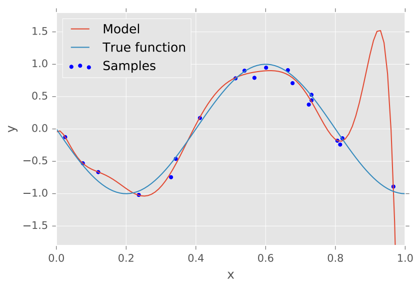


Figure 4.3: Overfitted model

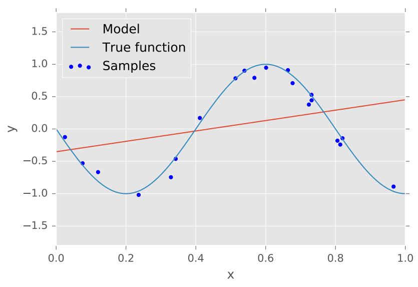


Figure 4.4: Underfitted model

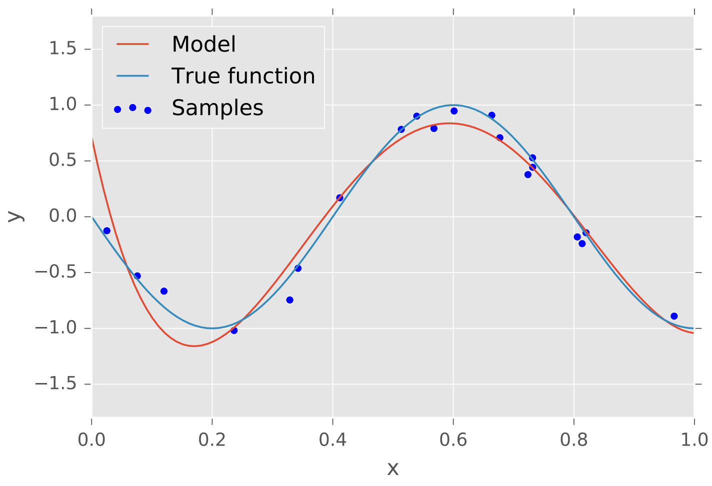


Figure 4.5: A perfectly trained model

## 4.6 Bias and Variance

Previously, we have seen that the training error of a model is not guaranteed to be the same as the generalisation error or out-of-sample error, which is the prediction error on the unseen data. In this section, we discuss how we can understand the error of a model better. We will do this by decomposing the error of a model in two components: error due *Bias* and error due *Variance*. The relation between Bias and Variance and quantifying them, allows us to better understand the error of a model. The Bias of a model is defined as the difference between the predicted values and the expected values:

$$\text{Bias}(x) = E[\hat{f}(x)] - f(x) \tag{4.1}$$

with $\hat{f}(x)$ the model and $f(x)$ the true function.

We speak of a model with a high Bias when the model is not specific enough and lacks in complexity. As a result, we get a big difference between the predicted values and the expected values. It is clear that we try to achieve a Bias as low as possible. Secondly, we have the Variance, which is quantified as the variability between the model's predictions:

$$\text{Variance}(x) = E[(\hat{f}(x) - E[\hat{f}(x)])^2] \tag{4.2}$$

The Variance tells us how much a certain point $x$ varies for different model configurations. This can we see in direct relation with overfitting, where we will see a high variance, because the model is adapting every time to the noise. As a consequence, we try also to achieve the lowest variance possible.
Now that we have defined these two components, we can bring them together to define the out-of-sample error (Friedman et al., 2001):

$$\text{Err}(x) = E[(Y - \hat{f}(x))^2] \tag{4.3}$$

If we rewrite this with the components we get:

$$\text{Err}(x) = (E[\hat{f}(x)] - f(x))^2 + E[(\hat{f}(x) - E[\hat{f}(x)])^2] + \sigma^2 \tag{4.4}$$

The last term $\sigma$ represents the noise on the data, which is irreducible.

By taking a closer look at equation 4.4, it is clear that there is a *bias-variance-tradeoff*. Once we want the decrease the bias, we increase the variance and visa versa. This can be brought in relation with what we discussed in section 4.5, regarding the balance between generalisation and specialisation of the model. Figure 4.6 illustrates this bias-variance tradeoff and allows us to define this sweet spot in terms of Bias, Variance and model complexity:

$$\frac{\partial Bias}{\partial Complexity} = -\frac{\partial Variance}{\partial Complexity} \tag{4.5}$$

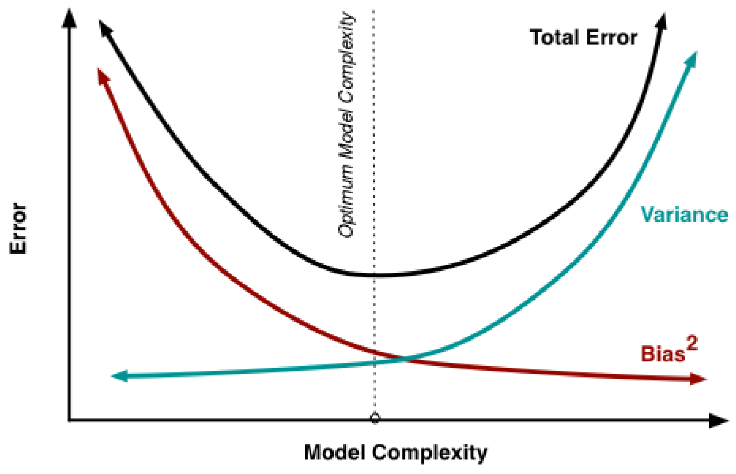


Figure 4.6: This plot visualises how bias and variance are contributing to the general error. (Source: `http://scott.fortmann-roe.com/`)

Note that, we are not able to calculate the real out-of-sample error, but always use an approximation. More on model validation will be discussed in section 4.7.

To conclude this section, we like to end with the dartboard analogy by Claude & I. (2011) in figure 4.7. This analogy is a good summary on how we could see Bias and Variance. The red bulls eye represents the target value. The goal is to have the predicted values as close as possible to the bulls-eye. As we can see, results a low bias in predicted values close to the bulls-eye, while a high bias causes the predicted values to position away from the bulls-eye. This is also what is happening with overfitting, where the model is to specialised and is eventually going away from the true function we want to learn. The effect on variance is also clearly visible. The predicted values are spread out with a high variance, while a low variance results in a concentration around a certain point. At last do we see that the left upper dashboard is the one we like to achieve. Unfortunately this is not possible, due the bias-variance-trade-off.

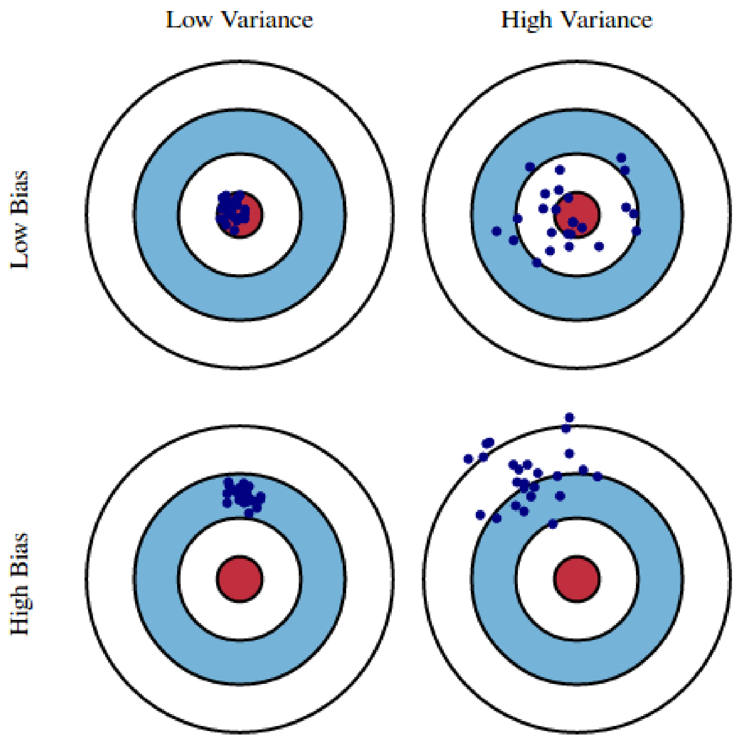


Figure 4.7: Dashboard analogy by Claude & I. (2011)

## 4.7 Model Validation

When we evaluate the performance of a trained model, we do this based on the generalisation error. Although, as earlier mentioned in section 4.6, it is not possible to calculate the generalisation error and we are only able to approximate it. How this is done, will be explained in this section. Earlier we saw that the trainings set is definitely not a good choice to estimate our generalisation error, since this is completely biased. If we would use the training set as approximation and we would try to minimise it, our model would be overfitted. Another better solution is to split

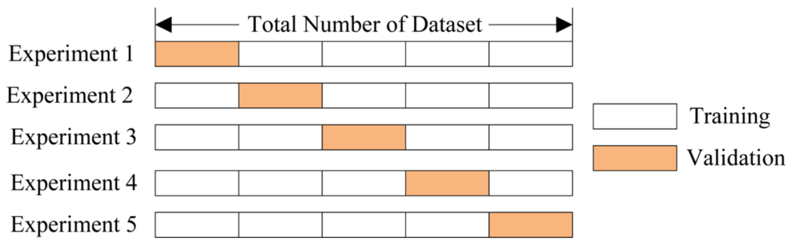


Figure 4.8: 5-fold cross-validation, illustrated by **?**

the data in a training-, validation- and test set and learn in three phases. The first phase is the training phase, where we use the training set to train our model. Afterwards we continue with the validation phase, where the validation set is used during the model selection. The error of this set will be used to tune the parameters of the model. In the last phase, we use the test set to evaluate our model. The error of the test set is considered to be a good approximation of the generalisation error. Note that, the test set is not used for tuning the model. It is very important to keep the test set separate from the validation or training set, otherwise would the approximation be too optimistic (Friedman et al., 2001).
Strictly defined rules about splitting the dataset do not exist. Empirically it is shown that the more training data you have, the better results you get. Also does the size of the validation set need to be big enough to be a good representation of the data distribution. In practice there are rules of thumb applied, for example 60%/20%/20% or 50%/25%/25% for training/validation/test. Unfortunately, our dataset is sometimes too small and the training data too valuable. Cross Validation allows us to still make a fair approximation, despite the small dataset.

### 4.7.1 K-fold Cross Validation

K-fold Cross Validation allows us to make a good approximation of the generalisation error of a model, despite a small dataset. The validation method splits the dataset in K parts and takes each iteration $i$, part $i$ as a validation set and the other K-1 parts as training set. K-fold cross validation repeats this in total K times. After K iterations, our prediction error is defined by the cross-validation error:

$$CV(\hat{f}) = \frac{1}{N}\sum_{i=1}^{N} CV_i \tag{4.6}$$

A disadvantage of K-fold cross-validation is the increase in computational time. The bigger K, the longer the computational time, since we need to repeat the learning K times. Nevertheless, we must be careful when choosing the value K, because it can have an influence on the cross-validation error. If we take K too small, we may end up with a high bias, since the chunks are too big and the training data is too small. On the other hand, using a big K would likely result in a high variance. In general are 5-fold and 10-fold cross validation suggested (Kohavi et al., 1995).

## 4.8 Principle Component Analysis

Principle Component Analysis (PCA) is a statistical procedure, which allows us to extract the most informative features or principle components of a dataset. This is done by looking for the directions with the highest variance. The general idea behind it, is that in those directions the data varies a lot and gives us the most useful information. Mathematically this is done by calculating the eigenvectors with the highest eigenvalues. An eigenvector can be seen as a direction and its corresponding eigenvalue as the variance for that direction. In machine learning, PCA is commonly used for dimensionality reduction. Formally, this can be done as follows: Suppose, we want to reduce an $n$-dimensional dataset to a $k$-dimensional dataset. The first step of the reduction process is calculating the covariance matrix $\Sigma$:

$$\Sigma = \frac{1}{m}\sum_{i=1}^{m}(x^{(i)})(x^{(i)})^T \tag{4.7}$$

with $x^{(1)}, x^{(2)}, ..., x^{(m)}$ the samples of the dataset.

The next step is the calculation of the eigenvectors. This can be done with *Singular Value Decomposition*, a operation from the linear algebra. With this technique, we factorize a real or complex matrix $M$ in:

$$M = U\Sigma V^T \tag{4.8}$$

With $U$ and $V$ the left- and right singular values of $M$ and $\Sigma$ a rectangular matrix with on its diagonal the eigenvalues of $M$.

By applying the Singular Value Decomposition on $\Sigma$ (from equation 4.7), taking the $k$ first columns of $U$ and multiplying these with sample $x^{(i)}$, we get as a result the dimensional reduced sample $x_k^{(i)}$. We will not go deeper into the mathematics of PCA.

Overall it is important to know that PCA can be used for dimensionality reduction and this can be done mathematically by calculating the covariance matrix and use Singular Value Decomposition.

In this chapter, we discussed machine learning in general and explained the K-Nearest Neighbors algorithm. In the next chapter we continue with another machine learning algorithm, called Artificial Neural Networks.

Chapter 5

# Artificial Neural Networks

The concept of artificial neural networks is inspired by the biological brain. The biological brain is built of complex webs of interconnected neurons, where each neuron receives input from other neurons and the effect of each input line of a neuron is controlled by synaptic weights. These synaptic weights are also adapting and give the brain the ability to learn something. This biological structure is also visible in artificial neural networks. Note that, an artificial neural network is not an artificial copy of the human brain but is inspired by. For example the communication between the neurons is completely different. Where an individual artificial neural outputs a single constant value, outputs a biological neuron a complex time series of spikes. Especially the ability of the human brain to process visual complex content in less than a second, encouraged researchers to use the human brain as an inspiration for a learning algorithm.
The switching speed of a neuron in the human brain is known to be around the order of $10^{-3}$ seconds, which is rather slow compared to the computer switching speed of $10^{-10}$ seconds. Surprisingly, the human brain is capable to visually recognise for example our mother in approximately $10^{-1}$ seconds. A very impressive result if you take the complexity of the problem and the switching speed of the biological neuron in account. A 100 inference steps to make such a complex decision doesn't seem enough. Based on these observation, researchers were convinced that the impressive processing abilities of the biological neural network was the result of highly parallel computations on the data, distributed over different neurons. This is also reflected in the artificial neural network.
In general, we define an artificial neural network as a learning method that provides a general and practical method to learn real-valued, discrete-valued and vector-valued functions from examples. Artificial neural networks are very useful and effective for certain types of problems, especially the type of problems where we need to interpret real-world complex sensor-data (Mitchell, 1997). For example the learning of recognising handwriting (Y. LeCun et al., 1989), speech (Lang et al., 1990) and face recognition (Cottrell, 1990) are a few of the many problems, where artificial neural networks have achieved great results and emphasize the effectiveness of this learning method on complex real-world sensor-data.

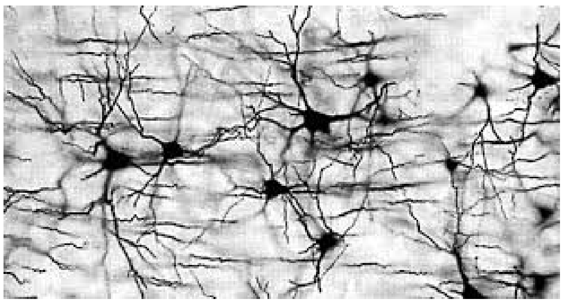


Figure 5.1: A natural neural network (Source:`http://www.alanturing.net`)

## 5.1 Building Blocks

An artificial network can be decomposed in different buildings blocks. In general, an artificial neural network consists of units, which can be grouped in layers. A unit is similar to a biological neuron. Each unit can have direct connections with other units. Also, each incoming connection has an associated weight. This real-value determines the contribution of the input signal to the output of the unit. By tuning the weights based on the performance of the neural network and favoring certain input signals, the artificial neural network is able to solve certain complex problems.
If we look at it from a higher level, we can group these units in layers. There exist three types of layers: an input layer, a hidden layer and an output layer. The input layer consists of the units that receive the input data for the neural network and has only outgoing connections. The output layer consists of units with only incoming connections and delivers the output of the artificial neural network. At last we have the hidden layer, this layer consists of units with incoming and outgoing connections. Figure 5.1 illustrates all the components of an artificial neural network. The displayed network is also called a *fully-connected* neural network. This means that all units from one layer are connected with all the units from the next layer.

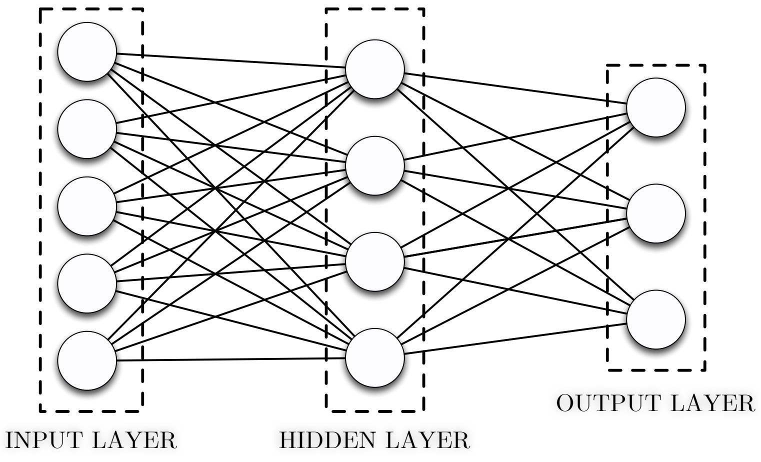


Figure 5.2: A fully-connected neural network with one hidden layer

As we said before, a unit outputs a single constant value. Although, we do see in figure 5.1, that it has multiple input signals. Well, a unit achieves this by applying a propagation function, often the weighted sum, on the input signals and transforming it to one constant value with an activation function. Every unit from a non-input layer has such an activation function. More on activation functions can be found in the next section.

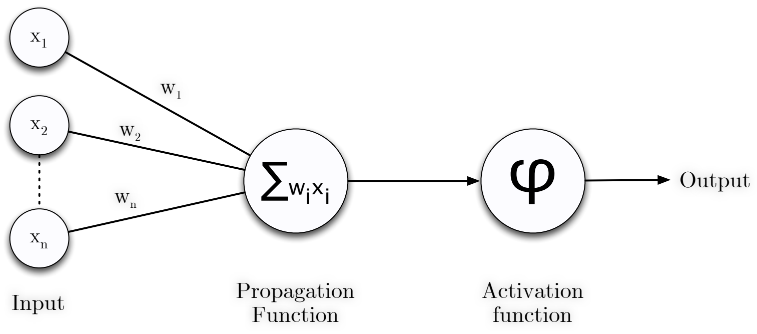


Figure 5.3: A unit with as propagation function the weighted sum

## 5.2 Activation Functions

An activation function is a function that transforms the multiple input signals of a unit to one constant output signal. This is the reason why the output of a unit is also called 'the activation'. As an equation, we write the activation function $\phi$ as:

$$o = \phi(\overrightarrow{w} \cdot \overrightarrow{x}) \tag{5.1}$$

with $o$ the output.

An activation function can be linear or non-linear, but mostly a non-linear version is used. The non-linear activation functions introduce non-linearity and allow the network to solve complex problems, which are linear non separable (Y. A. LeCun et al., 2012). Popular non-linear activation functions are the Sigmoid, Hyperbolic Tangent and the Rectified linear unit (ReLU). We will discuss these in the following subsections, but first we talk about a special type of unit, called a perceptron.

### 5.2.1 Perceptron

The perceptron is a special kind of unit designed by Rosenblatt (1958). This kind of unit transforms the input values to only two possible output values, namely 1 and -1. It calculates the weighted sum of the input values and if this sum is bigger than a certain threshold the output value of that unit is 1 and otherwise -1. Formally we can write this down as:

$$o(\overrightarrow{x}) = \begin{cases} 1 & if \overrightarrow{w} \cdot \overrightarrow{x} > 0 \\ -1 & otherwise \end{cases} \tag{5.2}$$

where $\overrightarrow{x}$ is a vector with the input signals and $\overrightarrow{w}$ the corresponding vector with the weights for the input signals.

The behaviour of a perceptron results in the creating of a hyperplane decision surface, which deviated the input signals in two groups. Unfortunately, not every output of the perceptron is differentiable and for certain optimisation algorithms this is a requirement. Take for instance the backpropagation algorithm in section 5.3. Luckily, we can solve this by using a non-linear activation function, such as the Sigmoid.

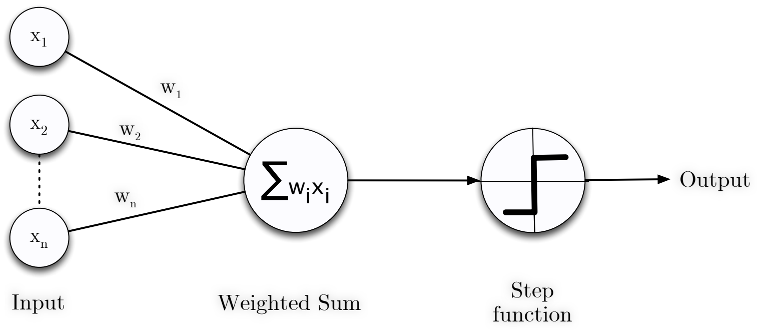


Figure 5.4: A Perceptron

### 5.2.2 Sigmoid

The Sigmoid function is a non-linear activation function and is differentiable for the whole output domain. The output value range of the sigmoid function is from 0 to 1. The difference of the sigmoid with the perceptron lies in the smooth transition when going to the extremes. This smooth transition makes the whole output domain differentiable and allows us to use optimisation algorithms, which require differentiability. Also, if the sigmoid is used as activation function for the units, we are able to process non-linear data, because of the non-linearity of the function. Formally we write the sigmoid function (denoted $\sigma(x)$) as follows:

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{5.3}$$

In general, it is known that an artificial neural network may learn faster when the units use an antisymmetric sigmoid activation function instead of a symmetric sigmoid activation function (Haykin, 2004). An activation function $\varphi(x)$ is antisymmetric when:

$$\varphi(-x) = -\varphi(x) \tag{5.4}$$

As you can see in figure 5.2.2 is the sigmoid function symmetric, but with small modifications we can make it antisymmetric. The Hyberbolic Tangent is such a antisymmetric variation of the sigmoid function.

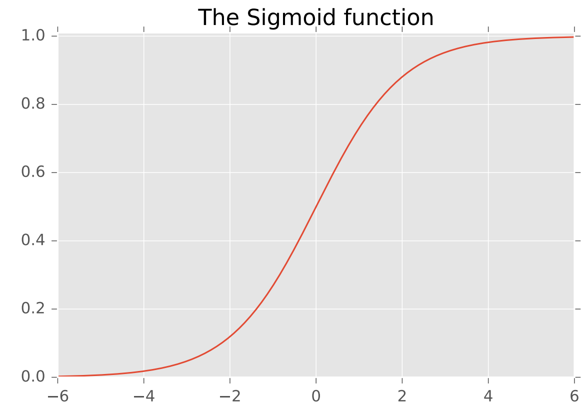


Figure 5.5: The Sigmoid Function

### 5.2.3 Hyperbolic Tangent

the Hyperbolic Tangent (denoted $htan(x)$) is a non-linear activation function and is a shifted and stretched version of the sigmoid function. The activation function is defined by:

$$\varphi(v) = a\varphi(bx) \tag{5.5}$$

with a en b two constants. Suitable values for a en b can be found in Y. LeCun et al. (1989). If we take for simplicity the $a = 1$ and $b = 1$, we derive:

$$\varphi(x) = htan(x) = \frac{1}{1 + e^{-2x}} + 1 \tag{5.6}$$

The following figure illustrates this Hyperbolic Tangent function, where the output is in the range of -1 and 1.

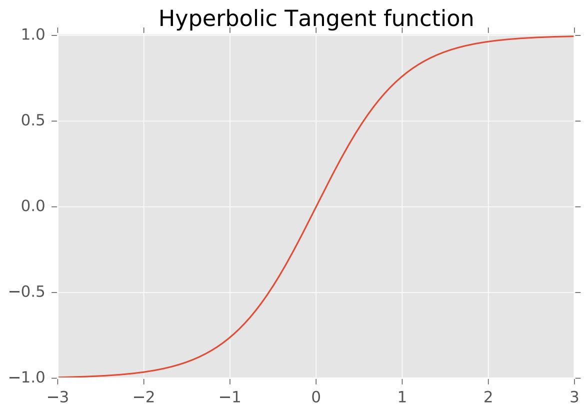


Figure 5.6: The Hyperbolic Tangent Function

### 5.2.4 Rectifier

At last, we have a Rectifier. A unit with a rectifier as activation function is also known as a Rectified Linear Unit or RELU. A basic rectifier can be defined by:

$$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases} \tag{5.7}$$

A rectifier is, as from 2015, the most popular activation function used in deep network learning (Y. LeCun et al., 2015). The main reason is the research by Glorot et al. (2011), where it is shown that supervised training of very deep neural networks is much faster if the hidden layers are composed of ReLU's, instead of a units with an activation function as the sigmoid or Hyperbolic Tangent. This discovery was really a breakthrough for deep network learning. The research showed that RELU's allowed us to do purely supervised learning with deep learning networks and achieved good results, where before it was deemed that for unsupervised learning pre-training was needed (Larochelle et al., 2007). Another advantage of RELU's is the introduction of sparsity, which is naturally done by the RELU's by producing zeroes for every input value less or equal to zero. Sparsity is really a concept of interest in machine learning. Especially because it closes the gap between artificial and biological neural networks. Studies have shown that the information encoding in a biological brain happens in a sparse and distributed way (Attwell & Laughlin, 2001). Only 1 to 4 % of the neurons are active at the same time (Lennie, 2003). Sparsity has already been the key for research towards deep learning networks (Lee et al., 2008; Ranzato et al., 2006).

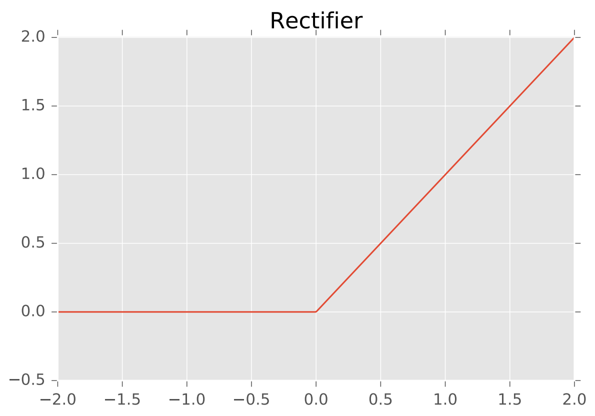


Figure 5.7: The activation function of a Rectifier Linear Unit

## 5.3 Gradient Descent and Backpropagation

In this section we talk about Gradient Descent and the Backpropagation algorithm. Gradient Descent is a general approach to minimise the error $E$ of a learning algorithm. The Backpropagation algorithm is a gradient descent algorithm that is commonly used to update the weights and to minimise the errors of an artificial neural network.
When we want to minimise the error of a learning algorithm, we see the error as a surface.

The surface, also called error surface, is the look of an algorithm's error $E$, in function of its parameters. When we are using gradient descent, must the error surface be differentiable. This means that every point on the surface, corresponding with a parameter configuration, must be differentiable. Visually this means that there can be no gap in the error surface. Figure 5.3 is an example of a differentiable error surface.
Despite the requirement of differentiability, we still see a bad performance of gradient descent on an irregular error surface. The irregularity results in a lot of local minima in the error surface, which causes the gradient sign to change often. Further is finding the global minimum not assured with gradient descent.

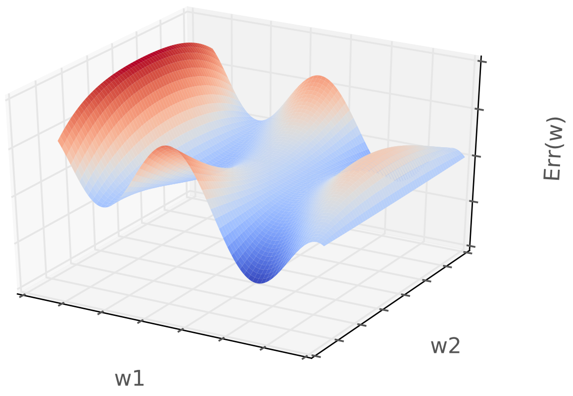


Figure 5.8: An example of an irregular error surface

### 5.3.1 Gradient Descent

Gradient descent or steepest descent is an optimisation algorithm, that works very intuitively. The gradient descent algorithm starts at a random point on the surface and moves across by following the path with the steepest slope. In the case of a learning algorithm's error, where we try to find a minimum, gradient descent will follow the path with the steepest slope downwards. Note that, every point on the surface is a different configuration of parameters (or weights) for the concerning learning algorithm. So every time we are following the steepest slope, we are adjusting the parameters of the learning algorithm. Because gradient descent is always following the steepest slope, we call it a greedy algorithm.
As stated before, gradient descent requires differentiability. This means that the algorithm's error $E$ must be derivable for the whole domain. Formally we can write the derivative or gradient of error $E$ in function of its weights $\overrightarrow{w}$ and with length $n$ as follows:

$$\nabla E(\overrightarrow{w}) = \left(\frac{\partial E}{\partial w_0}, \frac{\partial E}{\partial w_1}, \ldots, \frac{\partial E}{\partial w_n},\right) \tag{5.8}$$

If we take a closer look to the derivative $\nabla E(\overrightarrow{w})$, we see that $\nabla E(\overrightarrow{w})$ is a vector describing a size and direction and represents the steepest slope. This is a very powerful property. If we want to know the steepest slope for any point in the domain, we only need to calculate the derivative. Note that, this gradient describes the steepest slope *increasing* E, in function of the parameters

$\overrightarrow{w}$. We want to minimise the error, so we want the steepest slope to *decrease* $E$. This can be done by taking the negated gradient $-\nabla E(\overrightarrow{w})$. Now, the weight update or training rule can be defined as follows:

$$\overrightarrow{w} \leftarrow \overrightarrow{w} + \nabla \overrightarrow{w}$$

with

$$\Delta \overrightarrow{w} = -\eta \nabla E(\overrightarrow{w}) \tag{5.9}$$

In the equation, we do not subtract the complete gradient but we take a certain rate of the gradient. This rate $\eta$ represents the learning rate. The learning rate is used to tune the step size of the algorithm and varies between the value 0 and 1. The choice of the learning rate has an impact on the performance of gradient descent. Smaller values for $\eta$, let the algorithm converge much slower, while larger values introduce the risk of overstepping a local minimum. It is also know for gradient descent to 'zigzag' in convex valleys (Mitchell, 1997). As a solution, we can introduce an adaptive learning rate, which is changing over time. By doing this we have the best of both worlds, bigger step sizes in the beginning prevent a slow convergence and the smaller steps makes it less susceptible to overstepping or zigzagging.
Next, we want to describe gradient descent as a practical algorithm. This means we need to find an efficient way to calculate the gradient for each step and iteratively update the weights. Fortunately, this seems not too difficult. To show this, we must first define the algorithm's error $E$. During the training of the learning algorithm, we can use the training error as an indication on how the algorithm is performing. There exist a lot of error measures to define the learning algorithm's error $E$ w.r.t $\overrightarrow{w}$, we use the following:

$$E(\overrightarrow{w}) \equiv \frac{1}{2} \sum_{d \in D} (y_d - \hat{y}_d))^2 \tag{5.10}$$

With $D$ the dataset, $\hat{y}_d$ the predicted value for $d$, $y_d$ the expected value for $d$. The reason why we choose this equation will follow later.

Now that we have the error $E$, we can obtain the vector of $\frac{\partial E}{\partial w_i}$ derivatives by differentiating $E$ from equation 5.10. For the simplicity of the algebraic derivation we take for the learning algorithm $f$ the linear unit:

$$f(\overrightarrow{x}; \overrightarrow{w}) = \overrightarrow{x} \cdot \overrightarrow{w}$$

We could take a more sophisticated $f$, but this would make the algebraic derivation and calculations more complex and for this example a linear unit is sufficient. Eventually, we describe the partial derivative by:

$$\begin{aligned} \frac{\partial E}{\partial w_i} &= \frac{\partial}{\partial w_i} \frac{1}{2} \sum_{d \in D} (y_d - \hat{y}_d)^2 \\ &= \sum_{d \in D} (y_d - \hat{y}_d) \frac{\partial}{\partial w_i} (y_d - \overrightarrow{w} \cdot \overrightarrow{x_d}) \\ &= \sum_{d \in D} (y_d - \hat{y}_d)(-x_{id}) \end{aligned} \tag{5.11}$$

Where $x_{id}$ denotes the single input component $x_i$ for training example $d$.

Note that, the choice of $E$ in equation 5.10 and the linear unit $f$makes the calculation very convenient. For completion, we rewrite the update rule for a single component, based on equation 5.9, as:

$$\Delta w_i = \eta \sum_{d \in D} (y_d - \hat{y}_d) x_{id} \tag{5.12}$$

In summary, we describe the gradient descent algorithm for the linear unit by first initialising the weights randomly. Then for each iteration we update the weights by applying the learning algorithm on every training example and compute $\Delta w_i$ for each weight, according to equation 5.12. This process is repeated until the termination condition is met, for example a certain amount of passes of the training set or a sufficiently small error value. Because we update all the weights with respect to the whole training set, this vanilla version of the gradient descent algorithm is also known as *Batch Gradient Descent*. A big disadvantage of this implementation is the inefficiency for very large datasets, where it can take a long time before the weights are updated. This is the reason why sometimes *Stochastic Gradient Descent* is preferred, an incremental approach of gradient descent.

### 5.3.2 Stochastic Gradient Descent

Stochastic Gradient Descent or Incremental Gradient Descent is a common variation of Batch Gradient Descent. Instead of updating the weights with respect to the whole training set, we update the weights every time with respect to a single training example. As stated earlier, the standard gradient descent has some difficulties. First of all the the slow convergence towards the local minimum and secondly finding the global minimum is not assured. Stochastic Gradient descent is known to alleviate these difficulties (Mitchell, 1997).

Now that we have seen gradient descent as a optimisation method and know how to minimise the error of a function. We continue by discussing the actual learning method of an artificial neural network, which is done by using backpropagation.

## 5.4 Backpropagation

The backpropagation algorithm or just backprogation is a learning method, used to train an artificial neural network. It is in general considered as a supervised learning method, although it can be used sometimes in a unsupervised way, for example in section 6.3 with autoencoders. In section 5.1, we already mentioned that learning of an artificial neural network, happens by tuning the weights. The backpropagation algorithm will try to tune these weights in function of minimising the error of the artificial neural network. The tuning itself happens by using gradient descent. The measurement for the error of an artificial neural network is defined as the squared error of the difference between the output and and target values. Formally written down as:

$$E(\overrightarrow{w}) \equiv \frac{1}{2} \sum_{(x_j, y_j) \in D} \sum_{k \in outputs} (y_{kj} - \hat{y}_{kj})^2 \tag{5.13}$$

where $\hat{y}_{kj}$ is the predicted output value by the $kth$ output unit for input value $x_j$ and $y_{kj}$ is the target value for unit k.

Because backpropagation is using gradient descent, differentiability is also required (see section 5.3.1). This means that all units must use an activation function, which is differentiable for the whole output domain. Backpropagation does not work with perceptrons for example. Note that, there exist several types of neural networks, the backpropagation algorithm we discuss in this section is applicable on a **feedforward neural network**. This is an artificial neural network, where the data moves from the input layer through the hidden layer(s) to the output layer. Other types are for example recurrent neural networks and radial basis function networks, but these will not be discussed.

The goal of backpropagation is to minimise the total error of the network, defined by equation 5.13. To achieve this, we need to update each weight $w_{ji}$ in a way that the predicted output value of unit $j$ is closer to the target value. $w_{ji}$ presents the weight of the signal, going from unit $i$ to unit $j$. Earlier in section 5.3.1, we have seen that this can be done with gradient descent. So for every weight-update, we need to calculate the partial derivative $\frac{\partial E}{\partial w_j i}$ and use the negative gradient. We use $\partial_n$ to describe all partial derivatives w.r.t the error of unit n. Pay attention, $\partial_n$ is different for outer units and hidden units. The reason is that the output of an hidden unit has an impact on the output unit and thus on the error. The $\partial_k$ for outer unit $k$ is defined by:

$$\partial_k = \psi(o_k)(y_k - \hat{y}_k) \tag{5.14}$$

Where $\psi$ is the derivative of activation function $\phi$ and $(y_k - \hat{y}_k)$ the difference between the target value and predict value of unit $k$.

The $\partial_h$ for hidden unit $i$ is defined by:

$$\partial_h = \psi(o_i) \sum_{k \in outputs} w_{ki} \partial_k \tag{5.15}$$

where $o_i$ is the output of hidden unit $i$ and $w_{ki}$ the weight of the signal, going from unit $i$ to $j$. Note that, the $\partial_h$ defined in equation 5.15, is the formula for the unit right before the output unit. It is easy to extend this equation and generalise it for other units in deeper layers. We will not go deeper into these equations. Mainly, it is important to understand that there is a difference in the delta calculation for a hidden unit and an outer unit.

Now, we have everything in place to describe the backpropagation algorithm. The first step of the algorithm is randomly initialising the weights of the neural network with small zero-centered numbers for example between -0,5 and 0,5. Next we iteratively propagate each training example $x_j$ through the neural network and calculate the predicted value $\hat{y}_j$. Afterwards we check how far each unit is off of the target value by calculated the delta's. Once the delta's are calculated, we can update the weights of the network. This is repeated until a certain termination condition is met, for example an amount of iteration. The propagation of the weights through the network is also called **the forward pass** and the calculations of the delta's **the backward pass**.

We update the weights by following equation 5.9 of gradient descent (See section 5.3.1). Only we define $\Delta w_{ji}$ slightly different by:

$$\Delta w_{ji} = \eta \partial_j x_{ji} \tag{5.16}$$

where $\eta$ is the learning rate, $x_{ji}$ the training example and $w_{ji}$ the weight of the signal, going from unit $i$ to unit $j$.

As with gradient descent, we have also to possibility to update the weights after propagating the whole dataset and use the sum of all $\Delta w_{ji}$'s to update the weights. This is also known as Batch backpropagation. Although, this is unpopular in practice, because of the long learning time when having a large training set.

## 5.5 Other Gradient Descent Variants

To summarize, we have seen the general outline of a neural network, the optimisation method gradient descent and the gradient descent learning algorithm backpropagation. Previously, we stated that there are some important challenges with gradient descent. First of all, we are not assured that we will reach the global minimum. It is possible that the algorithm gets stuck in a local minimum. Another challenge is the choice of the learning rate. In section 5.3.1, we briefly discussed the issue with the learning rate, where a big learning rate value causes overstepping and possible zigzaging near the local minimum and suggested the solution of an adaptive learning rate. This might help, but we have still the issue of predefining the adaptation schedule. The schedule is not dependent on the characteristics of the dataset and must be set manually. Moreover, it would be more interesting to have a different learning rate for every parameter, instead of for all parameters the same.
In the following subsections, we will discuss four Gradient Descent variant, which will all try to tackle the above mentioned challenges.

### 5.5.1 Momentum

Adding momentum is a common extension of Gradient Descent and is believed to help against local minima. The idea of momentum is analogous of seeing the gradient descent as a ball on the error surface. Once you put the ball on the error surface, it tends, just like gradient descent, to roll or move with the greatest descent downwards. Once the ball starts rolling, it gets some momentum and wants to keep rolling in the same direction. This allows to ball to sometimes escape certain local minima and this is exactly the idea we try to achieve. By taking in account the previous weight updates and give them some momentum, we hope to escape certain local minima. If we look at equation 5.9, we can add momentum by changing $\Delta w_{ji}$ to:

$$\Delta w_{ji}(t+1) = \eta \delta_j x_{ji} + \alpha \Delta w_{ji}(t) \tag{5.17}$$

with $\alpha$ the momentum.

### 5.5.2 Adagrad

Adaptive gradient or Adagrad, is another variant of Gradient Decent. It has an adaptive learning rate, depending on the importance of the parameter or weight. The learning rate adapts itself based on how sparse the feature is. If the parameter is very sparse, it is considered to be from great interest and the learning rate will be big. If the parameter is not sparse, the learning rate will be small. As a result, Adagrad performs much better on sparse datasets than other non-adaptive gradient descent algorithms (Duchi et al., 2011).

Before we describe the gradient descent update rule, we set $g_{ti}$ as the shorthand for the gradient of the error of the weight $\theta_i$ at timestep $t$.

$$g_{t,i} \equiv \nabla_\theta E(\theta_i) \tag{5.18}$$

With this shorthand, we can define the weight update rule by:

$$\theta_{t+1,i} = \theta_{t,i} - \eta \cdot g_{t,i} \tag{5.19}$$

Next, we introduce a diagonal matrix $G_t, ii$ such that:

$$\theta_{t+1} = \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,ii} + \epsilon}} * g_{t,i} \tag{5.20}$$

The diagonal matrix $G_{t,ii}$ is a matrix, where each element $i, i$ corresponds to the sum of squares of the gradients w.r.t. the weight $\theta_i$ up to timestep $t$. $G_{t,ii}$ can be written down as:

$$G_{t,ii} = \sum_{\tau=1}^{t} g_{\tau,i} g_{\tau,i}^{\top} \tag{5.21}$$

At last, we can vectorize equation 5.20 and define it by:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t + \epsilon}} \odot g_t \tag{5.22}$$

with $\odot$ the element-wise matrix-vector multiplication.

Note that, the $\epsilon$ is a small non-zero value, which is added to avoid devision by zero. As a result of the adaptive gradient, we don't need to predefine a schedule for our learning rate $\eta$ anymore, which is an advantage. Although, using Adagrad has also some drawbacks since there is an accumulation of the positive gradients in the denominator. During the training, this sum gets bigger and bigger since the positive terms. Eventually, the learning rate $\eta$ becomes infinitely small and the algorithm learns nothing anymore. Also, it is still required to manually define a global learning rate $\eta$. The next gradient descent variant Adadelta, tries to resolve these issues.

### 5.5.3 Adadelta

Adadelta is another gradient descent variant with an adaptive learning rate and can be seen as an extension of Adagrad (Zeiler, 2012). We have seen in section 5.5.2 that Adagrad has two drawbacks. First of all the continuous decay of the learning rate and secondly the need to manually predefine the learning rate. Adadelta solves the issue of the continous decay of the learning rate by restricting the past gradients that are accumulated to a fixed window of size $w$. By introducing this window, we get a local estimate instead and the denominator will never accumulate to infinity. As a result, the algorithm will keep learning, even after many iterations. The fixed window is introduced by defining a decaying average of all past gradients:

$$E[g^2]_t = \gamma E[g^2]_{t-1} + (1-\gamma) g_t^2 \tag{5.23}$$

where $\gamma$ is a decay constant, for example 0.9 .

Considering this decaying average $E[g^2]_t$, the general gradient descent weight update rule (Equation 5.9) and the parameter update vector of Adagrad in Equation 5.22, we can describe the parameter update vector for Adadelta as:

$$\nabla\theta_t = -\frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} * g_t \tag{5.24}$$

Next we can replace the denominator with the shorthand of the root mean squared (RMS) error criterion of the gradient:

$$\nabla\theta_t = -\frac{\eta}{RMS[g]_t} * g_t \tag{5.25}$$

In Zeiler (2012), the authors mention that the units in equation 5.25 do not match. They give the example that if the parameter had some hypothetical units, the changes to the parameter should be changes in those units as well. To match the units, we introduce another exponentially decaying average but this time with squared parameter updates:

$$E[\nabla\theta^2]_t = \gamma E[\nabla\theta^2]_{t-1} + (1-\gamma)\nabla\theta_t^2 \tag{5.26}$$

and thus the RMS of $\nabla\theta$ is:

$$RMS[\nabla\theta]_t = \sqrt{E[\nabla\theta^2]_t + \epsilon} \tag{5.27}$$

Note that, $RMS[\nabla\theta]_t$ is not available. As a solution we use an approximation, namely $RMS[\nabla\theta]_{t-1}$. By using $RMS[\nabla\theta]_{t-1}$ instead of the learning rate we have the final weight update rule of Adadelta:

$$\theta_{t+1} = \theta_t - \frac{RMS[\nabla\theta]_{t-1}}{RMS[g]_t} * g_t \tag{5.28}$$

Because the learning rate $\eta$ is replaced by $RMS[\nabla\theta]_{t-1}$, we do not need to predefine it.

### 5.5.4 RMSProp

RMSProp is an unpublished gradient descent variant and was introduced during lecture6a of a Coursera course (2002) by Geoff Hinton (Geoffrey Hinton, 2012). RMSprop and Adadelta were indepedently development during the same time period, both with the goal to resolve the continuous decay of the learning rates in Adagrad. As a result is RMSprop the same as equation 5.25 from Adadelta with the suggested value $\gamma = 0.9$:

$$\theta_{t+1} = \theta_t - \frac{\eta}{RMS[g]_t} * g_t \tag{5.29}$$

with

$$E[g^2]_t = 0.9E[g^2]_{t-1} + 0.1g_t^2 \tag{5.30}$$

Hilton also suggests to use value 0.001 as the learning rate $\eta$.

# Chapter 6

# Deep learning

Deep learning is currently one of the most popular research fields in machine learning. Mostly because of the major breakthroughs in the past years. For example DeepMind, later acquired by Google, made a major breakthrough in handling complex learning problems such as game playing by combining deep learning techniques with reinforcement learning (Silver et al. (2016)). Maybe the most recent example is the development of Alpha Go, where the program was able to defeat the world champion Lee Sedol in Go (Moyer (2016)).

Deep learning can be considered a study of artificial neural networks. The term is used, when the learning involves an artificial neural network, with more than one layer and a lot of data. Jeff Dean, a Google Senior, made in 2016, during his talk, the following comment on deep learning (Dean, 2016):

*When you hear the term deep learning, just think of a large deep neural net. Deep refers to the number of layers typically and so it's kind of the popular term that's been adopted in the press. I think of them as deep neural networks generally.*

The reason why deep learning is currently so popular is because we now have the computing power and the huge amounts of data available. The improvements in algorithms, better insights and improved techniques help us to achieve better results. Not only are the current breakthroughs with deep learning encouraging, the scalability of deep learning is even more promising. The more data we have available the better results we can achieve. Andrew Ng, Standford professor and co-founder of Google Brain, emphasised this fact during his talk on ExtractConf 2015 (Ng, 2015). He stated that the performance of the traditional learning algorithms will plateau, while deep learning algorithms are the first class of algorithms that are scalable. Scalable in a sense that the performance of deep learning algorithms will just keep getting better as we feed them more data. As an illustration in his talk, Andrew Ng included the following cartoon, which emphasizes the power of deep learning algorithms:

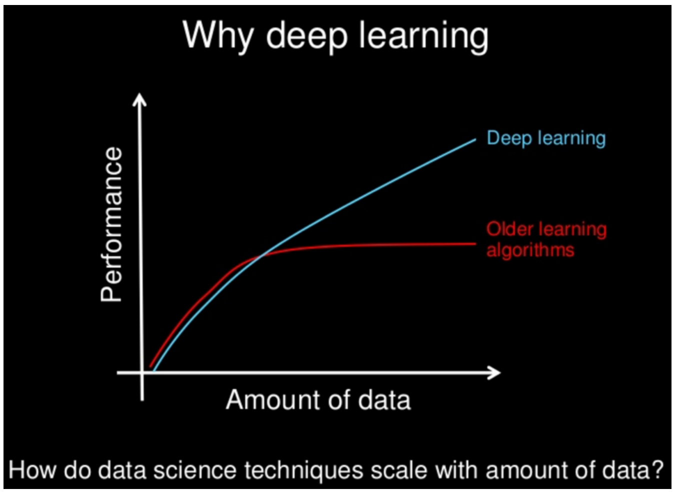


Figure 6.1: Cartoon on 'What data scientists should know about deep learning' by Andrew Ng during his talk on ExtractConf 2015 (Source:`https://www.youtube.com`)

The increase in network size allows us to model much more complex data and feed for example raw input as image- or audiofiles to the network. It does not restrict us to sorted or tabular input data. This allowed Yann Lecun to introduce a new neural network architecture, named Convolutional Neural Networks (CNN) (Y. LeCun et al., 1995). This technique shows great success in object recognition in images.

## 6.1 Convolutional Neural Networks

Convolutional Neural Networks (CNN), also called Convnets, were first introduced by Y. LeCun et al. (1995) and are now the standard technique for image classification. An contemporary example is the automated photo tagging by Facebook (Taigman et al., 2014), which is using CNN's at its core. A CNN is able to learn specific features from a raw image without any extra information. It allows also hierachical feature extraction by stacking the convolutional layers. Concrete, this means that a layer connected to the input layer can identify edges and the deeper layers can identify other higher-level features such as eyes. The inspiration of the Convolutional Neural Networks came from an animals visual cortex, where you have simple cells and complex cells. The simple cells serve as an edge-detector. They have a small receptive field and are very responsive to edges. The complex cells on the other hand have a larger receptive field and respond to much more specific impulses. An interesting feature of these complex cells are the spatial invariance. No matter where the specific impulse is located, the complex cells are still able to detect the impulse. Convnets have also this property of spatial invariance and perform very well in capturing local spatial patterns. In 2000, Simard et al. (2000) were able to increase the performance of Convnets even more by introducing an invariance for simple transformations, such as scaling, rotation and squeezing. As we will explain the architecture of Convolutional Neural Networks in the next section, we will be mainly inspired by the LeNet-5 network by Y. LeCun et al. (1995). A Convnet developed to recognise hand-written digits.

## 6.2 Building Blocks

Convolution Neural Networks are, as the name suggests, mainly based on convolution. In this section we will discuss the main components of a CNN.

### 6.2.1 Convolutional Layer

Convolutions layers are the core component of Convolutional Neural Networks. Let us consider the input of the Convnet to be an image for the remainder of the text. A convolution layer is a layer in the neural network, which is able to extract features from the image by using convolution. Although, we will see that the internal structure of a convolution layer is different than we have seen in chapter 5. Let us first consider convolution.

Convolution in image processing serves as a general purpose filter effect (Ludwig, 2013). An image can be represented by a 3 dimensional matrix with pixel values and size $w \times h \times d$ with $d$ the amount of channels. For example an RGB image has 3 channels, namely red, green and blue. To explain convolution, we consider grayscale images. These images have only one channel.

Convolution can be seen as sort of filtering. It is applied by sliding a matrix over the input image and output a modified filtered image. We call the matrix sliding over the image, the *kernel* or *mask*. Determining the pixels of the filter image happens by calculation the value of the central pixel by adding the weighted values of all its neighbours together. Mathematically we can describe convolution by:

$$Dest[i,j] = \sum_{k=0}^{K_w} \sum_{l=0}^{K_h} Kernel[k,l] Source[i+k, j+l] \tag{6.1}$$

with $K_w$ and $K_h$ the kernel width and height. Figure 6.2.1 illustrates the method.

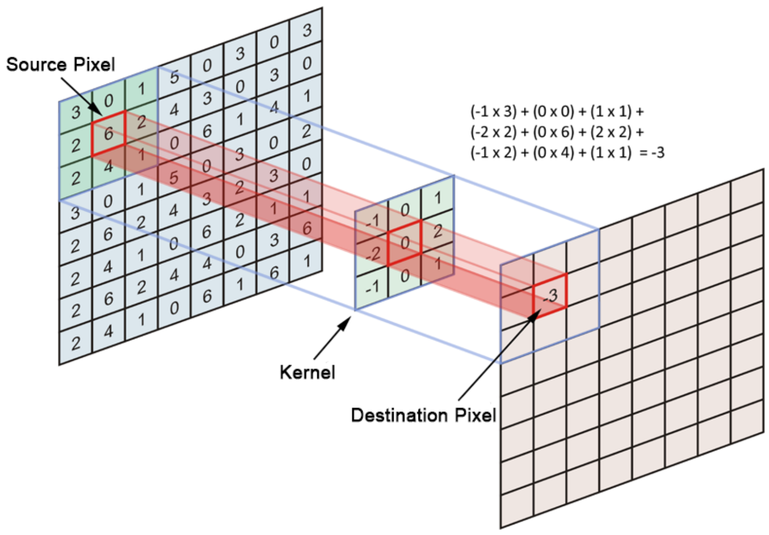

Source Pixel
(-1 x 3) + (0 x 0) + (1 x 1) +
(-2 x 2) + (0 x 6) + (2 x 2) +
(-1 x 2) + (0 x 4) + (1 x 1) = -3
Kernel
Destination Pixel

Figure 6.2: Illustration on how convolution works. The destination value gets calculated by taking the weighted sum of its neighbours and itself. The weight for each neightbour is defined by the value on the corresponding position in the kernel.

What makes convolution so great, is the ability to extract several features by simple changing the kernel values. Figure 6.2.1 illustrated how it is possible to do edge-detection by using specific kernels.

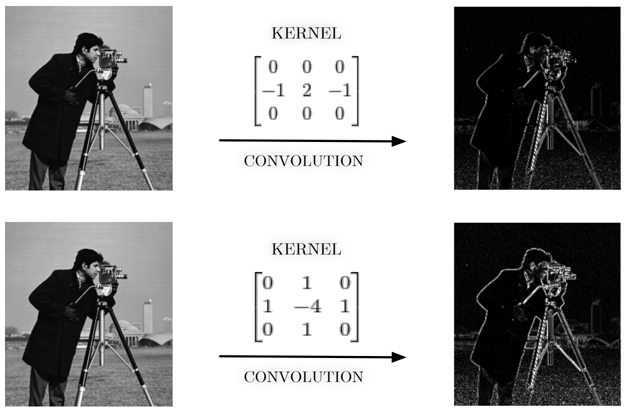

KERNEL
$\begin{bmatrix} 0 & 0 & 0 \\ -1 & 2 & -1 \\ 0 & 0 & 0 \end{bmatrix}$
CONVOLUTION
KERNEL
$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$
CONVOLUTION

Figure 6.3: Convolution allows us to extract features. This example illustrates how it is possible to do edge-detection by using a specific kernel. We can tune the sensitivity of the detection by changing the values of the kernel.

Now, how do we integrate convolution in a Neural Network? Well, each neuron of a convolutional layer is connected to a small region of the input. All the neurons together form a filter, which covers the whole input. This filter uses convolution to represent a flat 2D representation for a single feature of the input. The output is also called the *Feature map* or *Activation map*. Of course, we would like to extract multiple features from the input. By stacking the filters we can achieve this. Eventually, we can describe a convolutional layer as a stack of filters, which outputs different feature maps.
Important to note is that each filter represent as single feature and each exists out of a set of different neurons, which cover each a small region of the input. To be able to extract the same feature with the set of neurons, the weights have to be shared. These shared weights make convolution possible and allow us to extract a specific feature, wherever it is located in the input. The configuration of these weights is actually what the Convnet will try to learn. It will try to specific the features that are important to solve the problem. Despite the automatic configuration, we are still required to decide on several hyperparameters

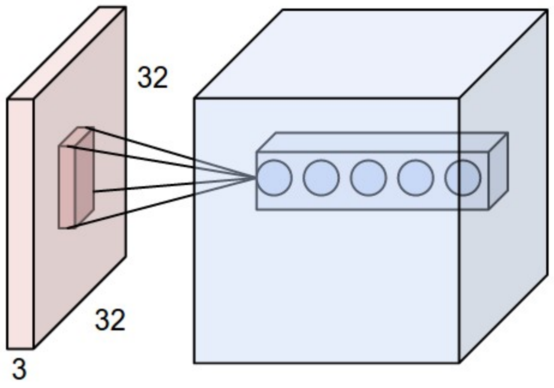


Figure 6.4: A convolutional layer connected to the input of 32x32x3. A row of neurons (5 in total) is connected to the small region of the input. Every neuron is part of a different filter and looks for another feature. The neurons that are all looking at the same region, are also called a *fiber* (Source: `http://cs231n.github.io/`)

**Hyperparameters**

Convolution Neural networks are tricky to train, since they have more hyperparameters than a normal Neural Network. After constructing the general architecture of the network, we are still required to set the amount of filters, size of the filters, stride and padding for each convolution layer.

**Amount of filters** The amount of filters decides how many feature maps we produce with the convolution layer. The more filters, the more features we can extract. But we need to keep in mind that the cost of a single convolution filter is rather high.

**Tile size** The size of the tile that we slide over the input is also a parameter. This really depends on the dataset and vary greatly in the literature.

**Stride** The stride determines the spacing between the receptive fields. It is the amount of pixels you slide your field or tile over the input. For example, if you have stride one, you shift your tile one pixel at the time. Depending on the size of your field and the stride, we get overlap of the receptive fields. When there is overlap, our feature map will have a larger spatial size. Only when the stride is equal to the size of the receptive field, there will be no overlap and is the spatial dimension equal. A very large stride is also possible, but it is less popular in practice.

**Padding** Sometimes it is the case that the input is too small to complete the convolution. A reason can be the size of the tile or the combination of the tile and the stride. As solution is zero-padding, which allows us to complete the convolution.

### 6.2.2 Non Linearity

In the explanation of the convolution layers, we left out an important detail about the feature maps, namely the application of non linearity. Since Convnets are also a neural networks, they are also able to introduce non linearity. So in general, the neuron of a convolutional layer first convolutes the input, which is a linear function, adds a bias and then applies a non-linear function such as the ReLU or Tanh (see section 5.2.4). Mathematically we can describe the $k$-th feature map $h_k$ by:

$$h^k_{ij} = \tanh((W^k * x)_{ij} + b_k). \tag{6.2}$$

with $h^k_{ij}$ the pixel on the $ijth$ position in the feature map.

### 6.2.3 Pooling Layer

Another important component of Convolutional Neural Networks are Pooling layers. Pooling layers are commonly used in-between two convolutional layers. The goal of pooling is to reduce the computation and overfitting. This is done by reducing the size of the representation. An important idea behind the pooling is that the activation of a feature and its relation towards others is more important than the location of the feature. But note that the relative location of the features is preserved. A very popular type of pooling is MAX-pooling, demonstrated in figure 6.2.3.

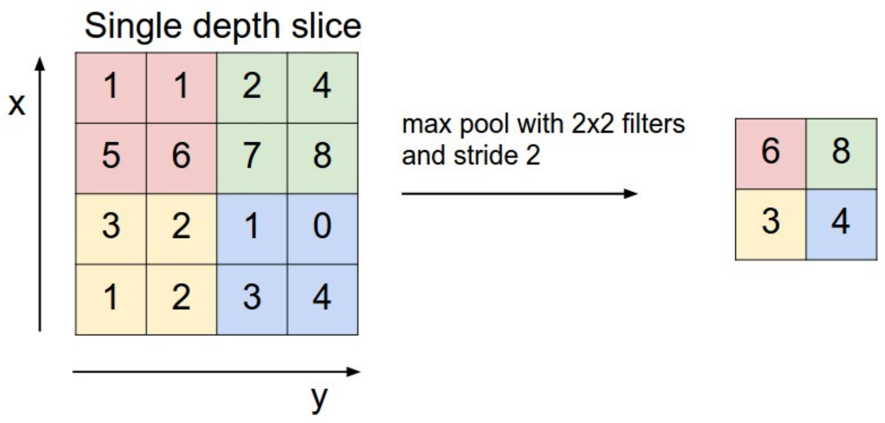


Figure 6.5: Max pooling with a 2x2 filter and stride 2. The filter is sliding over the input and takes the max value as outputvalue for the pooling. (Source: `http://cs231n.github.io/convolutional-networks/#pool`)

So far convolution layers. We've discussed convolution, the different building blocks of the network and the two most important aspect of Convnets: location invariance and compositionality. Next, we continue by briefly discussing autoencoders, another interesting Neural Network architecture.

## 6.3 Autoencoder

So far, we have seen Neural Networks, which apply to supervised learning. An autoencoder is a special neural network architecture, which applies to unsupervised learning, since they don't need labeled data. An autoencoder is an neural network, which tries to reconstructs its input values, by doing backpropagation on the output values. An example of an autoencoder is:

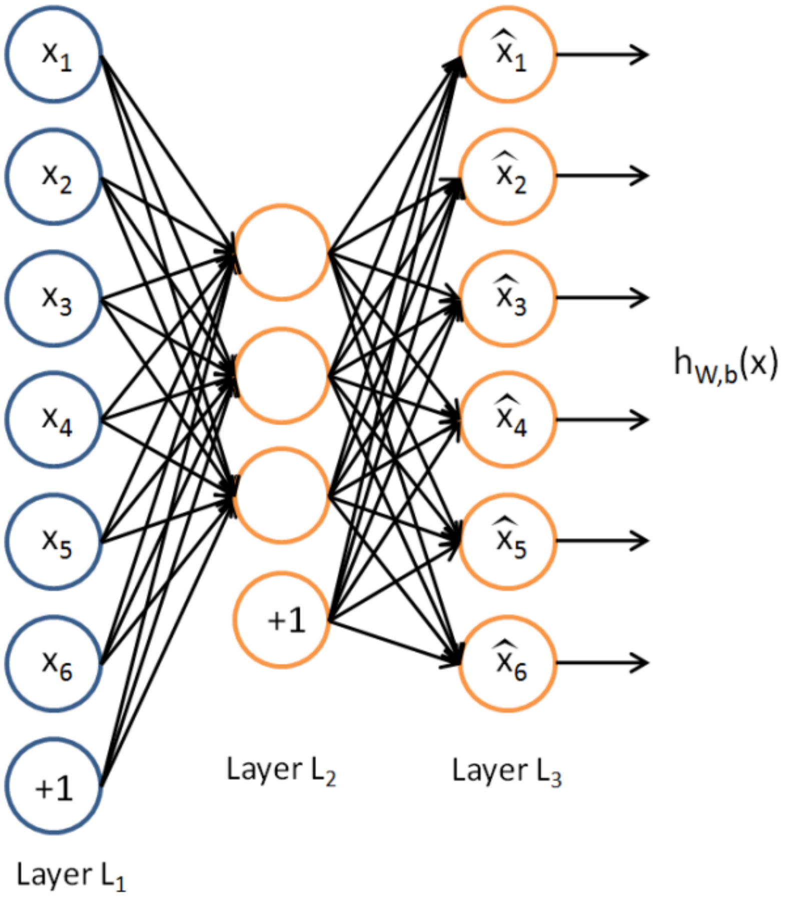


Figure 6.6: figure
Example of an autoencoder (Ng, 2011)

The autoencoder tries to learn the function $h_{W,b}(x) \approx x$ and to approximate the identity function. It may seem trivial that we want to learn the identify function, but it allows us also to discover interesting structures in the data. We can force the autoencoder to learn a 'compressed' representation of the data by limiting the amount of hidden units. When the input data is completely random, it is very difficult for an autoencoder to learn a compressed representation. But when there is a certain structure in the data, it could let us discover some interesting data structures (Ng, 2011). The low-dimensional representation is mainly the reason why researchers are interested in autoencoders (Vincent et al., 2010). It allows them to replace the handcrafted features with features, generated by the autoencoder in an unsupervised way.

### 6.3.1 Stacked autoencoders

Even if we don't limit the amount of hidden units, it is possible to learn a compressed representation. We do this by imposing a 'sparsity' constraint $\rho$ on the hidden units. Even if the amount of hidden units is large, this will still allow us to discover interesting data structures. When we explicitly impose a sparsity constraint on autoencoders, we speak of *sparse autoencoders*

### 6.3.2 Denoising autoencoders

Another way to retrieve a good representation is denoising. Vincent et al. (2008) introduced this technique with the idea that a good representation is one that can be obtained from a noisy input and allows the autoencoder to reconstruct to the original noise free input. In figure 6.7 is the architecture of a *denoising autoencoder* displayed. Denoising an autoencoder requires only a small adjustement to the classical autoencoder. The inputvector $x$ is converted to inputvector $\tilde{x}$, by a stochastic mapping $\tilde{x} \sim q_D(\tilde{x}|x)$. The mapping makes $x$ corrupt by masking certain inputs with zero. An example of a stochastic mapping is Gausian Noise. After the corruption,the autoencoder tries to do two things. First, it tries to transform the corrupted input $\tilde{x}$ to a good presentation $y$. Subsequently it tries to use this good representation $y$ to reconstruct the original input, with as a result the reconstruction $z$. The autoencoder does the reconstruction by capturing the statistical dependencies between the inputs. Note that, the reconstruction error $L_H(x, z)$ is measured between the original input $x$ and $z$

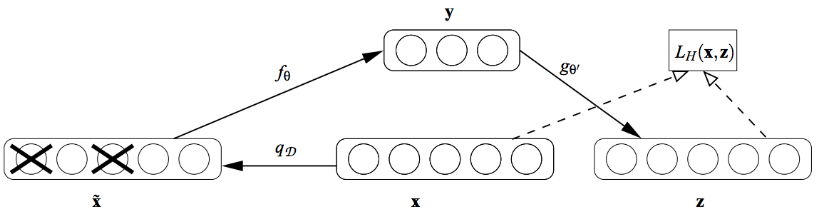


Figure 6.7: figure
Architecture of a denoising autoencoder (Vincent et al., 2008)

### 6.3.3 Stacked Autoencoders

Another interesting variant of autoencoders are *stacked autoencoders.* These architectures are found to be useful to capture hierarchical grouping or part-whole decomposition. A stacked autoencoder can be defined as a neural network consisting of multiple layers of autoencoders in which the outputs of each layer is wired to the inputs of the next layer. Let us explain this with an example. Consider the stacked autoencoder in figure 6.10. The first hidden layer outputs the compressed representation of the input and serves this to the next layer (see figure 6.8). In the second hidden layer, we see the same structure only this time outputs the layer the compressed representation of the input by the first hidden layer (see figure 6.9). Eventually, we can connect this layer to a dense output layer to do classification. Because we have for each hidden layer an autoencoder, we introduce a hierachical feature extraction. The first hidden layer extracts low-level features, where deep hidden layers are extracting high-level features.

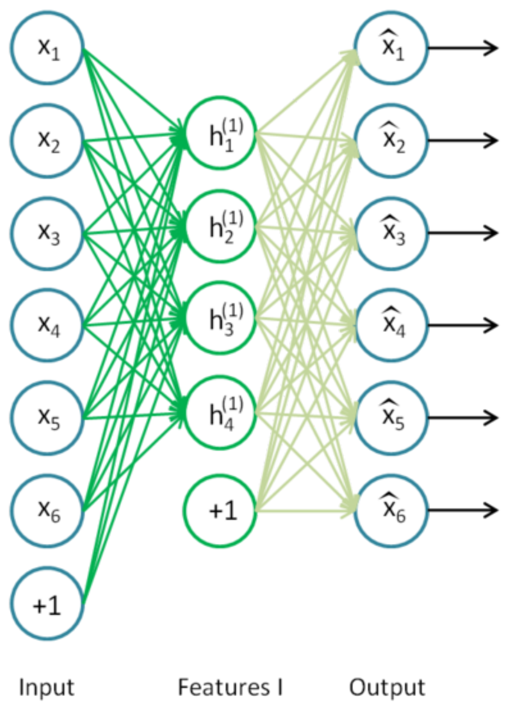


Figure 6.8: First hidden layer of Stacked autoencoder in figure 6.10 (Source: `http://ufldl.stanford.edu/`)

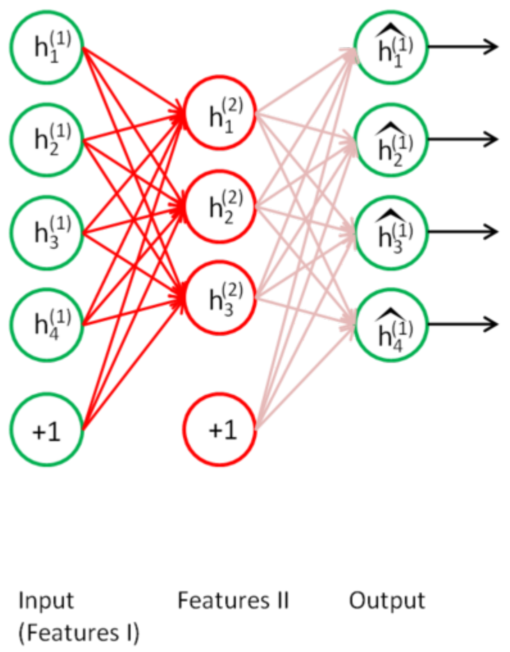


Figure 6.9: Second hidden layer of Stacked autoencoder in figure 6.10 (Source: `http://ufldl.stanford.edu/`)

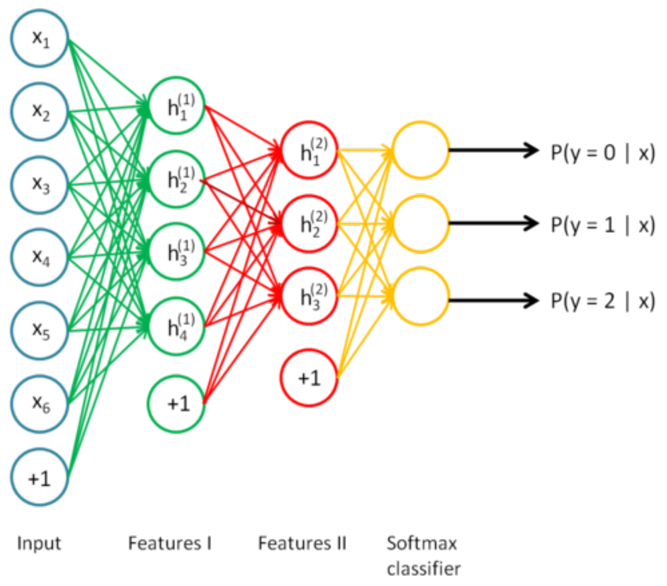


Figure 6.10: Stacked autoencoder architecture (Source: `http://ufldl.stanford.edu/`)

In summary, we have seen that deep learning is a term for very big multi-layered neural networks. An important property of deep learning algorithms is that the increase in performance goes together with the increase in data. Also, we described Convolutional Neural Networks and several types of autoencoders, two deep learning architectures. Convolutional Neural Network are the standard for image classification and have the important quality of being location invariance and compositionality. Autoencoders are mainly used to retrieve, in a unsupervised way, deep representations of data.

# Chapter 7

# Experiments and Results

## 7.1 Problem Statement

In section 3, we described canvases and how they are used to identify a device. Pixel wise comparison was mainly the method suggested in the literature. In this research, we approach it in a more advanced way. Our goal is to develop a new authentication method. A method that uses the combination of machine learning algorithms and canvases to identify a device. We see this in the context of an authentication mechanism, where we register a device to the system and subsequently the device can be identified by our method. As we said before in chapter 2, the use of machine learning is nothing new for authentication mechanisms and is currently very popular. What our research explores is the combination with canvases. There is not much research available about the authenticity of canvases and especially on how they perform in combination with machine learning. So, we don't know if it is even possible to create an authentication method with this combination.
For this research, we work together with the Université Catholique de Louvain (UCLouvain). They developed a very simple canvas based authentication method, which we will use as the starting point for our authentication method.

The rest of this chapter is organized as follows. First we describe the starting point, delivered by UCLouvain. Then, we specify the construction of the canvases. Next, we continue with describing the collected data and the data gathering process. After this, we run several experiments, which are all sequentially. In section 7.8, we get familiar with the collected data and do an initial exploration. This will give us an idea on how to approach things. Thereafter, we empirically evaluate several ideas of possible authentication methods. Eventually we describe in section 7.10 and 7.11 two final authentication methods.

We ran all the experiments on the HPC-Cluster of the AI lab at the VUB and each experiment had approximately a runtime of 7 to 10 days.

## 7.2 Starting Point

Like we said, we do not start from scratch. The starting point is a authentication system, delivered by the ICTM team at the UCLouvain. They constructed a simple authentication method, based on canvases. We use their conceptual model for our authentication method and consider the performance of their method as our baseline.

The conceptual model is as follows. First, the user has to register his device to the system. This is done by sending a big amount of random canvas drawings and the user's e-mail address to the system. Once the registration is completed, the user can identify himself to system with the registered device by submitting the registered email and a new set of random generated canvases. Note that, the user does not see the canvas drawings and has only to submit an email. Figure 7.1 illustrates the conceptual model. As stated before, we use the same conceptual model for our authentication method.

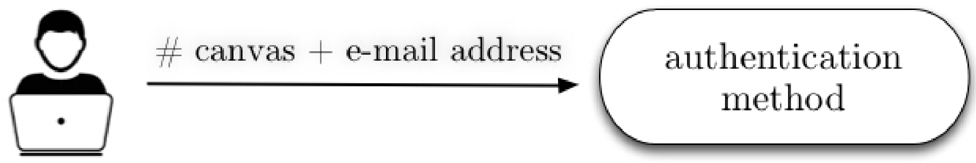


Figure 7.1: Conceptual model of the authentication method.

Of course, the essential work happens in the backend, where the canvases are analysed by an algorithm and an unique fingerprint is generated. The authentication method, delivered by UCLouvain, uses a very simple algorithm. It defines a threshold based on the pixel values of the collect canvases during the registration. This threshold is defined by calculating the variance of the non-zero pixel values in a canvas. If the canvases, sent during the authentication attempt, falls within the interval of the average pixel value for the canvases of that user and two times the variance, the authentication is accepted. Formally we write this down as:

$$authstatus = |\overline{PC}_{login} - \overline{PC}_{register}| < 2 * Variance_{register} \tag{7.1}$$

With $\overline{PC}_{login}$ the average amount of non-zero pixels in a canvas that is sent during the authentication. $\overline{PC}_{register}$ is the average amount of non-zero pixels in a canvas that is sent during registration and $Variance_{register}$ is the variance of these pixel counts.

This method is very simple, but it is performing very bad. It performs the same as an authentication method that would accept users randomly. Later in section 7.7, we will explain the performance of this method in more detail. Our goal is to develop a new more advanced method in the hopes to increases the performance significantly. In the next section we continue by discussing how we can evaluate the performance of our authentication method.

## 7.3 Task and Performance Measure

We want to develop a new authentication method that works well. But how do we evaluate the performance? First, we define the task $T$ as: Given the user's identity, his device and the knowledge of a labeled set of canvases, make a correct judgement about the origin of the unknown canvases with regard to the user's device. The judgement consists of saying if a canvas originates from a certain device. This means that there are only two possible outcomes, namely 'Yes, the canvas originates from the device' or 'No, the canvas originates not from the device'. This means that we see our authentication method as a binary classification. Subsequently, we define the performance $P$ as the percentage of good judgments by the authentication method.

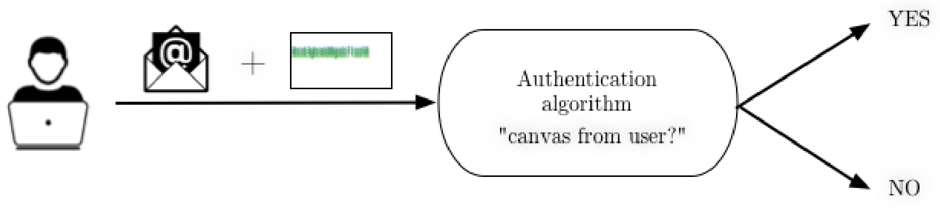


Figure 7.2: The performance of an authentication method is evaluated, based on its ability to decide if the canvas is from the user's device or not.

So, we have stated our task $T$, performance $P$ and the general outline of our method, but we are still missing the construction method of the canvases.

## 7.4 Canvas Fingerprint

In chapter 3, we already described the many possibilties that can be drawn on a *<canvas>* element. For this research, we use the drawing technique developed by UCLouvain.
This drawing technique is very simple and follows for each canvas the same steps. It draws a random composed string multiple times on a slightly different position. The parameters: size, string length, position, font and color are all predefined. The drawn string is the only variable for each canvas. Figure 7.3 is an example of a canvas generated by this technique.

The following javascript code is used to generate a canvas:

Listing 7.1: Construction of a canvas

```
function generateRandomCanvas(txt) {

  var canvas =  document.getElementById("canvas");
  var ctx = canvas.getContext("2d");

  if (txt == null)
     txt = generateRandomText(25);
  ctx.clearRect(0,0, 300,150);

  ctx.font= "18px 'Arial'";
  ctx.textBaseline = "alphabetic";

  //static colors and style
  ctx.fillStyle = "#069";
  ctx.fillText(txt, 2, 50, 350);
  ctx.fillStyle = "rgba(102, 204, 0, 0.7)";
```

```
    ctx.fillText(txt, 4, 54, 350);
    ctx.fillStyle = "#069";
    ctx.fillText(txt, 2, 58, 350);
    ctx.fillStyle = "rgba(102, 204, 0, 0.7)";
    ctx.fillText(txt, 4, 62, 350);
    ctx.fillStyle = "#069";
    ctx.fillText(txt, 2, 66, 350);
    ctx.fillStyle = "rgba(102, 204, 0, 0.7)";
    ctx.fillText(txt, 4, 70, 350);
}
```

In words, this means that we specify a canvas as a $300 \times 150$ container of 6 strings with all the same color and font Arial with size 18px. Also, every string has a slightly different position. This is the specification of a canvas, we will use throughout the research.

We also considered more advanced drawing techniques, since there is not really a reference on how we should draw a canvas, suitable for authentication. For example, we could add different shapes or fonts. All these extra's would add more information to the canvases. But we decided to stay with this simple technique, because it shows, later in the experiments, to be informative enough and it can serve as a reference for more advanced drawing techniques in future research.

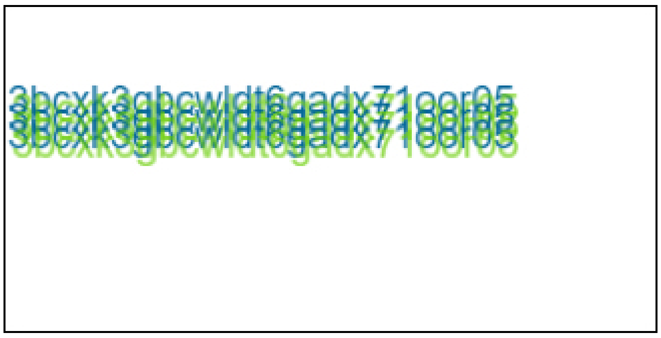


Figure 7.3: Example of a canvas drawing, generated by the code in snippet 7.1.

## 7.5 Data Gathering

If we want to use machine learning, we need data. At the start of this research, there was no data available, so we needed to collect it. We did this by building a data collecting website. The canvases where constructed with the code described in snippet 7.1 and from each device were 2000 canvases collected. We also labeled each canvas with meta-data, so that we could keep track of the origin of the canvases. We collected the data by asking friends, family and fellow students on campus to register all their available devices to the system, such as a smartphone, tablet or PC. We assured that users could not register their device multiple times by using the fingerprintjs2[1] library. This is a browser fingerprinting library, which uses all the available public browser data to generate a unique fingerprint.

[1]`https://github.com/Valve/fingerprintjs2`

The following table and pie charts show use some information about the collected data:

| | # |
|---|---|
| **Devices** | 80 |
| **Users** | 49 |
| **Canvases** | 147387 |

Table 7.1: Amount of collected users, devices and canvases.

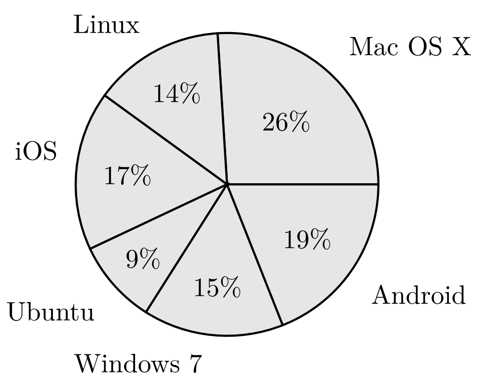


Figure 7.4: Pie chart about the Operation Systems

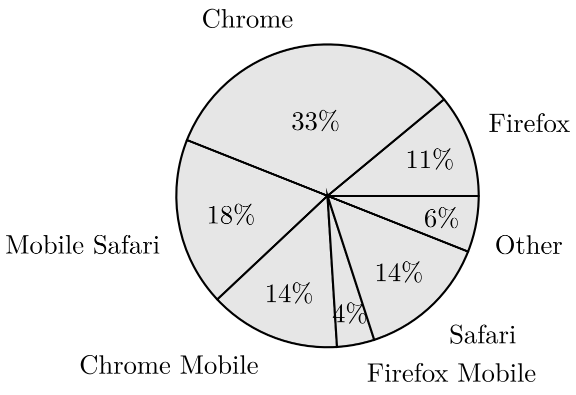
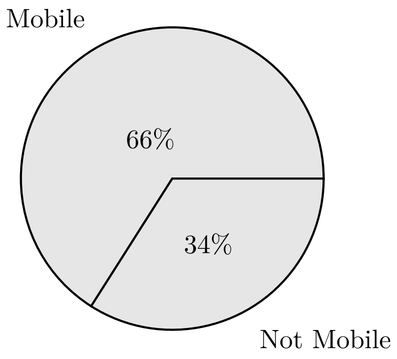


Figure 7.5: Pie chart with the different types of browsers

Figure 7.6: Partioning between mobile and non-mobile devices



In total, we have collected 80 devices. This seems enough, since we have 147387 canvases to learn from. Note that, the uneven amount of canvases is caused by uncompleted registrations. Still, 87,5% is a decent completion rate. The information about the origins of the canvases are rather informative and less important, considering the fact that normally we do not know our data and only gain knowledge from the canvases.

## 7.6 The Dataset

In machine learning, we learn a concept to a learning algorithm by feeding it data. This makes the dataset an important part of our experiments. How do compose the dataset and how do we assure that the learning algorithm learns our desired concept? All these questions are answered in this section.

As we said earlier, we want to learn an algorithm to recognize if a given canvas originates from a certain device or not. We do this by training the algorithm specifically for one device. Consider, we want to learn a learning algorithm to recognize the canvases of device $A$. The desired behaviour of the learning algorithm would be to output 'Yes', when the given canvas originates from device $A$ and 'No', when the given canvas originates from a device that is not $A$. To learn this concept, we compose our dataset with canvases from device $A$, labeled with 'Yes' and canvases that are not from device $A$, labeled with 'No'. In the context of our authentication method and the experiments, we have the following scenario: A user registers his device to our authentication system by sending an amount of canvases. The authentication system receives the canvases and composes a dataset for the learning algorithm. The dataset consists of the received canvases with the label 'Yes' and other canvases from other devices with the label 'No'. The 'No'-canvases are selected from the authentication system's database, where other canvases from other registered devices are stored. This dataset of canvases with 'Yes'- and 'No'-labels is than used to train the algorithm. At last, whenever the user tries the authenticate with the registered device to the authentication system by sending his canvases, the system will verify these canvases by feeding it to the trained algorithm, associated with the user's device. If the algorithm predicts 'Yes' for the canvases, the user is successfully authenticated. Note that, the 'No'-labeled canvases are as important as the 'Yes'-labeled canvases. This is because the canvases of the originating device are not informative enough. If we would feed only 'Yes'-labeled canvases to the learning algorithm, the algorithm would learn nothing. The information in the 'No'-canvases is as essential as in the 'Yes'-canvases to learn the concept.

The above mentioned composition is the composition of the dataset we use for all experiments. More over, we always compose a balanced dataset for the experiments. This means that the amount of 'Yes-labeled' canvases is equal to the amount of 'No'-labeled canvases in the dataset. The size of the dataset is always the same for all experiments. It has always a total size of 4000 canvases, with 2000 'Yes'-labeled canvases and 2000 'No'-labeled canvases. How we use the dataset for training, testing and evaluation, is discussed in each experiment separately.

## 7.7 Baseline Experiment

The first experiment is a baseline experiment. We take as the baseline the performance of the developed system by UCLouvain. We evaluate this system by using the defined performance measure $P$ in section 7.3. As explained in section 7.2, UCLouvain's authentication method calculates a threshold to decide whether a canvas originates from a given device or not. The system calculates for each device a corresponding threshold. The threshold is obtained by taking two times the variance of the non-zero pixels in the canvases that are sent during the registration of the device. If the user authenticates with his registered device and sends in a canvas, the system will calculate the variance of the non-zero pixels in that canvas and compare this with the threshold. If the value is smaller than the threshold, the canvas is considered to be from the same device. If the value is bigger, it is considered to be from another device.

In this experiment, we simulate this scenario by taking 80% of the ‘Yes’-canvases in the dataset to calculate the threshold. This 80% represents the canvases that are sent during the registration. Further, we use 20% of the ‘No’-canvases and the remaining 20% of the ‘Yes’-canvases to evaluate the performance of the authentication method. This simulates the tries of the authorized user and the unauthorized users. Authorized users are users that the system should accept and unauthorized users are the users the system should deny. Note that, we use a balanced set of ‘Yes’-canvases and ‘No’-canvases to evaluated the method. This simulates that the amount of tries of the authorized user is equal to the amount of tries of unathorized users. This allows us to use the accuracy of a random authentication system as a baseline for this experiment, which is 50%. A random authentication system is an authentication system that accepts and denys authentication attempts with an equal probability. The results of this experiment are the average of 10 runs, where every time another dataset is used.

| | **2*var-Threshold** | **1*var-Threshold** |
|---|---|---|
| **Accuracy** | 0.5 | 0,49 |
| **Std** | 0 | 0,02 |

Table 7.2: Results Baseline Experiment

Based on the results, we can say that UCLouvain’s method is not working. We tested it with the delivered threshold of two times the variance and even tried it with a stricter threshold of one time the variance, but both give the same result. If we take a look at the confusion matrix in figure 7.7, we see that this method allows every canvas. This means that every authencation attemp gets accepted, no matter from which device the canvases are coming. For the rest of the experiments, we will consider the baseline of 50% on a balanced set, which is also the performance of a random authentication system.

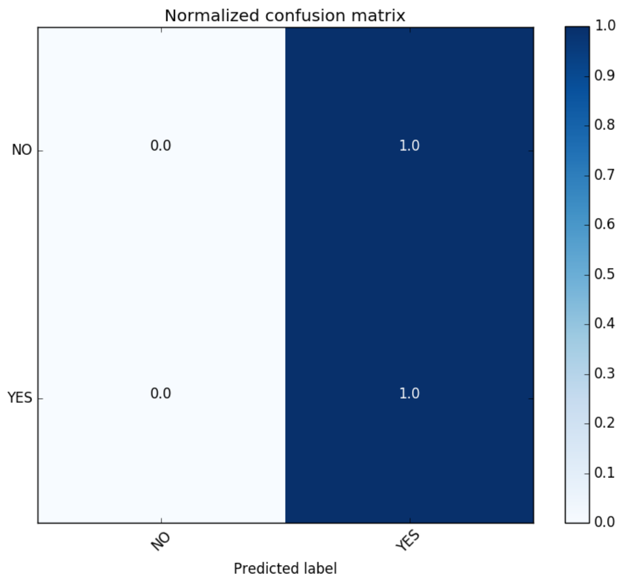


Figure 7.7: Confusion Matrix of Baseline experiment with 2*Var

## 7.8 Data Exploration

There is not much research available about canvases and how to use them as fingerprints. It is useful to do some explorational experiment to see what is possible. Remember, we have 2000 canvases per device available and each canvas is 300 × 150 pixels.
A good first step is converting the canvases to a flat matrix. Then reduce it to two features and visually plot it on a 2D surface. This allows us to see if it is possible to visually separate the canvases of two devices. We do this by taking the RGBA value for every pixel, and use PCA from section 7.9.2 to reduce the dimensions. After doing this for a few random pairs of devices, we discovered that we can group the plots in two groups. One group of plots shows that it is possible to separate the canvases, while the other group shows on big chaos. This is very surprising. Especially because we are reducing a canvas from 180000 features to 2 features. This is a massive reduction in information. Overall, this is already an interesting and encouraging finding and it is a good indication for the next experiments. Figure 7.8 shows examples of the groups of plots we encountered.

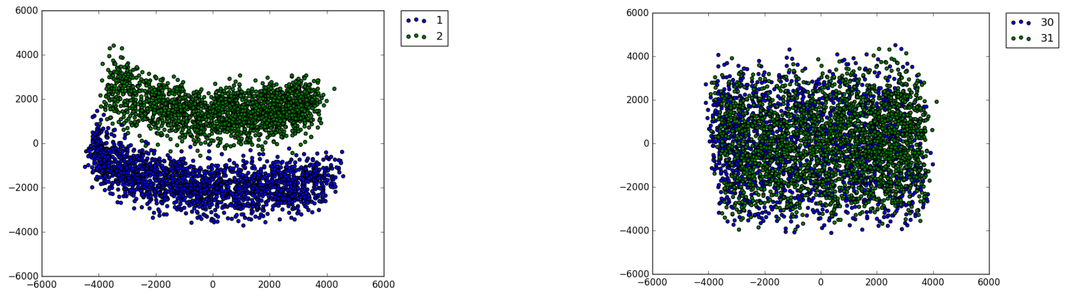


Figure 7.8: Samples of the two kind of plots that we could observe during the data exploration

## 7.9 K-NN

Our first idea is to use a K-Nearest Neighbors Classifier. As earlier mentioned in section 4.3.2, the K-Nearest Neighbors Classifier is an instance-based learning algorithm, which uses samples from the memory to decide on other samples. It is an easy method and often undervalued for image classification (Boiman et al., 2008). We discuss in this section three approaches, all related on how we pre-process the data, before we feed it to the K-NN Classifier. In section 7.9.1, we start very simple and feed the raw form of the canvases to the classifier. Next, we try in the two following sections to reduce the dimensionality of the canvases, by using PCA and autoencoders.

### 7.9.1 Raw

To evaluate, what is exactly possible with the K-NN Classifier, we start very simple. We do not pre-process the canvas and we feed it in its raw RGBA-matrix form to the K-NN Classifier. Note that, this is actually very unusual to do, since the classifier has to work in an 1800000 dimensional space. Before we actually perform the experiment, we need to tune the classifier. The K-Nearest Neighbors Classifier has two hyperparameters , namely the number of neigbors $K$ and their weights. For the weights, we have the choice between the uniform and distance weight types. If we choose uniform weights, all the neighbors are weighted equally. If we use distance as a weight type, the weight point of the neighbors are equal to the inverse of their distance. This results in a greater influence for closer neighbors and less influence for neigbors further away. We evaluate each setting with 10-fold cross-validation.

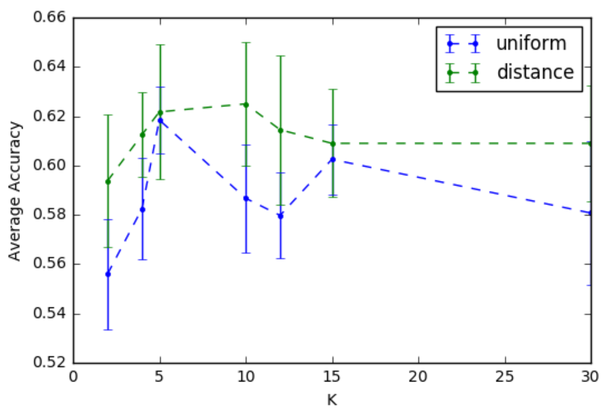


Figure 7.9: Tuning of number of features for PCA

Overall, the 'distance' weights have the best performances and for this configuration seems K=10 the optimal configuration.

Now if we run the tuned configuration, K=10 and weight type distance, for 10 runs, we get the following result:

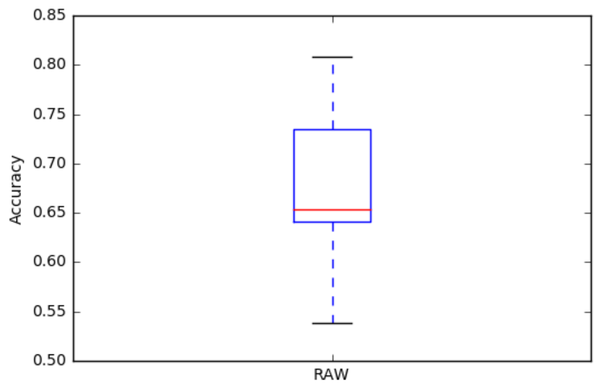


Figure 7.10: Box plot with the results of the K-NN with RAW input

| **Accuracy** | |
|---|---|
| **Mean** | 0.67 |
| **Std** | 0.08 |

Table 7.3: Results RAW K-NN Classification

The K-NN Classifier performs better than the baseline, with an average accuracy of 67%. But we see that the data is widely spread. It would be better if we see a more constant accuracy.

Overall it is a good start, but there is still room for improvement. Currently we feed too much information per canvas to the classifier and this results possible in overfitting. As an adjustment, we reduce these 1800000 features to a smaller and more efficient representation.

### 7.9.2 PCA

We saw already in section 7.8 that a reduction with PCA to two dimensions can work and indicates that it is possible to reduce the canvases to a more efficient representation. When we take into acount the results of the previous experiment, it is interesting to investigate the combination of PCA and the K-NN Classifier on our dataset. We continue by using the optimal configuration of the previous experiment, which is a 10-Nearest Neighbors classifier with weight type distance. First, we decide on the number of features to reduce. We evaluate this again by using 10-Fold Cross-Validation.

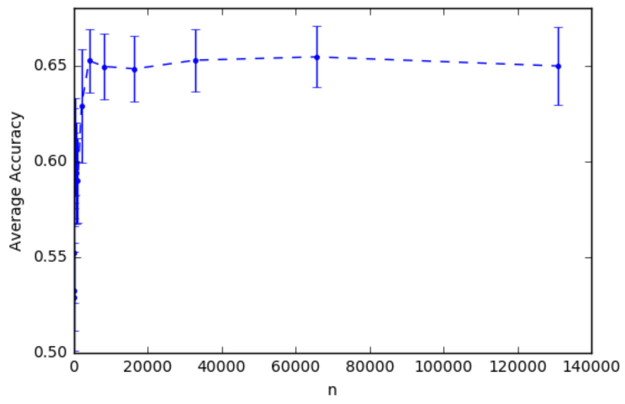


Figure 7.11: Tuning of number of features for PCA

We see in figure 7.11 that there is a stagnation after an amount of 4096 features. We take this as the new number of features to represent our canvases. Now, we run the experiment for 10 runs, where the canvases get reduced to 4096 features by applying PCA before they get classified by the 10-NN Classifier.

| **Accuracy** | |
|---|---|
| **Mean** | 0.65 |
| **Std** | 0.08 |

Table 7.4: Results of combination PCA and K-NN

The first thing we notice is that the box plot for PCA is more compressed. This means that the performance lies closer to each other. Although, this seems only to be the case for the upperbound. The lowerbound has not moved and is for both experiment 55%. As a result, the mean accuracy is also slightly lower. Overall, we can say that this setup does not show the results

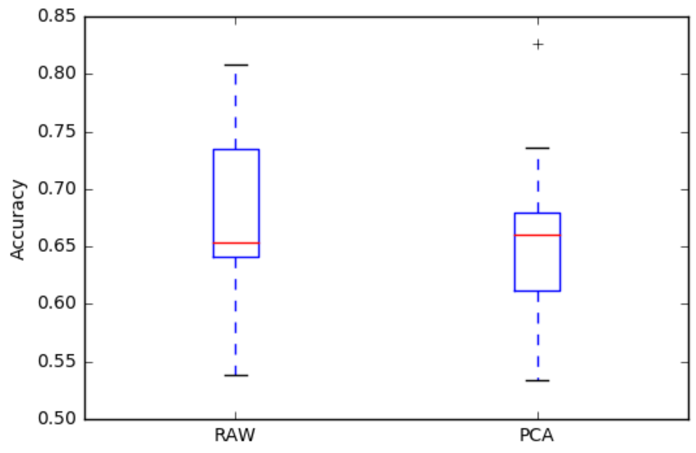


Figure 7.12: Box plot of Accuracy by PCA and RAW

we hoped for. For the next experiment, we still stay with the K-NN classifier, but we are going to look for a non-linear approach by using autoencoders.

### 7.9.3 Convolutional Autoencoders

The next thing we try are autoencoders. An autoencoders is a special variant of a neural network, where the network tries to reconstruct its own input. The interesting thing about autoencoders is that they allow us to extract new features (Vincent et al., 2008) and this is what we will use. We have seen in section 6.3 that an autoencoder first reduces the input to a certain amount of features and than starts to reconstruct the original input, based on these features. The idea of this experiment is to extract the reduced representation and use these features as the representation of a canvas for the K-NN classifier. This reduced representation is located in the most hidden layer, the so called 'bottle neck layer'.

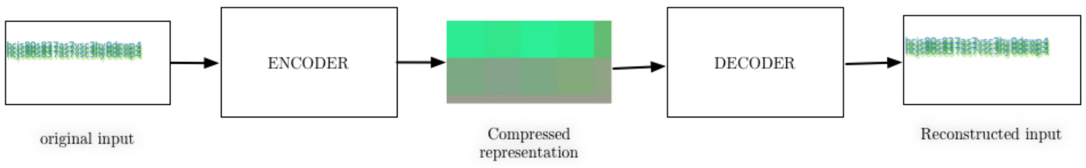


Figure 7.13: Simplified schema of the autoencoder

Since our canvases are images, we can use convolutional layers in our autoencoder. The layout of the autoencoder looks as follows:

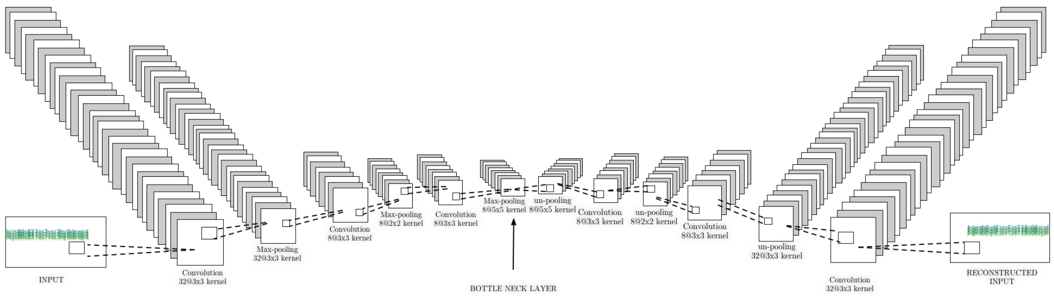


Figure 7.14: Architecture of the autoencoder

We try two approaches for this experiment. With the first approach, we let the autoencoder learn a reconstruction for all canvases from all devices in the training set. The second approach is slightly different. We train the autoencoder only with the canvases from one device. In this way, we can extract a specialised device-specific representation. Both methods get evaluated over 10 runs. In each run, we let the autoencoder train for 100 epochs.

| | **Accuracy** | **Std** |
|---|---|---|
| **General** | 0.62 | 0.03 |
| **Device Specific** | 0.71 | 0.13 |

Table 7.5: Results of combination with autoencoders and K-NN

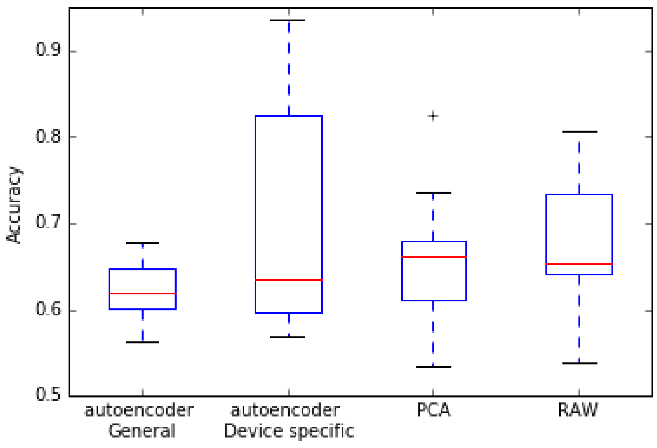


Figure 7.15: Box plot of all the results with K-NN

In table 7.5, we see that the device-specific approach works the best. The idea of a device-specific representation seems to work. We see some higher classification results. Although, the data seems still to vary a lot. Sometimes we have very good results and sometimes bad results. Also the consistent performance of the general autonencoder approach is interesting to see. Based on these results, we say that autoencoders are a good addition to our authentication method. Still, we have not the performance we like to see. Also, we leave the idea of the K-NN classifiers for a canvas based authentication method, since it is rather limited in scaling for large datasets. We believe that we should use the full potential of a neural network. This is why we continue in the next section with convolutional networks.

## 7.10 Convolutional Network

In section 7.9, we experimented with a K-NN classifier and several pre-processing techniques. Mainly autoencoders showed promising results. We build further on these findings by constructing a convolutional network, based on the architecture of the autoencoder in figure 7.14. Convolutional networks are know to perform very well for image classification (Krizhevsky et al., 2012) and we hope by using them to achieve also good result.

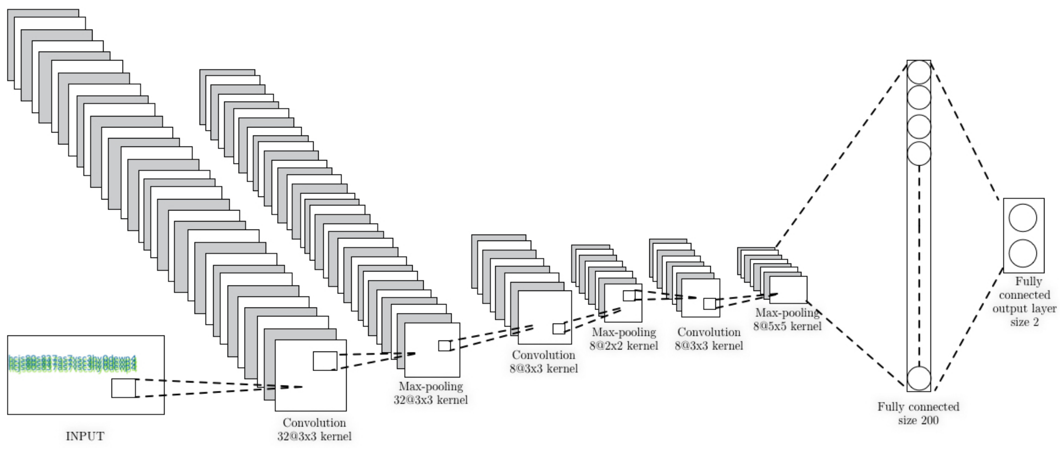


Figure 7.16: Architecture of the Convolutional Network

The convolutional network is constructed by removing the reconstructing layers of the autoencoder and replacing them with a dense layer and an output layer. The size of the dense layer can be anything. 200 seems us a good size, because it's halving the size of the compressed representation in the layer before. As a result, the architecture allows us to do classification.

### 7.10.1 Artificial Simulation

Again, we evaluate the performance of the network, by taking the average of 10 runs. In each run, we train the network for 100 epochs and we partition the dataset in 80/20 for the training and test set. We also normalize the dataset with min-max normalisation. Normalisation is a simple technique, which is shown to be very effective (Al Shalabi et al., 2006). Min-max normalization scales down the feature values to a fixed range of $[0, 1]$. An advantage of the fixed range is a suppression of outliers in the dataset. Especially with neural network, this can influence the performance. Mathematically, we write min-max normalization down as:

$$X_{norm} = \frac{X - X_{min}}{X_{max} - X_{min}} \tag{7.2}$$

where X is the original feature value.

Min-Max normalisation applied to the canvases, results in dividing each pixel value by 255.

| | |
|---|---|
| **Mean Accuracy** | 0.94 |
| **Std** | 0.03 |
| **Loss** | 0.20 |

Table 7.6: Results of Convolution Network

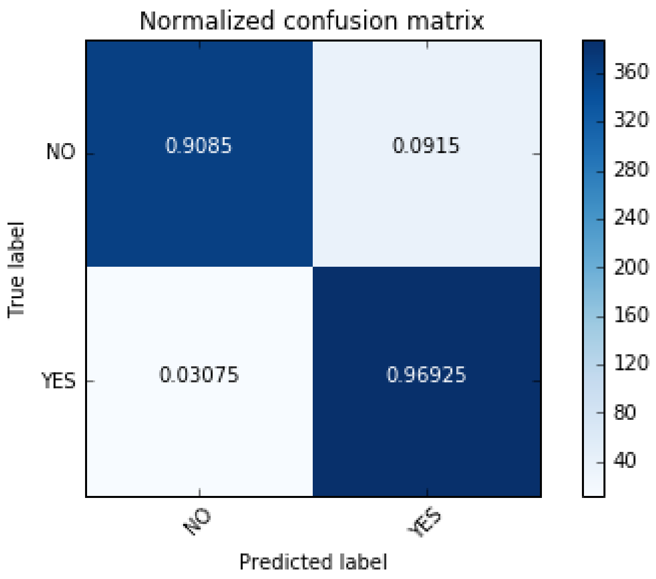


Figure 7.17: Confusion Matrix of Convolutional Network

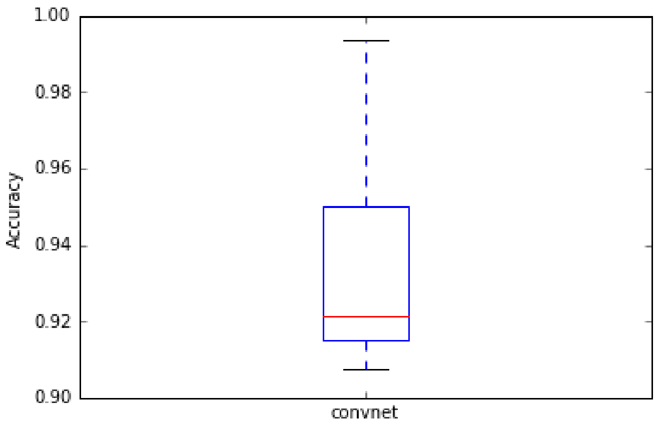


Figure 7.18: Box plot of the accuracy of the Convolutional network

The results are significantly better. The convolutional network clearly outperforms the previous results, with an average accuracy of 94%. Also, it performs well on a constant basis. In box plot 7.18, we see that the lower bound over the 10 runs is around 93%. That's very good, especially if you compare it with the lowerbound of 55% in section 7.9.3. It is also interesting to look at the confusion matrix in figure 7.17. If we want to use this architecture for authentication, it is important to have a low false positive rate. The false positive rate indicates how much unauthorized users are accepted by the system. A false positive rate of 9% is certainly not bad. Of course, we want to have 0%, but overall we are content since the system is developed for weak authentication. Further, figure 7.19 and 7.20 show that the network is able to learn quite fast.

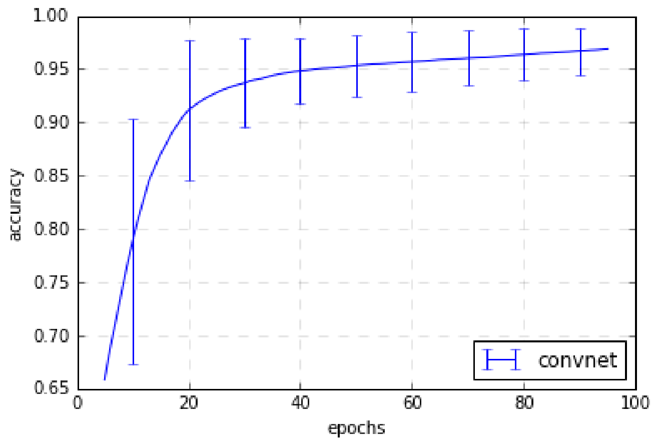


Figure 7.19: Accuracy of the Convolutional Network during the learning

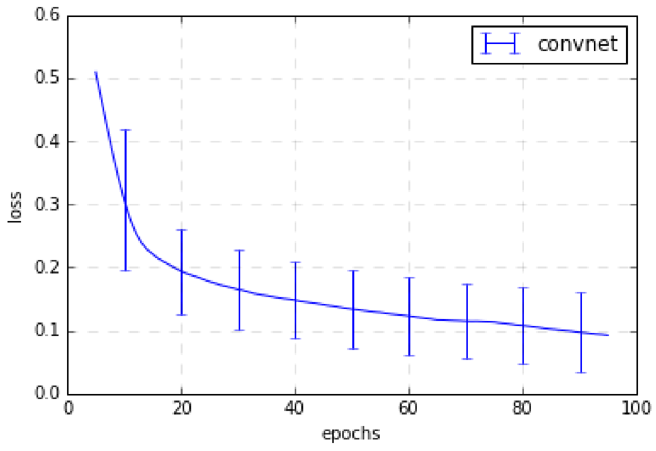


Figure 7.20: Loss of the Convolutional Network during the learning

Based on the results, we say that the convolutional network is a useful algorithm to use for authentication with canvases. But we must be aware that we evaluated this and the previous experiments in an artificial environment. The experimental setups were not a real representation of what the authentication method could encounter in the real world. Let us explain by showing figure 7.21:

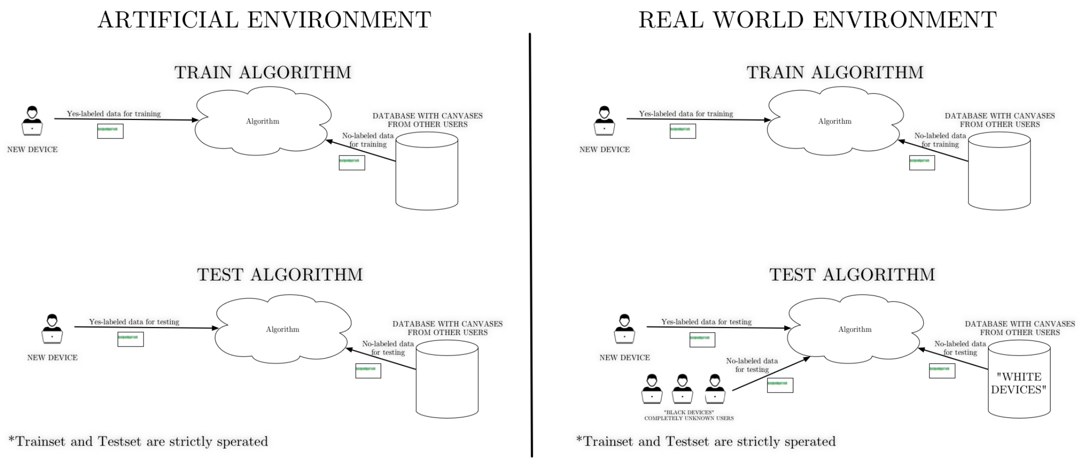

ARTIFICIAL ENVIRONMENT
REAL WORLD ENVIRONMENT
TRAIN ALGORITHM
Yes-labeled data for training
Algorithm
DATABASE WITH CANVASES FROM OTHER USERS
No-labeled data for training
NEW DEVICE
TEST ALGORITHM
Yes-labeled data for testing
No-labeled data for testing
"BLACK DEVICES" COMPLETELY UNKNOWN USERS
"WHITE DEVICES"
*Trainset and Testset are strictly sperated

Figure 7.21: Artificial Environment vs Real World Environment

In the current setup, we did not consider devices that are completely unknown to the system. Let us explain. In section 7.6, we discussed how we constructed the dataset. The 'Yes'-labeled canvases are the canvases from the device we want to learn and the 'No'-labeled canvases are canvases from random other devices in our database. The fact that we use canvases from our database can introduce a bias. Consider the following example: We want to learn to recognize canvases from device $A$. The authentication method constructs the dataset by labelling the canvases from device $A$ with 'Yes' and other randomly selected canvases from other devices in the database with 'No'. Assume that two canvases $s_1$ and $s_2$ are part of the 'No'-labeled canvases and are both originating from the same device $S$. Now in this experiment we split the dataset in 80/20 for training and testing. Assume now that $s_1$ is in the training set and $s_2$ in the test set. Now, If we train the convolutional network with the training set and evaluate it with the test set, our results are slightly biased. The reason is that $s_1$ might contain information that biases the decision on $s_2$. If we put this in the perspective of an actual authentication system, it comes down to the fact that we are only evaluation our authentication system on authorized and unauthorized attempt from users registered to the system. Although, in the real world it might be the case that there are unauthorized attempts of users that are not registered to the system, for example hackers. This is what we will take in consideration in the so called 'Real World environment'. Note that, there is for the artificial and real-world simulation no difference in how we train the convolutional network. The difference lies in the evaluation. In the artificial simulation, we only considered authorized and unauthorized attempts from registered users. In the real world simulation, we evaluate unauthorized attempts from unregistered users.

### 7.10.2 Real-world Simulation

Before we describe the experimental setup for the real world simulation, we introduce the terms **White Device** and **Black Device**. A device is considered a white device, when it is a registered device and it has some canvases that are used as 'No'-labeled canvases for training and testing. A black device is a unregistered device that is used to simulate the unauthorized requests from an unregistered device. For the experiment, we pick 10 black devices from the database. Then we compose a dataset with the remaining devices, following the method described in section 7.6. We apply again a partitioning of 80/20 for the training- and test set and train the algorithm. Next, we test the algorithm. This happens differently than in the artificial environment. Instead of using only the 20% of the dataset, we add extra canvases from the black device to the test set. In total 400 canvases of each black device. This ‘enriched’ test set is used to evaluate the experiment. Again we evaluate the experiment by taking 10 runs and train the convolutional network for 100 epochs.

| | |
|---|---|
| **Mean Accuracy** | 0.93 |
| **Std** | 0.01 |
| **Loss** | 0.25 |

Table 7.7: Results of convolution network in a real world environment

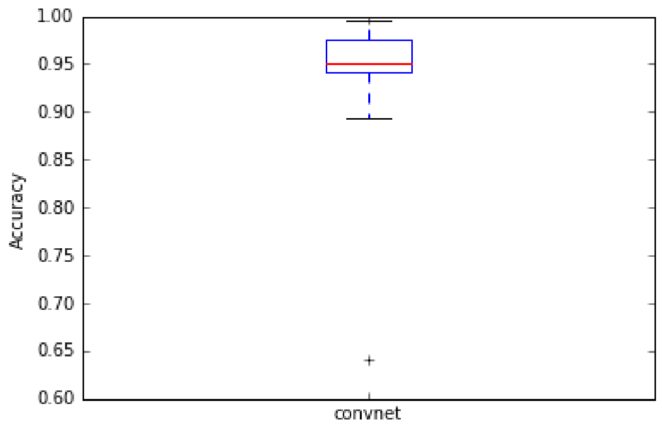


Figure 7.22: Box plot of the accuracy of the Convolutional network in a real world environment

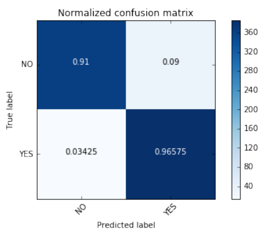


Figure 7.23: Confusion Matrix of the performance by convolutional network in a real world environement for the White Devices

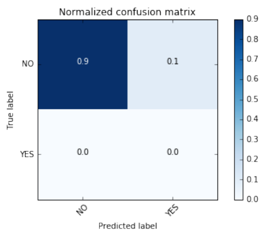


Figure 7.24: Confusion Matrix of the performance by convolutional network in a real world environement for the Black Devices

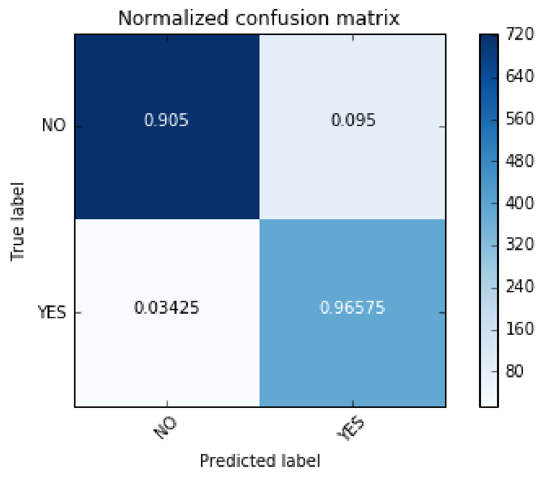


Figure 7.25: Confusion Matrix of the overall performance by the convolutional network in a real world environment

We see a similar result. The accuracy is 1% lower, compared to the artificial simulation, but we see in figure 7.22 that the box plot has a serious outlier. An outlier, where the performance is drastically lower. After checking this run for bugs and looking at the composition of the dataset, nothing special could be found. Although, we found a deviant behaviour in the performance of this run. All the Black devices were classified as false positives. For the white devices is the behaviour similar to the other runs. This confirms our theory of the artificial environment and the introduction of a small bias. Overall, we conclude that this architecture is a good authentication method that has a good performance. Also for a real world environment.

## 7.11 Autoencoders and Semantic Hashing

The encouraging results of autoencoders, inspired us to develop another authentication method. Earlier, we saw that we could learn a compressed representation with authoencoders. Our idea is to force these compressed representations to binary representations and use them for semantic hashing. Salakhutdinov & Hinton (2009) introduced semantic hashing by mapping documents to a memory address in such a way that semantically similar documents are located at nearby addresses. The resembles with our research is that he did this also with autoencoders. An encoded document in his research represented also the memory address of that document. We try to introduce the same idea in the authentication method of this experiment. The authentication method uses an autoencoder to retrieve a compressed representation of every canvas it receives. These compressed representations are seen as memory addresses and allow the system to calculate distances between the canvases. The idea is that canvases that are originating from the same device are mapped to memory addresses that lie close to each other. This also applies the other way around. Canvases that are not originating from the same device are mapped to memory addresses that lie far apart. The authentication method verifies the origin of a canvas by considering a distance-related threshold. The threshold is calculated by using the reference canvases. The reference canvases are the canvases that are sent during the registration of a device to the authentication system and are all originating from the same device. The system will calculate the threshold by considering the mutual distances between the memory addresses of the reference canvases. A example threshold would be the mean distance between two reference canvases. To continue with the example, if the system wants to verify the origin of a canvas $s$, it calculates the mean distance of canvas $s$ with each reference canvas and compares this with the threshold. If the value is smaller than the threshold, canvas $s$ is considered to come from the same device as the reference canvases. If the value is larger than the threshold is considered to originate from another device. Note that, the threshold does not need to be dynamic such as in the example but can be manually configured by the security expert. The threshold defines how the distance between an unknown canvas and the reference canvases should relate. How this is done can be completely decided by the security expert.

In this experiment, we consider the hamming and euclidean distance as two distance measures. We evaluate two types of thresholds. First we use the 'Mean-Threshold'. This is the mean distance between two reference canvases as a threshold. Next, we use as a threshold the variance of the distances between two reference canvases, which we call the 'Variance-Threshold'. The architecture of the autoencoder to convert the canvases to a compressed representation is slightly different than the one in 7.14. We add a Gaussian noise signal just before the bottle neck layer. This will push the activation functions to extreme values and results in binary compressed representation for a canvas. We test also different group sizes. This mean that we use multiple canvases instead of one canvas, to verify the authentication attempt of a device. Each canvas in the group is compared with the reference canvases and the average distance of that group is compared with the threshold. All results are the average of 10 runs. The performance is measured with the accuracy of the authentication system. We test if it accepts the authorization attempts, when the canvases come from the same device and if it denies them when they come from another device. Figure 7.27 illustrates the working of the authentication system. The authentication method in the figure is configured with the Mean-Threshold and $groupsize = 1$. $Groupsize = 1$ indicates that the user authenticates himself only with one canvas to the system. The authentication in figure 7.27 starts with submitting an email and one canvas to the system. The system uses the autoencoder to convert the canvas to a binary compressed representation. Next, the reference canvases from the user's device are queried from the database and the distance value of the canvas is calculated. We have configured the authentication system with the Mean-threshold. So, the distance value of the canvas is equal to the mean distance between the canvas and the reference canvases. The threshold is the mean distance between the reference canvases. If the distance value is smaller than the threshold, the system accepts the authentication. If the distance value is greater than or equal to the threshold, the system denies the authentication.

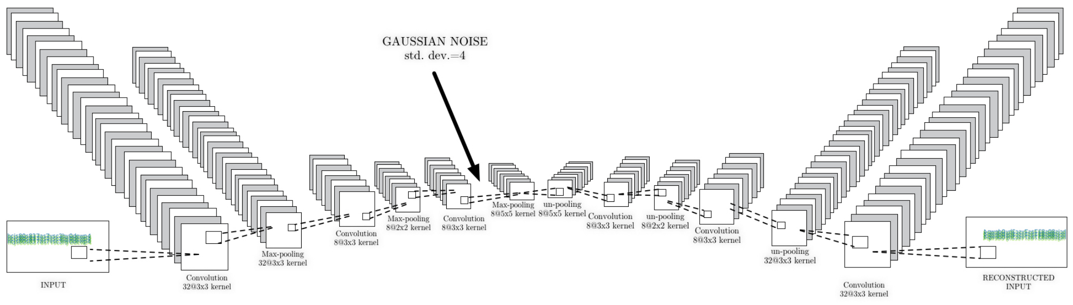


Figure 7.26: Architecture of the autoencoder with noise

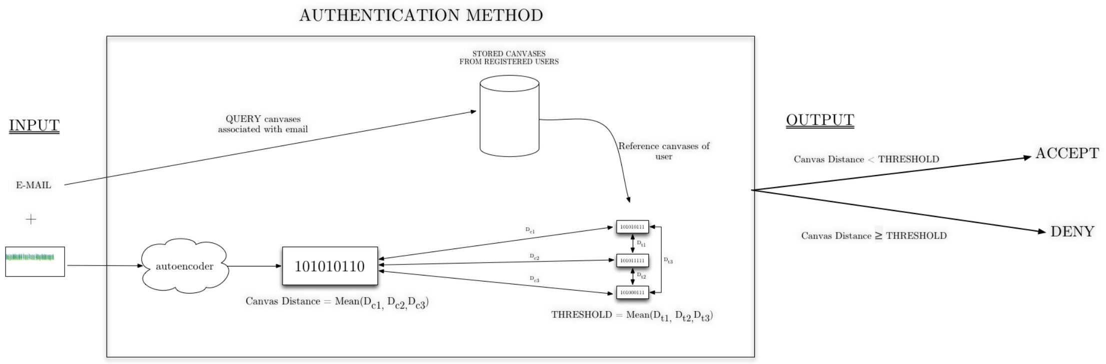

AUTHENTICATION METHOD
INPUT
E-MAIL
+
QUERY canvases associated with email
STORED CANVASES FROM REGISTERED USERS
Reference canvases of user
autoencoder
101010110
Canvas Distance = Mean(Dc1, Dc2, Dc3)
THRESHOLD = Mean(Dt1, Dt2, Dt3)
OUTPUT
Canvas Distance < THRESHOLD
ACCEPT
Canvas Distance ≥ THRESHOLD
DENY

Figure 7.27: Illustration of authentication method with as configuration the Mean-Threshold and groupsize=1.

| **Group size** | **Mean-Threshold** | **Std Mean-Threshold** | **Variance-Threshold** | **Std Variance-Threshold** |
|---|---|---|---|---|
| 1 | 0.73 | 0.15 | 0.52 | 0.02 |
| 20 | 0.52 | 0.23 | 0.31 | 0.33 |
| 40 | 0.54 | 0.23 | 0.22 | 0.29 |
| 50 | 0.5 | 0.2 | 0.36 | 0.36 |
| 100 | 0.57 | 0.23 | 0.36 | 0.37 |
| 400 | 0.61 | 0.24 | 0.44 | 0.39 |

Table 7.8: Accuracy on all authentication attempts with the euclidean distance as distance measure

| Group size | Mean-Threshold | Std Mean-Threshold | Variance-Threshold | Std Variance-Threshold |
|---|---|---|---|---|
| 1 | 0.55 | 0.0 | 0.03 | 0.05 |
| 20 | 0.74 | 0.0 | 0.31 | 0.21 |
| 40 | 0.78 | 0.0 | 0.24 | 0.19 |
| 50 | 0.8 | 0.0 | 0.33 | 0.22 |
| 100 | 0.84 | 0.0 | 0.33 | 0.24 |
| 400 | 0.82 | 0.0 | 0.29 | 0.29 |

Table 7.9: Accuracy on unauthorized attempts with the euclidean distance as distance measure

| Group size | Mean-Threshold | Std Mean-Threshold | Variance-Threshold | Std Variance-Threshold |
|---|---|---|---|---|
| 1 | 0.9 | 0.3 | 1.0 | 0.0 |
| 20 | 0.3 | 0.46 | 0.3 | 0.46 |
| 40 | 0.3 | 0.46 | 0.2 | 0.4 |
| 50 | 0.2 | 0.4 | 0.4 | 0.49 |
| 100 | 0.3 | 0.46 | 0.4 | 0.49 |
| 400 | 0.4 | 0.49 | 0.6 | 0.49 |

Table 7.10: Accuracy on all authorized attempts with the euclidean distance as distance measure

| Group size | Mean-Threshold | Std Mean-Threshold | Variance-Threshold | Std Variance-Threshold |
|---|---|---|---|---|
| 1 | 0.52 | 0.25 | 0.5 | 0.27 |
| 20 | 0.67 | 0.24 | 0.48 | 0.39 |
| 40 | 0.64 | 0.25 | 0.46 | 0.38 |
| 50 | 0.75 | 0.23 | 0.41 | 0.36 |
| 100 | 0.77 | 0.23 | 0.41 | 0.35 |
| 400 | 0.56 | 0.23 | 0.62 | 0.37 |

Table 7.11: Accuracy on all attempts with the hamming distance as distance measure

| Group size | Mean-Threshold | Std Mean-Threshold | Variance-Threshold | Std Variance-Threshold |
|---|---|---|---|---|
| 1 | 0.55 | 0.0 | 0.19 | 0.14 |
| 20 | 0.74 | 0.0 | 0.46 | 0.29 |
| 40 | 0.78 | 0.0 | 0.51 | 0.27 |
| 50 | 0.8 | 0.0 | 0.53 | 0.25 |
| 100 | 0.84 | 0.0 | 0.52 | 0.25 |
| 400 | 0.82 | 0.0 | 0.54 | 0.29 |

Table 7.12: Accuracy on unauthorized attempts with with the hamming distance as distance measure

| Group size | Mean-Threshold | Std Mean-Threshold | Variance-Threshold | Std Variance-Threshold |
|---|---|---|---|---|
| 1 | 0.5 | 0.5 | 0.8 | 0.4 |
| 20 | 0.6 | 0.49 | 0.5 | 0.5 |
| 40 | 0.5 | 0.5 | 0.4 | 0.49 |
| 50 | 0.7 | 0.46 | 0.3 | 0.46 |
| 100 | 0.7 | 0.46 | 0.3 | 0.46 |
| 400 | 0.3 | 0.46 | 0.7 | 0.46 |

Table 7.13: Accuracy on authorized attempts with the hamming distance

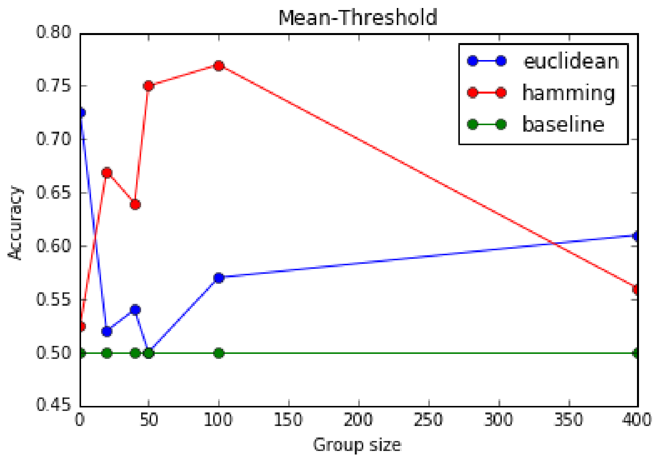


Figure 7.28: Accuracy of the authentication system with the Mean-Threshold configuration

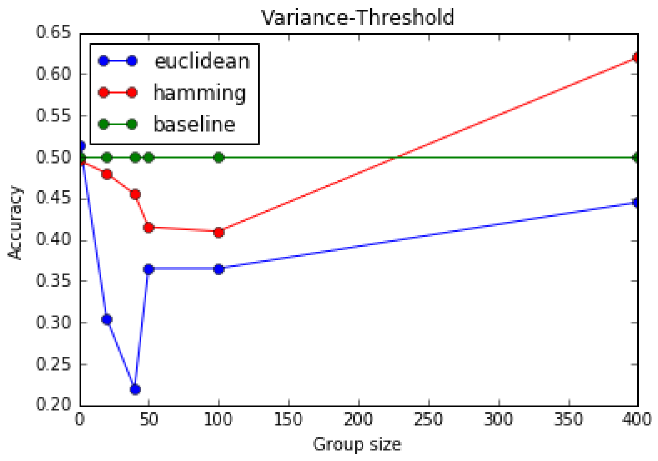


Figure 7.29: Accuracy of the authentication system with the Variance-Threshold configuration

From table 7.8 to table 7.10, we present the results of the authentication method with as distance measure the euclidean distance. From table 7.11 to table 7.13, we present the results of the authentication method with as distance measure the hamming distance. The tables 7.8 and 7.11 show the results of the authentication method for all the authentication attempts. We speak of an authentication attempt, when a user tries to authenticate himself to the system. We consider an attempt unauthorized, when a user tries to authenticate as someone else. When the user tries to authenticate as himself, we speak of a authorized attempt. The tables 7.10 and 7.13 show both the results for authorized attempts and the tables 7.9 and 7.12 show the results for unauthorizated attempts.

In each table, we show every time the accuracy and standard deviation of the accuracy for the Mean-Threshold and the Variance-Threshold. As we said earlier, the Mean-threshold represents a configuration where a threshold is calculated by taking the mean distance between two reference canvases. the Variance-threshold is a configuration where the threshold is calculated by taking the variance of the distance between the reference canvases.

Let us consider the overall results in tables 7.11 and 7.13. We plotted these tables in figures 7.28 and 7.29 to get an overview of the results. From the plots, we can derive several things. First of all, the hamming distance seems to be the best distance measure. For both threshold configurations, the hamming distance performs mainly better. We say that the hamming distance is a more fitting distance measure for the binary compressed representation of the canvases. Further, we see that the Mean-threshold configuration achieved the highest accuracy with an overall accuracy of 77% for a group size equal to 100. If we take a closer look at the accuracy in the Mean-Threshold column for the authorized (tables 7.10 and 7.13) and unauthorized attempts (tables 7.9 and 7.12) , we see that the difference in performance between both distance measures is caused by the performance on the authorized attempts. The performance on unauthorized attempts is identical for both distance measures, but the accuracy on authorized attempts is higher for the hamming distance as distance measure. This means that the hamming distance is better in measuring the relation that canvases from the same device are lying closer to each other than canvases the are originating from different devices. In Figure 7.29, we see the Variance-Threshold is performing bad. It performs for all group sizes equally or under the baseline. The only excepting is $groupsize = 400$ with the hamming distance, where we see an accuracy of 62%, but this still doesn't weigh up against the performance of the configuration with the Mean-Threshold.

Other metrics to consider the performance of the algorithm are the false positieve and false negative rate. The false positive rate is the percentage of users that try to authenticate to the system as someone else and get accepted. The false negatieve rate is the percentage of users that gets unfairly denied by the authentication system. The perfect authentication system would have a false positive and false negatieve rate of 0%. This is not the case with our authentication system, but we should aim to have the lowest percentage possible. If we consider the tables from 7.14 to 7.17, we see that again that the authentication system with the hamming distance as distance measure and the Mean-threshold is performing the best. With the best resuls for $groupsize = 100$, where we have a false positive rate of 18% and a false negative rate of 30%.

| **Group size** | **False Positive Rate** | **False Negative Rate** |
|---|---|---|
| 1 | 0,45 | 0,1 |
| 20 | 0,26 | 0,7 |
| 40 | 0,22 | 0,7 |
| 50 | 0,2 | 0,8 |
| 100 | 0,16 | 0,7 |
| 400 | 0,18 | 0,6 |

Table 7.14: False Positive and False Negatieve Rate of an authentication system with the Euclidean distance as distance measure and the Mean-Threshold.

| **Group size** | **False Positive Rate** | **False Negatieve Rate** |
|---|---|---|
| 1 | 0,97 | 0 |
| 20 | 0,69 | 0,7 |
| 40 | 0,76 | 0,8 |
| 50 | 0,67 | 0,6 |
| 100 | 0,67 | 0,6 |
| 400 | 0,71 | 0,4 |

Table 7.15: False Positive and False Negatieve Rate of an authentication system with the euclidean distance as distance measure and the Variance-Threshold.

| **Group size** | **False Negative Rate** | **False Positive Rate** |
|---|---|---|
| 1 | 0,5 | 0,45 |
| 20 | 0,4 | 0,26 |
| 40 | 0,5 | 0,22 |
| 50 | 0,3 | 0,2 |
| 100 | 0,3 | 0,16 |
| 400 | 0,7 | 0,18 |

Table 7.16: False Positive and False Negatieve Rate of an authentication system with the hamming distance as distance measure and the Mean-Threshold.

| **Group size** | **False Positive Rate** | **False Negative Rate** |
|---|---|---|
| 1 | 0,81 | 0,2 |
| 20 | 0,54 | 0,5 |
| 40 | 0,49 | 0,6 |
| 50 | 0,47 | 0,7 |
| 100 | 0,48 | 0,7 |
| 400 | 0,46 | 0,3 |

Table 7.17: False Positive and False Negatieve Rate of an authentication system with the hamming distance as distance measure and the Variance-Threshold.

At last, we included in figure 7.30 a few examples of the reconstruction by the autoencoder in the authentication system. As we can see, the reconstruction looks very similar for all inputs. It is clear that the autoencoder is not able to make the reconstruction perfectly.

As a conclusion for this authentication method, we can say that it shows promising results and has a lot of potential for further research. Note that, the two threshold configurations, the Mean-Threshold and Variance-Threshold, are chosen by us to evaluate the authentication method. It is possible to replace this with any other threshold configuration. This is a free choice that can be made by the security expert, managing the authentication system.

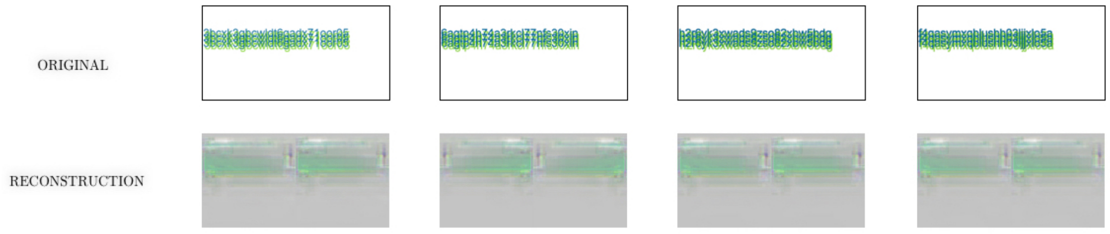


Figure 7.30: Original inputs and reconstructions by autoencoder

## 7.12 Discussion

In the previous sections, we evaluated several authentication methods. First we experimented with K-NN classifier in section 7.9. Authentication methods with this algorithm showed no good results, so we continued with more advanced authencation methods, based on deep learning. In section 7.10 we made use of a convolutional network for our authentication method, which showed very good results with an accuracy of 93% and an acceptable false positive rate. On the other hand in section 7.11, we constructed another promising method, where we used autoencoders for semantic hashing. This method showed less good results than the convolutional network, with for his optimal configuration an accuracy of 77%. Although, this authentication method has much more flexibility than the convolutional network. Which method should we choose? Let us elaborate this. If we have to base ourselves on the performance of the authentication methods, we would choose the convolutional network. But, the deployment of a convolutional network in a live authentication system is less interesting. The reason is that for every new device, a new convolutional network needs to be trained. Considering the runtime of the experiments, this can take a long time. Alternatively, if we would use the authentication method with the autoencoder and semantic hashing, this would not be needed. We only need to train the autoencoder once. After that, the same autoencoder is used everytime to convert the canvases to a compressed representation. So, if we would add a new device to the system, it only has to calculate the compressed representation. This take significanly less time than training the algorithm again. Also, the autencoder offers much more flexibility for the security expert, managing the authentication system. The fact that he has the free choice on how to configure the threshold and group size, gives him the possibilty to define his own policy.

In general, the authentication method with convolutional networks offers us a higher perfomance, but is less generic and flexible than the authentication method, based on an autoencoder and semantic hashing.

# Chapter 8

# Conclusion and Future Work

In this chapter, we summarize the main results and key observations of this thesis. To end, we provide an overview of other opportunities with regard to future research.

## 8.1 Conclusion

Recent developments in the research towards canvases has motivated us to develop a new authentication method, based on canvases. Nowadays, canvases are only used for simple tracking methods. Currently, these tracking methods are not complex enough to use for authentication. We developed several methods, based on machine learning techniques and canvases, that have the complexity to allow authentication. This was a very specific research question and there was no research available. Mainly, the inspiration for this research was drawn from image classification, a similar research domain.

The goal of this thesis was to develop a seamless authentication method, based on canvases and machine learning techniques and intended for weak authentication. The outline of the authentication method was as follows. First, the user registers a device by feeding an email and an amount of random canvases to the authentication system. After the registration, the user should be able to identify himself with the registered device by sending again his email and an amount of random canvases. The drawing method, chosen during this research, is very simple. The method draws a random string multiple times on a fixed position. This allowed us to observe all the possibilties of a canvas drawing in its simplest form. Also, we tried to gradually increase the complexity of the methods we experimented with. This gave use the knowledge to incrementally improve our authentication method. In general, we developed three authentication methods.

The first authentication method used a K-NN Classifier, a very intuitive and easy to understand algorithm. It allowed us to see the possibilities with the canvases. It was clear that this method was not suitable as an authentication method. It showed bad performances and scaled very bad for large dataset. Despite the bad results, the method inspired us for new approaches in the next authentication methods. We used convolutional autoencoders to retrieve a compressed representation of the canvases. This had a positive impact on the results of the method. It achieved more consistent and better results. This inspired use for the second developed authentication method with convolutional networks.

The second authentication method was fully based on convolutional networks. The authentication system trained for each device a new convolutional network and used the prediction of the neural network to predict if a canvas comes from a certain device. This method is the best performing method of all the developed methods. It has an accuracy of 93% in a real world environment and has a low false positive rate of 9%. The drawback of this method is the fact that it needs to train a new convolutional device, every time a new device is added to the authentication system. This can be time expensive.

Alternatively, we developed the third and last authentication method. This method used an autoencoder to semantically hash the canvases. The idea was that the autoencoder maps the canvases to a binary compressed representation in a way that canvases from the same device have a small distance between their representations and canvases from different devices have a large distance between their representations. The authentication method used a threshold related to the distance, to decide whether or not the canvases originated from the same device. An advantage of this system is the flexibility. It allows the security expert to choose the threshold freely. In this way, the security expert can define his own policy for the authentication system. Another advantage is the fact that is is a generic authentication method. When a new device is added to the system, no new training of the autoencoder is required. The autoencoder can generate a binary compressed representation for every new device, when it is only trained once. We evaluated this method by considering several group sizes, two distance measures, namely the hamming distance and euclidean distance and two threshold configurations. As a result, we saw that the hamming distance was the best distance measure for the system and the Mean-threshold, together with a group size equal to 100, results in a good performance of 77%. The Mean-threshold was defined as the mean distance between two canvases from the same device.

Overall we can say that we succeed in our goal. We have developed and evaluated several authentication methods, which could serve for weak authentication. Further, we have shown that we can identify devices, based on canvases and their render differences. More over, we demonstrated that it is possible to pick up these difference by using simple and random canvas drawings of several devices. This makes canvases also a very interesting subject for future research.

## 8.2 Future Work

- Tuning of the architectures: The autoencoders and convolutional networks require a lot of decision making with regard the hyperparameters and the architecture. All the deep learning methods in our experiments have the same architecture and configuration. It would be interesting to see for future work how other architectures or configurations would perform compared to ours. An example would be the use of stacked or denoising autoencoders, instead of our normal autoencoder.

- Advanced canvases: In this research we considered only a very simple drawing technique to represent the canvases. It would be interesting to see how our methods would perform with more complex canvases. A nice addition to the canvas are for example shapes or emoijs. Especially emoijs, they introduce immediately a difference, since it is drawn differently for each operating system. This could help the algorithms to make a more accurate decision.

- Security of the authentication method: The security of the authentication method is not discussed in this thesis. Nevertheless, it is important to make an assessment of the security of the method. How can the authentication be evaded and is it possible to fake someone's identity? These are questions that could be asked in future research.

- Large scale evaluation: We evaluated this method with a rather limited dataset. In the real world there is much more variation. An large scale evaluation of the authentication method could give us more insight on the entropy of the canvases and the performances of our systems.

# References

Acar, G., Eubank, C., Englehardt, S., Juarez, M., Narayanan, A., & Diaz, C. (2014). The web never forgets: Persistent tracking mechanisms in the wild. In *Proceedings of the 2014 acm sigsac conference on computer and communications security* (pp. 674–689). November 3-7, 2014, Scottsdale, AZ, USA.

Al Shalabi, L., Shaaban, Z., & Kasasbeh, B. (2006). Data mining: A preprocessing engine. *Journal of Computer Science*, *2*(9), 735–739.

Attwell, D., & Laughlin, S. B. (2001). An energy budget for signaling in the grey matter of the brain. *Journal of Cerebral Blood Flow & Metabolism*, *21*(10), 1133–1145.

Ayenson, M. D., Wambach, D. J., Soltani, A., Good, N., & Hoofnagle, C. J. (2011). Flash cookies and privacy ii: Now with html5 and etag respawning.

Baker, S. (2010, November). *Re: [public webgl] about the vendor, renderer, and version strings. public webgl mailing list.* Retrieved 18/05/2017, from `https://www.khronos.org/webgl/public-mailing-list/archives/1011/msg00221.html`

Boiman, O., Shechtman, E., & Irani, M. (2008). In defense of nearest-neighbor based image classification. In *Proceedings of ieee conference on computer vision and pattern recognition 2008* (pp. 1–8). June 23-28, 2008, Anchorage, Alaska.

Claude, S., & I., W. G. (2011). *Encyclopedia of machine learning.* Berlin: Springer.

Consortium, W. W. W. (2017). *Number of internet users.* Retrieved 17-04-2017, from `http://www.internetlivestats.com/internet-users/`

Cottrell, G. W. (1990). Extracting features from faces using compression networks: Face, identity, emotion and gender recognition using holons. In D. Touretzky (Ed.), *Connectionist models: proceedings of the 1990 summer school.* San Mateo, CA: Morgan Kaufmann Publishers Inc.

Crone, S. F., Lessmann, S., & Stahlbock, R. (2006). The impact of preprocessing on data mining: An evaluation of classifier sensitivity in direct marketing. *European Journal of Operational Research*, *173*(3), 781–800.

Dean, J. (2016). *Deep learning for building intelligent computer systems.* Retrieved 3/05/2017, from `https://www.youtube.com/watch?v=QSaZGT4-6EY`

Dictionary, O. (2017). *Authentication.* Online. Retrieved 18/05/2017, from `https://en.oxforddictionaries.com/definition/authentication`

Duchi, J., Hazan, E., & Singer, Y. (2011). Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, *12*(Jul), 2121–2159.

Friedman, J., Hastie, T., & Tibshirani, R. (2001). *The elements of statistical learning* (Vol. 1). Berlin: Springer.

Geoffrey Hinton, K. S., Nitsh Srivastava. (2012). *Lecture 6a overview of mini-batch gradient descent.* Retrieved 18/05/2017, from `http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf`

Glorot, X., Bordes, A., & Bengio, Y. (2011). Deep sparse rectifier neural networks. In *Proceedings of the 14th international conference on artificial intelligence and statistics* (pp. 315–323). April 11-13, 2011, Ft. Lauderdale, FL, USA.

Haykin, S. (2004). *Neural networks: A comprehensive foundation* (Vol. 2). New Jersey: Prentice Hall.

Jacob, B. (2010, December). *Re: [public webgl] information leakage and the extension regisrty. public webgl mailing list.* Retrieved 18/05/2017, from `http://www.khronos.org/webgl/public-mailing-list/archives/1012/msg00083.html`

Kohavi, R., et al. (1995). A study of cross-validation and bootstrap for accuracy estimation and model selection. In *Proceedings of the 14th int. joint conf. artificial intelligence* (pp. 338–345). August 20 - 25, 1995, Montreal, Quebec, Canada.

Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In P. Bartlett (Ed.), *Proceedings of the 26th annual conference on neural information processing systems* (pp. 1097–1105). December 3-6, 2012, Lake Tahoe, Nevada, USA.

Lang, K. J., Waibel, A. H., & Hinton, G. E. (1990). A time-delay neural network architecture for isolated word recognition. In *Neural networks* (Vol. 3, pp. 23–43). Oxford, UK: Elsevier.

Larochelle, H., Erhan, D., Courville, A., Bergstra, J., & Bengio, Y. (2007). An empirical evaluation of deep architectures on problems with many factors of variation. In *Proceedings of the 24th international conference on machine learning* (pp. 473–480). June 20 - 24, 2007 , Corvallis, OR, USA.

LeCun, Y., Bengio, Y., & Hinton, G. (2015). Deep learning. *Nature*, *521*(7553), 436–444.

LeCun, Y., Bengio, Y., et al. (1995). Convolutional networks for images, speech, and time series. *The handbook of brain theory and neural networks*, *3361*(10), 1995.

LeCun, Y., Boser, B., Denker, J. S., Henderson, D., Howard, R. E., Hubbard, W., & Jackel, L. D. (1989). Backpropagation applied to handwritten zip code recognition. *Neural computation*, *1*(4), 541–551.

LeCun, Y., et al. (1989). Generalization and network design strategies. In F. F. L. S. R. Pfeifer Z. Schreter (Ed.), *Connectionism in perspective* (pp. 143–155). North-Holland, Amsterdam: Elsevier.

LeCun, Y. A., Bottou, L., Orr, G. B., & Müller, K.-R. (2012). Efficient backprop. In *Neural networks: Tricks of the trade* (pp. 9–48). Berlin: Springer.

Lee, H., Ekanadham, C., & Ng, A. Y. (2008). Sparse deep belief net model for visual area v2. In Y. S. S. R. J.C. Platt D. Koller (Ed.), *Advances in neural information processing systems 20 (nips'07)* (pp. 873–880). Cambridge, MA: MIT Press.

Lennie, P. (2003). The cost of cortical computation. *Current biology*, *13*(6), 493–497.

Ludwig, J. (2013). Image convolution. *Portland State University.*

Mitchell, T. M. (1997). *Machine learning.* Boston, MA: McGraw-Hill.

Mokrohuz, A., & Kazymyr, V. (2014). Information technologies of biometric security for mobile devices.

Mowery, K., & Shacham, H. (2012). Pixel perfect: Fingerprinting canvas in html5. In M. Fredrikson (Ed.), *Proceedings of w2sp 2012.* May 24, 2012, The Westin St. Francis Hotel, San Francisco: IEEE Computer Society.

Moyer, C. (2016). How google's alphago beat a go world champion. *The Atlantic, March*, *28*. Retrieved https://www.theatlantic.com/technology/archive/2016/03/the-invisible-opponent/475611/, from 2/05/2017

Ng, A. (2011). Sparse autoencoder. *CS294A Lecture notes*, *72*(2011), 1–19.

Ng, A. (2015). *What data scientists should know about deep learning.* Retrieved 3/05/2017, from https://www.youtube.com/watch?v=O0VN0pGgBZM

Ranzato, M., Poultney, C., Chopra, S., & LeCun, Y. (2006). Efficient learning of sparse representations with an energy-based model. In *Proceedings of the 19th international conference on neural information processing systems* (pp. 1137–1144).

Rosenblatt, F. (1958). The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological review*, *65*(6), 386.

Rouse, M. (2014, December). *What is user authentication.* Online. Retrieved 18/05/2017, from http://searchsecurity.techtarget.com/definition/user-authentication

Russell, S., & Norvig, P. (1995). *Artificial intelligence: A modern approach.* Englewood Cliffs, New Jersey: Prentice-Hall.

Saevanee, H., & Bhatarakosol, P. (2008). User authentication using combination of behavioral biometrics over the touchpad acting like touch screen of mobile device. In *Computer and electrical engineering, 2008. iccee 2008. international conference on* (pp. 82–86).

Salakhutdinov, R., & Hinton, G. (2009). Semantic hashing. *International Journal of Approximate Reasoning*, *50*(7), 969–978.

Samuel, A. L. (1959). Some studies in machine learning using the game of checkers. *IBM Journal of research and development*, *3*(3), 210–229.

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Van Den Driessche, G., ... others (2016). Mastering the game of go with deep neural networks and tree search. *Nature*, *529*(7587), 484–489.

Simard, P. Y., Le Cun, Y. A., Denker, J. S., & Victorri, B. (2000). Transformation invariance in pattern recognition: Tangent distance and propagation. *International Journal of Imaging Systems and Technology*, *11*(3), 181–197.

Taigman, Y., Yang, M., Ranzato, M., & Wolf, L. (2014). Deepface: Closing the gap to human-level performance in face verification. In *Proceedings of the ieee conference on computer vision and pattern recognition* (pp. 1701–1708). June 23-28, 2014, Columbus Convention Center, Columbus, OH, USA.

Vincent, P., Larochelle, H., Bengio, Y., & Manzagol, P.-A. (2008). Extracting and composing robust features with denoising autoencoders. In *Proceedings of the 25th international conference on machine learning* (pp. 1096–1103). July 5-9, 2008, Helsinki, Finland.

Vincent, P., Larochelle, H., Lajoie, I., Bengio, Y., & Manzagol, P.-A. (2010). Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion. *Journal of Machine Learning Research*, *11*(Dec), 3371–3408.

Zeiler, M. D. (2012). Adadelta: an adaptive learning rate method. technical report, arxiv 1212.5701.