

# HoGent

**BUSINESS AND  
INFORMATION  
MANAGEMENT**

Professional Bachelor in Applied Computer Science  
Academic year 2012-2013

---

## **Solving CAPTCHA using neural networks**

---

Submitted June 6<sup>th</sup> 2013

*Student:*  
Pieter Van Eeckhout

*Mentor:*  
Johan Van Schoor



HoGent Business & Information Management  
Professional Bachelor in Applied Computer Science  
Academic year 2012-2013

---

# **Solving CAPTCHA using neural networks**

---

Submitted June 6<sup>th</sup> 2013

*Student:*  
Pieter Van Eeckhout

*Mentor:*  
Johan Van Schoor



# Contents

|          |  |           |
|----------|--|-----------|
| <b>1</b> | <b>Solving CAPTCHA using neural networks</b>             | <b>5</b>  |
| <b>2</b> | <b>Premise and research questions</b>                    | <b>7</b>  |
| 2.1      | Premise . . . . .  | 7         |
| 2.2      | Research questions . . . . .                             | 7         |
| <b>3</b> | <b>Methodology</b>                                       | <b>9</b>  |
| 3.0.1    | Research philosophy . . . . .                            | 9         |
| 3.0.2    | Approach . . . . .                                       | 9         |
| <b>4</b> | <b>Corpus</b>  | <b>11</b> |
| 4.1      | CAPTCHA . . . . .  | 11        |
| 4.1.1    | CAPTCHA, an explanation . . . . .                        | 11        |
| 4.1.2    | The history of CAPTCHA . . . . .                         | 12        |
| 4.1.3    | Types of CAPTCHA . . . . .                               | 12        |
| 4.1.4    | Data extraction . . . . .                                | 13        |
| 4.1.5    | The future of CAPTCHA . . . . .                          | 14        |
| 4.2      | Neural Networks . . . . .                                | 17        |
| 4.2.1    | The concept of time . . . . .                            | 17        |
| 4.2.2    | The components of an artificial neural network . . . . . | 17        |
| 4.2.3    | Neural network topologies . . . . .                      | 23        |
| 4.2.4    | Neural network learning . . . . .                        | 28        |
| 4.3      | Neural networks for pattern recognition . . . . .        | 34        |
| 4.3.1    | Perceptron networks . . . . .                            | 35        |
| 4.3.2    | Hopfield networks . . . . .                              | 37        |
| 4.3.3    | Self-organizing maps . . . . .                           | 39        |
| 4.4      | Implementation . . . . .                                 | 42        |
| 4.4.1    | Captcha builder . . . . .                                | 42        |
| 4.4.2    | Neural networks . . . . .                                | 44        |
| <b>5</b> | <b>Conclusion</b>  | <b>49</b> |

|          |                                       |           |
|----------|---------------------------------------|-----------|
| <b>A</b> | <b>Code</b>                           | <b>51</b> |
| A.1      | ColorRangeContainer.java . . . . .    | 51        |
| A.2      | Captcha.java . . . . .                | 55        |
| A.3      | ImageToArray.java . . . . .           | 57        |
| A.4      | TrainingSet.java . . . . .            | 60        |
| A.5      | EncogHopfieldNetwork.java . . . . .   | 63        |
| A.6      | EncogPerceptronNetwork.java . . . . . | 65        |
| A.7      | EncogKohonenNetwork.java . . . . .    | 68        |
| <b>B</b> | <b>Images</b>                         | <b>73</b> |
| <b>C</b> | <b>Tables</b>                         | <b>82</b> |
| <b>D</b> | <b>CD</b>                             | <b>93</b> |

## **Abstract**

This thesis is divided into several parts, each part covering one area of the research subject. The thesis starts out by researching what CAPTCHAs are, why they were developed and if there are downsides to using the CAPTCHA system. After that the thesis will focus on what neural networks are and how they function. As a last part an experimental implementation of a few selected networks will be discussed. The majority of this thesis is a literature study, which leads up to the experimental phase. The major conclusion of this thesis is that neural networks are capable of recognizing patterns, if configured correctly, and that they can be used for solving CAPTCHA tests. However, it should be noted that this usage is not economically valid outside of an academic setting. The CAPTCHA generation methods are too volatile to be able to implement a 'one-solves-all' solution.





# Preamble

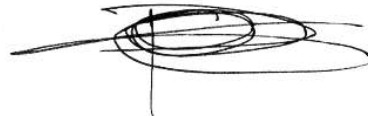
First, dear reader, I would like to thank you for taking the time to read this thesis. Without an audience this entire endeavour would not mean as much as it does right now, with you reading its results. I believe this is because I would like my life not to go unnoticed. So if this thesis helps, or influences you in any way, then this work has gained more meaning.

Second I would like to thank the following people who have made it possible for me to arrive at this point. Special thanks and mentions go to:

- my parents, for supporting me, giving me the opportunity and supplying the means for me to pursue my academic career.
- my girlfriend, Anne Charlotte Magdaraog Mendoza. Because she has helped me countless times through the rough spots. Not once did she complain about the time consuming job of writing this work.
- my good friends, willing proof readers and content critics: Wouter Dekens, Patrick Van Brussel and Thijs van der Burgt.
- Johan Van Schoor and Bert Van Vreckem for the support, organisation, guidance and feedback.

Bear in mind that this is not an exclusive list. Finally I would like to thank all the other people who are not mentioned by name: such as the teaching and support staff at University College Ghent.

Ghent BELGIUM, June 2013

A handwritten signature in black ink, consisting of several overlapping loops and a long horizontal stroke extending to the left.

Pieter Van Eeckhout



# Chapter 1

## Solving CAPTCHA using neural networks

**The target audience.** This thesis was written with an audience in mind that already has some technical understanding of computers and how they operate on hardware level (processor etc.). If you feel that your current knowledge is insufficient, or just want to read up some more, then I refer you to the "How Computers Work - Processor and Main Memory" [Young, 2001] e-book.

**The history of SPAM.** Ever since the internet found its way into our daily life, there have been people out there who don't always have other people's best interest in mind. I am referring to spammers, people aiming to advertise their product, services, etc ... in an aggressive manner. The methods of advertising include but are not limited to:

- Sending bulk emails without the recipients permission (SPAM).
- Posting irrelevant links and information on fora and various social media.
- Flooding chat channels with their links and information.

These emails, posts and messages inconvenience the end-users, requiring time to filter out the junk. The economic costs of SPAM has led to a decrease in the Japanese GDP by 500 billion Yen (3.78 billion Euro) in 2004 and was projected to reach a decrease of 1% of the total GDP by 2010 unless adequate countermeasures were taken [Ukai and Takemura, 2007]. [Khong, 2004] researched the economic arguments for regulating junk mails and the efficiency of these regulations.

**Birth of CAPTCHA.** The two previously mentioned researches signify the importance and impact of SPAM on our daily life. The users of the internet quickly

tried to implement methods to prevent spammers from spreading their advertisements to the masses. Several prevention and detection methods and systems were developed successfully. These methods and mechanisms range from hidden text to invalid HTML tags, all used to confuse and interrupt automated programs. One of the methods developed to prevent SPAM is a CAPTCHA test. CAPTCHA is an acronym based on the word "capture" and stands for 'Completely Automated Public Turing test to tell Computers and Humans Apart'. An attempt to trademark the term was made by Carnegie Mellon University on 15 October 2004, but the application was eventually dropped on 12 April 2008

**Spammers fight back.** All these prevention and detection methods did not stop the spammers from trying to reach an audience as large as possible. The spammers rely on a large target audience because of the return rates being as low as 0.0023% [Cobb, 2003]. The spammers started to device ways to circumvent or break the existing systems in order to reach a large enough audience. One of these methods is solving CAPTCHA tests by making use of the adaptive learning and pattern recognizing capabilities of neural networks. These networks can be used to recognize letters from images with adversarial clutter. This is the area I will focus on in this thesis. This thesis will list some of the difficulties regarding the extraction of relevant data from a CAPTCHA and how to possibly overcome these difficulties. However the main focus will be on understanding the inner workings of neural networks and on searching for types and configuration of neural networks best used for pattern recognition.

# Chapter 2

## Premise and research questions

### 2.1 Premise

The main objective of this thesis is to ascertain whether neural networks are capable of solving the current generation of CAPTCHA images. we will define the premise as following:

*"Are neural networks a viable tool for solving the current generation of CAPTCHA?"*

### 2.2 Research questions

The research can be divided into two separate subjects. If one were to develop software for automatic CAPTCHA solving, the following questions and problems would need to be addressed.

#### **CAPTCHA:**

- What are CAPTCHAs?
- Why were they developed?
- Are there downsides to using CAPTCHA tests?

#### **Neural networks:**

- How do neural networks operate?
- Which types of neural networks are well suited for pattern recognition?
- What network configuration would perform best?

**General:**

- Will the solution be usable for long time?
- Is there enough economic incentive to invest in development?

# Chapter 3

## Methodology

This thesis is made up out of several parts, the first parts are a study on CAPTCHA and neural networks. The last part is a description of applying the gathered information.

### 3.0.1 Research philosophy

The research philosophy used in this thesis is post-positivism. It will use quantitative, qualitative and other methods (such as interpreting and combining several sources). The premise in this philosophy is that the world can be objectified, but the objectification is always coloured based on the subjective knowledge, experiences and beliefs of each individual.

### 3.0.2 Approach

Using a practical approach to the solving the CAPTCHA tests by implementing them in an experimental setting provided some obstacles. The information in this thesis was gathered primarily by performing literature studies on the subjects of these obstacles.





# Chapter 4

## Corpus

### 4.1 CAPTCHA

#### 4.1.1 CAPTCHA, an explanation

A CAPTCHA (pronounced /'kæp.tʃə/) is a type of challenge-response test that aims to make sure the response was made by a human. These tests are designed in such a manner that they should be easy to generate and grade by a computer. At the same time should it be difficult for a computer to solve while maintaining the rule that a human should be able to solve the test without much difficulty. If a test was solved successfully one can assume that the response was entered by a human.

These test are mostly found on sites where one would like to prevent the access to unwanted bots<sup>1</sup>. This is because having lots of spam on a site or in a service can have real detrimental consequences for that site or service. This is because most contemporary interactive sites store and serve their content from a database. When a database gets filled up the site can become slow and sluggish, degrading the customer's experience. This is only one of the many useful applications of CAPTCHAs. On the other hand, legitimate users also need to solve these tests, so it requires them to perform an extra task before they can post their content, create an email or view a certain page. While this 'simple' extra task does not seem like a large barrier, it does inconvenience some people enough to prevent them from posting valid content. This problem becomes even more apparent when dealing with non-native speakers [Banday and Shah, 2011]. Protecting your site with a CAPTCHA can even have a detrimental effect on the conversion rates<sup>2</sup>.

---

<sup>1</sup>Automated scripts, in this context often used to gather personal information or used to distribute unwanted messages

<sup>2</sup><http://www.seomoz.org/blog/captchas-affect-on-conversion-rates>

### 4.1.2 The history of CAPTCHA

Moni Naor was the first one to think of the concept of CAPTCHA in 1996. He proposed that reverse Turing testing, as CAPTCHAs are often called, should consist of "tasks where humans excel in performing, but machines have a hard-time competing with the performance of a three year old child." Some of these tasks were [Naor, 1996]:

- gender recognition
- understanding facial expressions
- understanding handwriting
- filling in words

In 1997 'Yahoo!' was having a gargantuan problem with spammers using bots to create free email addresses used to spread a huge amount of unwanted advertisement, giving Yahoo email addresses a bad reputation. 'Yahoo!' contacted Carnegie Mellon University<sup>3</sup> for help, by 2000 the first real CAPTCHA as we know them was invented [Egen, 2009]. These were also the people who first used the term "CAPTCHA" and tried to trademark it.

As computing power increased, so did the amount of CAPTCHA tests being broken. By 2008 there was an 30% to 60% success rate on the most used CAPTCHA systems [Yan and El Ahmad, 2008]. As a response to this Von Ahn and his team at Carnegie Mellon University released reCAPTCHA (Figure B.2, page 74) in September 2008, a popular system which is still in use.

CAPTCHAs have always undergone changes once it became clear a certain generation method didn't stop the spammers anymore. The first CAPTCHAs generated by EZ-Gimpy for 'Yahoo!' looked completely different from the CAPTCHAs that are currently being generated. A good example of the adaptive nature of CAPTCHAs is reCAPTCHA, where you can see the changes depending on when a CAPTCHA was generated. (Figure B.3, page 74)

### 4.1.3 Types of CAPTCHA

Following is a list and description of the different types of CAPTCHA, courtesy of [Sauer and Hochheiser, 2008].

**Character based** In this category a string of characters is presented to the user.

This string can contain either words or random alphanumeric characters.

The task is to identify the string of characters.

---

<sup>3</sup><http://www.cylab.cmu.edu/research/projects/2008/captcha-project.html>

**Image based** In this category images or pictures are presented to the user. This is normally in the form of an identifiable real-world object, but can also be presented in the form of shapes. The task is to identify the object shown in the picture.

**Anomaly based** In this category a series of different objects, shapes, characters,... is presented to the user. The task is to determine which object, character or shape does not belong in a set of images displayed on the screen.

**Recognition based** In this category all previous categories can be used. The user is tasked to determine what is being presented to them and respond accordingly.

**Sound based** In this category an audio version of a CAPTCHA is presented. The task is to identify the words and letters or image presented to the user.

### 4.1.4 Data extraction

As previously stated, the data extraction part of solving CAPTCHAs is not the main focus of this thesis. Therefore I will not give in-depth explanations of the algorithms used and described here.

CAPTCHAs are by design tough to solve for a computer. The majority of times a CAPTCHA gets cluttered with noise, or the letters get crowded together. This crowding or noise makes it so that the characters on the image are not separate entities. This is to impede the segmentation of the CAPTCHA. Measures against segmentation are necessary to prevent an OCR<sup>4</sup> algorithm from simply reading and solving the test. This could be possible, as computers can (given the right algorithms) be very efficient at pattern recognition. People trying to solve the CAPTCHA test automatically, have to separate the individual characters first before they can pass the characters to an OCR algorithm for classification.

[Yan and El Ahmad, 2008] described a working segmentation algorithm in 2008, but [Huang et al., 2010] has significantly improved on the performance, so it should be able to segment the contemporary CAPTCHAs.

In the unlikely case that the CAPTCHAs you are trying to solve don't have the segmentation issues, then you can first try to reduce the noise and then segment the characters by using the flood-fill method, as described by [Cai, 2008].

---

<sup>4</sup>Optical Character Recognition

### 4.1.5 The future of CAPTCHA

The arms race between the makers of CAPTCHA systems and people trying to break them favours the defender. This is different from other computer security arms races, where the odds are in favour of the adversary. This is because CAPTCHA has broken the traditional pattern where the attacker's role is to generate new instances while the defender must recognize them, recognizing a problem is almost always harder than generating them. Websites and services using CAPTCHAs can easily change the CAPTCHA generation algorithm, creating new unsolvable CAPTCHAs, while the attackers now have the challenging recognition problem. This battle has brought advances to the field of Automated Pattern Recognition and Artificial Intelligence. Some people even believe that eventually the solving algorithms will become so sophisticated they could be classified as a sentient AI<sup>5</sup> (Figure B.1, page 73). All the positive aspects and technological innovations aside, CAPTCHAs are inherently flawed. As the solving agents got better, the CAPTCHAs had to become harder. We have reached the point where the average user is having difficulties solving the standard CAPTCHAs<sup>6</sup>.

CAPTCHAs of the future will need to explore completely new test systems. As an example of this, [Sauer and Hochheiser, 2008] and colleges did a small research about how the current CAPTCHA (even the audio CAPTCHA) has serious shortcomings when trying to accommodate for blind or visually impaired users. They suggest a new system where the sound and image part of the test are integrated, opposed to the current system where the audio part and the visual part are on independent development and maintenance paths. With their suggested test all visually impaired and hearing impaired users should be able to solve the test.

But even the newly developed systems will eventually succumb to the ever increasing computing power [Beede, 2010]. The question is whether CAPTCHAs are the right way to prevent spammers, because how many "unsolvable CAPTCHAs" (Figure B.4, page 75) is a user going to tolerate before giving up? The malcontent of some has even led to the creation of intentionally unsolvable CAPTCHAs (Figure B.5, page 76) through a service called "CRAPCHA"<sup>7</sup>, aimed at ridiculing the real CAPTCHA services.

It would seem evident from years of use and research that CAPTCHAs are far from perfect as a solution. Remove spammers from the equation and we remove the need for CAPTCHAs entirely; this is the mentality

---

<sup>5</sup><http://thenextweb.com/2009/10/15/inevitable-future-captcha/>

<sup>6</sup>[http://www.internetevolution.com/author.asp?section\\_id=587&doc\\_id=259406](http://www.internetevolution.com/author.asp?section_id=587&doc_id=259406)

<sup>7</sup><http://crapcha.com/>

we should be aiming for. The perfect CAPTCHA is no CAPTCHA at all. [Bushell, 2011]



## 4.2 Neural Networks

The following section is a synthesis of 2 books describing neural networks, so similarities in structure and wording will be found. However, no malicious copyright infringement is intended. The two books used are "Neural Networks" by David Kriesel [Kriesel, 2013] and "Introduction to neural networks with Java" by Jeff Heaton [Heaton, 2005].

### 4.2.1 The concept of time

Time is not measured by seconds inside of a neural network. Instead, time is expressed as an amount of cycles. The continuous time distribution has been replaced with a discrete one.

**Definition 1.** (The Concept of time). We reference the current time as  $(t)$  the next step in time is referred to as  $(t + 1)$  and the previous step as  $(t - 1)$ . The other steps further ahead or behind are referenced analogously.

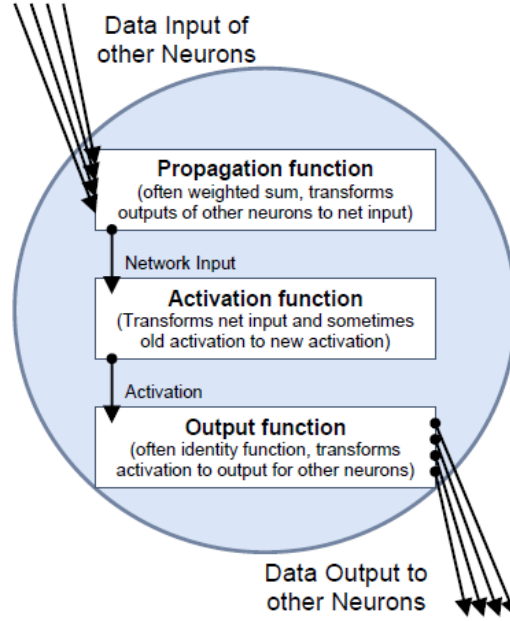
When we refer to a certain point in time for mathematical variables (e.g.  $net_j$  or  $o_i$ ) we will do this as following  $net_j(t - 1)$  or  $o_i(t + 3)$ .

### 4.2.2 The components of an artificial neural network

An artificial neural network comprises the basic processing units (the *neurons*) and the directed weighted connections between these neurons. These connections have a certain strength, called the weight of the connection. A connection weight is written as  $w_{i,j}$  for the connection between neuron  $i$  and neuron  $j$ .

**Definition 2.** (Neural network). A neural network is a sorted triple  $(N, V, w)$  which contains two sets  $N, V$  and a function  $w$ .  $N$  is a set of neurons,  $V$  is a set of connections where for each connections between neuron  $i$  and  $j$  applies  $\{(i, j) | i, j \in \mathbb{N}\}$ . The function  $(w : V \rightarrow \mathbb{R})$  defining the connection weights with  $w((i, j))$  defining the weight for the connection between neuron  $i$  and neuron  $j$ , shortened to  $w_{i,j}$ .

Depending on the implementation, a connection in the network does not exist or has no influence when the weight is undefined or equal to zero. All the weights can be combined in a square *weight matrix*  $W$ , or in a *weight vector*  $W$ . The row number indicates where a connection begins and the column number where it ends. When using this representation, a zero indicates a non-existing or inactive connection. This kind of representation is also called a *Hinton Diagram* (Figure B.6, page 77).



**Figure 4.1:** Neuron Data processing (Source: [Kriesel, 2013] page 39, Figure 3.1)

### Neuron Data Processing

The following sections will discuss the other distinguishable components and processes of the neuron and its connections.

#### Propagation function

In a network there will likely be more than one neuron linking to neuron  $j$ . This means that each one of these neurons will pass its output to neuron  $j$ . The propagation function receives all these neuron outputs  $o_{i_1}, \dots, o_{i_n}$  for the neurons  $i_1, i_2, \dots, i_n$  and combines those into a single value, the *network input*  $net_j$ . The network input is directly based on the connection weights  $w_{i,j}$ .

**Definition 3.** (Propagation function and network input) The network input, or  $net_j$ , can be calculated with the propagation function  $f_{prop}$  as follows:

$$net_j = f_{prop}(o_{i_1}, \dots, o_{i_n}, w_{i_1,j}, w_{i_n,j}) \quad (4.1)$$

Where  $I = \{i_1, i_2, \dots, i_n\}$  is the set of neurons so that  $\forall z \in \{1, \dots, n\} : \exists w_{i_z,j}$ . The *weighted sum* is a very popular, and the most used propagation function.

$$net_j = \sum_{i \in I} (o_i \cdot w_{i,j}) \quad (4.2)$$



### Neuron threshold value

**Definition 4.** (Threshold value in general). The threshold value  $\Theta_j$  for neuron  $j$  is uniquely assigned to  $j$  and defines the position of the maximum gradient of its activation function.

This means that the activation function of a neuron will react very sensitive to changes when  $net_j$  is around the threshold value.

### Neuron activation function.

**Activation status.** Neurons do not have a static activation status, the activation state depends on the  $net_j$ , the previous activation state and the activation function. A neurons activation state is important, because it will directly influence a neuron's reaction to the  $net_j$ .

**Activation function.** The activation, also called the transfer function,  $a_j$  of neuron  $j$  depends on on previous states, this means  $a_j$  will change overtime. The activation function uses the network input  $net_j$ , the threshold value and the previous activation state  $a_j(t - 1)$  to create a new *activation state*. The neuron activation function is also called the neuron identity.

**Definition 5.** (activation and activation function). In neuron  $j$  the activation function is expressed as

$$a_j(t) = f_{act}(net_j(t), a_j(t - 1), \Theta_j) \quad (4.3)$$

**Binary threshold function.** The binary threshold function, also called the *Heaviside function*, is the simplest activation function as can only take on two values, on or off. The activation will change, depending on whether the input value is above or below the threshold value. In either case the function changes from one value to another. This activation function doesn't have a variable. This means that its derivative is 0, making backpropagation learning impossible for this threshold function.

**Fermi function.** The Fermi function, also called the logistic function, maps a range of values between  $(0, 1)$ . The function used to obtain this behaviour is the following:

$$\frac{1}{1 + e^{-x}} \quad (4.4)$$

**hyperbolic tangent.** The hyperbolic tangent function maps to values within the range of  $(-1, 1)$ . Its function is described as following:

$$\tan(x) \quad (4.5)$$

**Fermi function with temperature parameter.** The Fermi function can be expanded with a temperature parameter. The smaller this parameter becomes the colder the activation function and the steeper the gradient of the activation function becomes. The reverse also holds true. By making the temperature parameter very small, we can emulate the Heaviside activation function's behaviour. The advantage of this is that the activation would still be virtually binary, but it has a derivative  $\neq 0$  and can therefore be used with backpropagation. The extended Fermi activation formula:

$$\frac{1}{1 + e^{\frac{-x}{T}}} \quad (4.6)$$

### The bias neuron

The bias neuron (also called the on neuron) is a technical trick used to make training the network easier. The purpose of a bias neuron is to represent a neurons threshold value as a connection weight. The threshold values normally are stored inside the neuron itself. This makes it necessary to access the threshold value inside the neuron before the neurons activation or new threshold value can be calculated. This unnecessarily complicated things, so this is where the bias neuron comes in.

Threshold values  $\Theta_{j_1}, \dots, \Theta_{j_n}$  for neurons  $j_1, j_2, \dots, j_n$  can be expressed as a connection weight of a constantly firing neuron. This constantly firing neuron is the bias neuron. To implement this change, an additional neuron, whose output is always 1, will be added tot the network and connected to neurons  $j_1, j_2, \dots, j_n$ . The newly made connections' weights will be set to the negative threshold values.  
 $w_{BIAS, j_n} = -\Theta_{j_n}$

Now that the threshold values are implemented as connection weights, the threshold value  $\Theta_{j_n}$  of neurons  $j_1, j_2, \dots, j_n$  is set to 0. This also means that the threshold value can be directly trained with the same operation as the connection weights. This makes the learning process considerably easier. The threshold value is no longer included in the activation function, but its effect is now included in the propagation function.

The advantages of the bias neuron are obvious when implementing and training a neural network. However they do have a disadvantage when you want to

visually represent the neural network. These extra connections quickly clutter up the visual representation, and this becomes only worse when a lot of neurons are being used. But since the implementation of a bias neuron is almost a certainty, a lot of people omit the bias neuron from their illustrations for improved clarity. We know it exist and know that threshold values can simply be treated as weights because of it.

**Definition 6.** (Bias neuron). A bias neuron is a neuron whose output value is always 1 and is used to represent neuron threshold values as connection weights, which enables any weight training algorithm to train the biases at the same time.

Let  $j_1, j_2, \dots, j_n$  be neurons with threshold values  $\Theta_{j_1}, \dots, \Theta_{j_n}$ . By inserting a bias neuron whose output value is always 1, generating connections between the bias neuron and neurons  $j_1, j_2, \dots, j_n$  and weighting these connections  $w_{BIAS,j_1}, \dots, w_{BIAS,j_n}$  with  $\Theta_{j_1}, \dots, \Theta_{j_n}$ , we can set  $\Theta_{j_1} = \dots = \Theta_{j_n} = 0$  and receive an equivalent neural network whose threshold values are realized by connection weights.

### Neuron activation order.

The sequence in which the neurons of a neural network receive and process the input critically influences the result.

**Synchronous activation.** In this activation scheme all neurons update at the same time. They simultaneously calculate the input, activation and output. This type of activation scheme closest resembles actual biological neural networks. However implementing this is only possible on hardware capable of parallel processing. This is the most generic activation scheme and can be used for any network, independent of network topology. Keep in mind that even though this scheme can be used for any network, some network types like feedforward networks would not benefit from this scheme.

**Asynchronous activation.** The counterpart of synchronous activation. The neurons won't fire all at once in this scheme, but in different points in time. Asynchronous activation can be split up in several subcategories.

**Random order activation.** In this activation scheme, a random neuron  $i$  is chosen from all the neurons. This neuron will then calculate its input, activation and output. the activation cycle will end when  $n$  neurons have been updated in a neural network consisting of  $n$  neurons. During these random cycles some neurons can be update multiple times, while others don't get updated at all. This order of

activation is not always useful because of this, as the network might accidentally favour some neurons.

**Random permutation activation.** In this scheme a list of each neuron in a randomized order is made for each cycle. Therefore during one cycle, each neuron will be activated exactly once. But in a random order.

This activation order doesn't get used much, just like random order activation. This is because of the huge overhead generated by having to build a random order list of each neuron for each activation cycle. While Hopfield network topology technically should be updated at random like this, in practice a fixed order activation is preferred.

**Topological order activation.** In this scheme the neurons are update in a fixed order. The order is defined by the network topology. First the input neurons get activated then the layer after that, and so on until the output neurons are reached. This activation order cannot be used for recurrent networks due to the definition of recurrent networks, as they don't have a clearly defined start or end. This is the most efficient activation schema when using feedforward networks.

**Fixed order activation.** In this scheme the order of activation gets set during implementation. This can be useful in some cases (again feedforward networks come to mind). This activation order will have detrimental effects when a network can change topology during runtime, this because the activation order cannot and will not change based on the new topology.

### Neural output function

The output function calculates what value neuron  $i$  will pass to its connected neurons  $j$ . The output value  $o_j$  of neuron  $j$  is calculated from its activation state  $a_j$ .

**Definition 7.** (Output function). For neuron  $j$  the output function is formally described as

$$f_{out}(a_j) = o_j \quad (4.7)$$

The output function generally is the same as the activation  $a_j$ , which is directly returned as output.

$$f_{out}(a_j) = a_j, SO o_j = a_j \quad (4.8)$$

### Input and output of data

The purpose of a neural network is to receive, process and return data. The input components for a network with  $n$  input neurons can be scribed by vector  $x = (x_1, x_2, \dots, x_n)$  and the output components for a network with  $m$  output neurons can be scribed by vector  $y = (y_1, y_2, \dots, y_m)$ . this leads us to following definitions:

**Definition 8.** (Input vector). A network with  $n$  input neurons needs  $n$  input values  $x_1, x_2, \dots, x_n$  they can be combined in an input vector  $x = (x_1, x_2, \dots, x_n)$ .

**Definition 9.** (Output vector). A network with  $m$  output neurons has  $m$  output values  $y_1, y_2, \dots, y_m$  they can be combined in an input vector  $y = (y_1, y_2, \dots, y_m)$ .

### 4.2.3 Neural network topologies

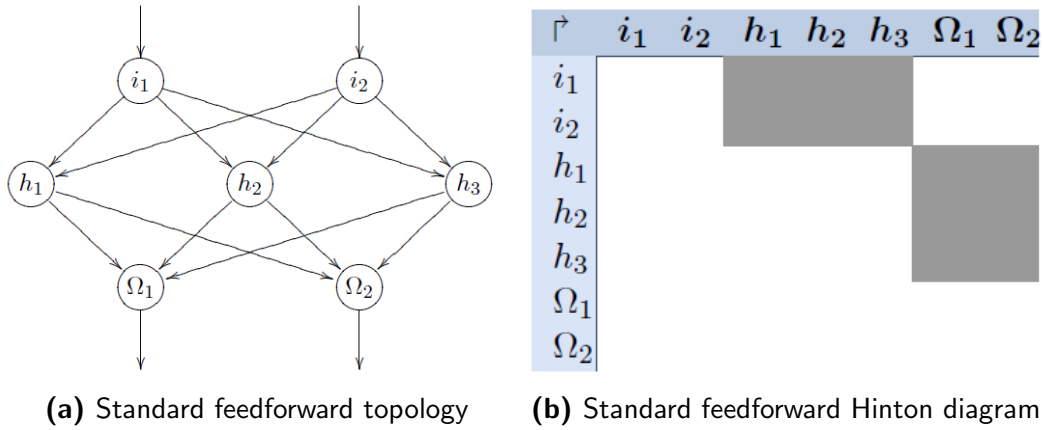
Until now, we have only looked at the different elements that make up a neuron. It is time to zoom out and start looking at how several neurons can be organized together to effectively make a neural network. The following is a list of the most frequently found network designs. Every topology described will be accompanied by a visual representation and a Hinton diagram, both courtesy of [Kriesel, 2013]. The Hinton diagram is supplied so that the characteristics of the network can immediately be seen.

To make the transformation from the visual representation to the Hinton diagram, the following rules were used:

- dotted weight connections were represented as light gray fields
- solid weight connections were represented as dark grey fields
- the directional arrows between individual neurons cannot be added. The  $\rightharpoonup$  symbol has been introduced in the upper left corner to express how the connections start from the line neurons and end at the column neurons.

### Feedforward networks

**Standard feedforward network.** The first topology we will discuss is the one of the standard feedforward network. In a feedforward network the neurons are divided into layers. These layers can be grouped in three types of layers: one *input layer*,  $n$  *hidden layers* and one *output layer*. The hidden layers, also often referred to as the processing layers, cannot be seen from the outside (hence the name hidden). The neurons in those layers are referred to as hidden neurons. All



**Figure 4.2:** Standard feedforward network; Source:[Kriesel, 2013]

connections in a standard feedforward network go towards the next layer. The next layer is defined as the layer one step closer towards the output layer. In many cases we will find that each neuron  $i$  is connected to every neuron  $j$  of the next layer. These networks are called completely linked feedforward networks. As a naming convention, the output neurons will often be referred to as  $\Omega$ . (Figure 4.2)

**Definition 10.** (Feedforward network).

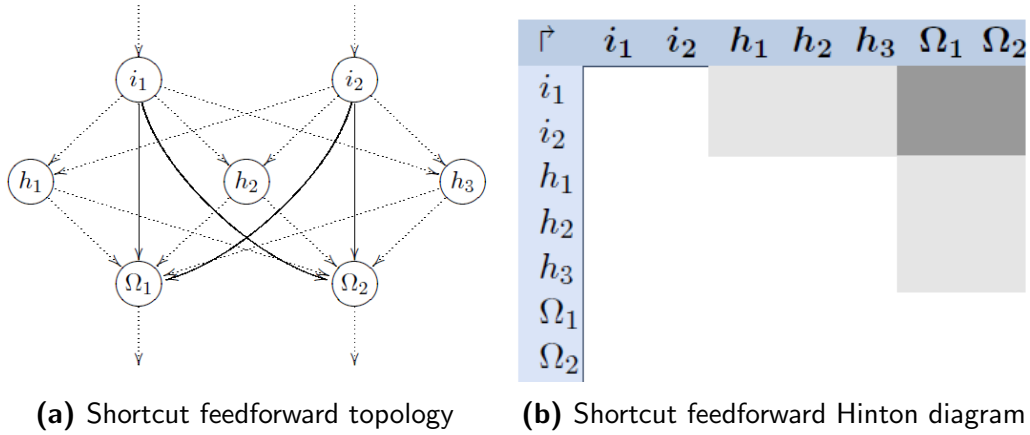
A feedforward network consists of three types of layers, one input layer, one output layer and optionally one or more processing layers. These layers are clearly separated and the connections only go from one neuron layer to the next, directed towards the output layer.

**Shortcut connections.** This is essentially a standard feedforward network (see 4.2.3) where connections exist that skip one or more levels. These so-called shortcut connections may only be directed towards the output layer. (Figure 4.3)

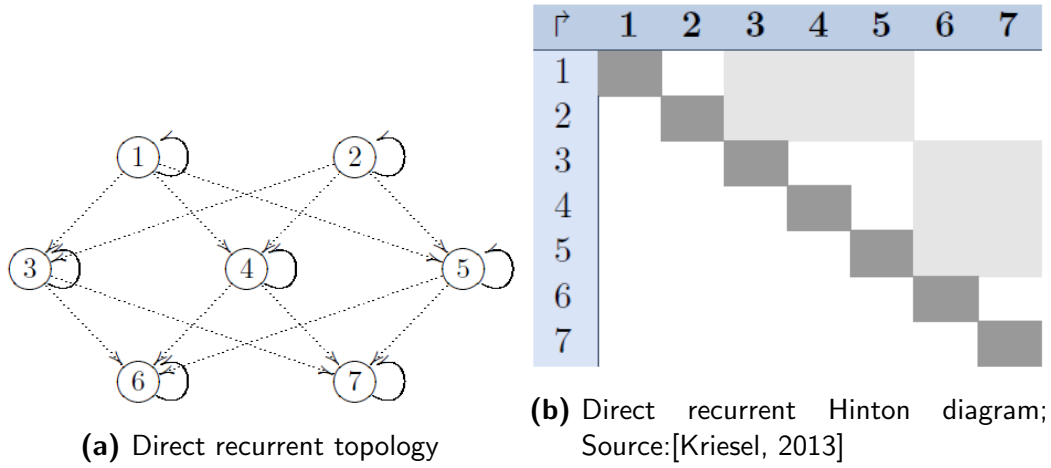
**Definition 11.** (Shortcut feedforward). A shortcut feedforward network consists of three types of layers, one input layer, one output layer and optionally one or more processing layers. These layers are clearly separated and the existing connections always are directed towards the output layer, the connections may span over one or more layers.

### Recurrent networks

Recurrence is the event where a neuron influences itself. This can be done through any means or by any connection. It is because of this that recurrent networks often don't have explicitly defined input and output neurons.



**Figure 4.3:** Shortcut feedforward network; Source:[Kriesel, 2013]



**Figure 4.4:** Direct recurrent network

**Direct recurrent.** Direct recurrence happens in a network when neurons feed their output back to themselves as input. This is also called self-recurrence. The neurons inhibit themselves, and as a result of this strengthen themselves in order to reach their activation limits. (Figure 4.4)

**Definition 12.** (Direct recurrence). A direct recurrence network is a network where a neuron  $j$  is connected to itself. The connection weight is then defined as  $w_{j,j}$ . The diagonal of weight matrix  $W$  will no longer be zero as a direct result from this.

**Indirect recurrent.** Indirect recurrence happens when in a network connections towards the input layer are allowed. A neuron can influence itself by passing its

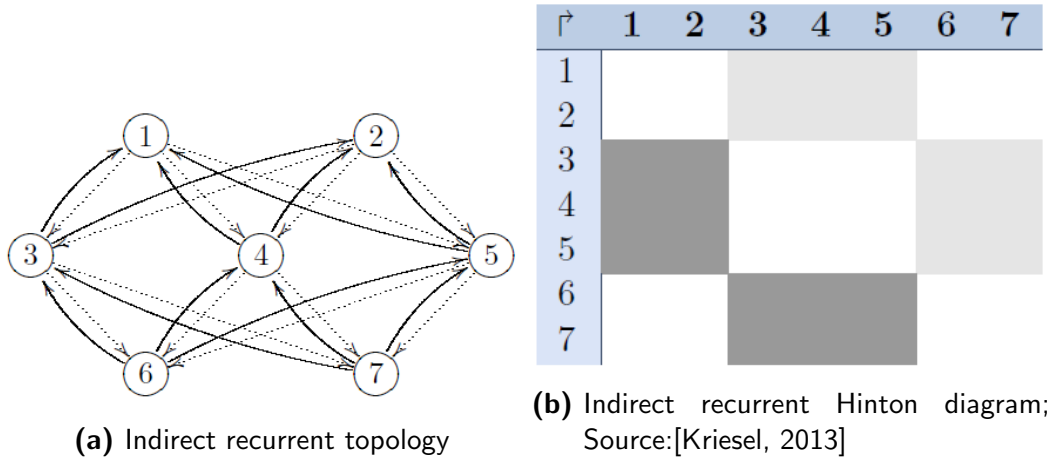


Figure 4.5: Indirect recurrent network

output to one of the previous or one of the next layers. (Figure 4.5)

**Definition 13.** (Indirect recurrence). An indirect recurrent network is a network where connections forward and backward in the network may exist. The neurons influence themselves by influencing the predecessor layers.

**Lateral recurrent.** Lateral recurrence happens when connections between neurons of the same layer exist. These connections often make it so that each neuron inhibits the other neurons and strengthens itself. This often means that only the strongest neuron becomes active (*winner-takes-all scheme*). (Figure 4.6)

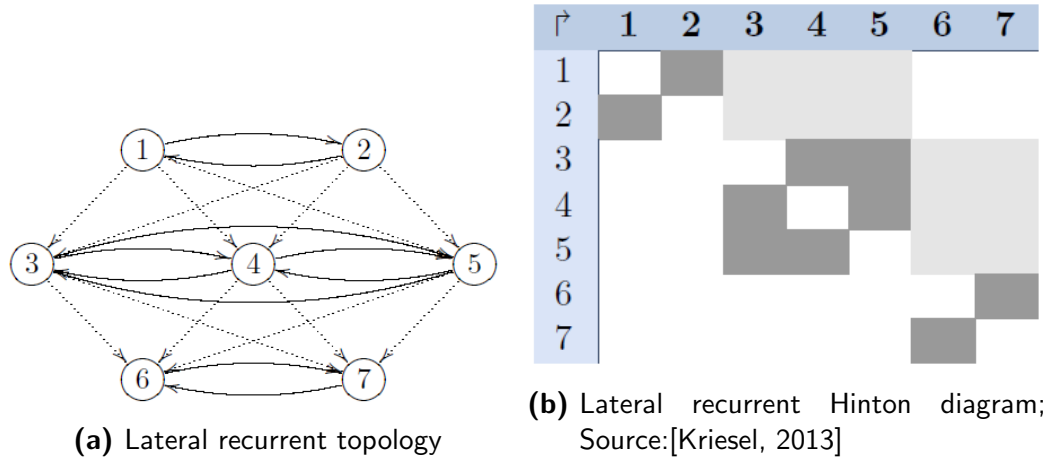
**Definition 14.** (Lateral recurrence). A lateral recurrent network is a network where connections between the neurons of the same layer are allowed.

### Completely linked networks

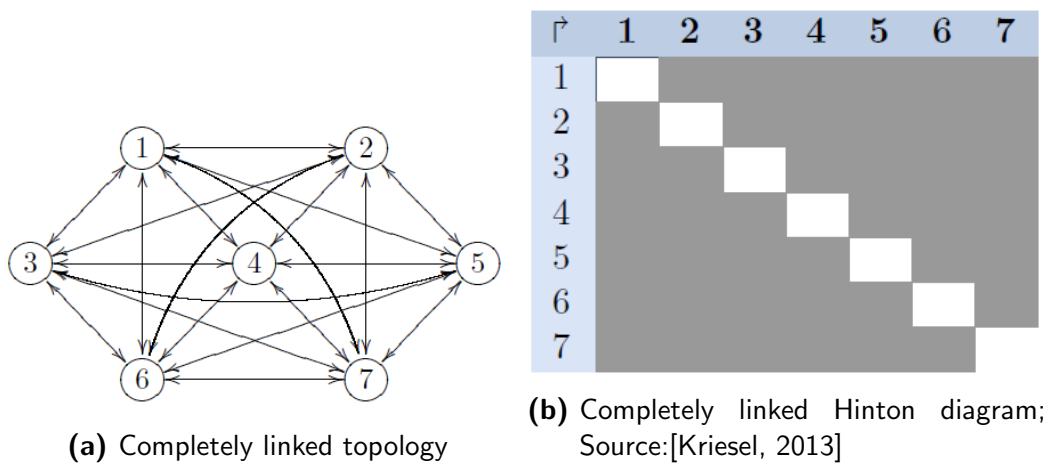
A completely linked network happens when all neuron connections are allowed, except for direct recurrent connections. All neuron connections must be symmetric, what means that  $w_{i,j} = w_{j,i}$ . As a direct result of the complete linking, input and output neurons can no longer be strictly defined and clearly defined layers do not longer exist. Another result is that the weight matrix  $W$  can be different from zero everywhere, except along the diagonal. (Figure 4.7)

**Definition 15.** (Complete interconnection). A complete interconnected network is a network where each neuron is always connected to every other neuron. As a result every neuron can become an input neuron.





**Figure 4.6:** Lateral recurrent network



**Figure 4.7:** Completely linked network

#### 4.2.4 Neural network learning

One of the most interesting features of a neural network is the ability to familiarize themselves with a problem by means of training. This means that after sufficient training, the network will be able to solve unknown problems of the same type. This is called generalization. Training a network involves some form of learning procedure. A learning procedure is a rather complex thing, so a brief explanation will be given first.

**Learning paradigms.** We must be careful with the term "learning" as it is a very comprehensive term. A system learns when it changes itself in order to adapt to changes in the systems environment. A neural network as a learning systems changes, learns, whenever its components change. So any change in the components previously mentioned due to environmental changes will result in a changing network. This process is classified as neural network learning. Looking at the previous components capable of changing we can form a list of all theoretically possible means of neural network learning.

1. developing new connections
2. deleting existing connections
3. changing connecting weights
4. changing the threshold values of neurons
5. varying one or more of the three neuron functions (activation function, propagation function and output function)
6. creating new neurons
7. deleting existing neurons (and so its existing connections)

We can safely assume that changing the connection weights will be the most commonly used method of training. Especially because of the simple fact that deleting a connection (no longer training a connection with weight) and creating new connections (changing the connection weight from 0) in the connection matrix can be done by connection weight adjustment training. With the introduction of the bias neuron ( 4.2.2) the threshold values can also be changed by using connection weight adjustments. We can perform any of the first four of the learning paradigms by just training synaptic weights.

Changing neuron functions is difficult to implement and not very intuitive. Because of this, those paradigms aren't really popular and will not be discussed any

further. The option of generating new, or removing old neurons has a few advantages. This method not only provides a way to keep the connection weights well adjusted during the training of a neural network, it also optimizes the neural network topology. This type of training is gathering a growing amount of interest because of these reasons. A method of achieving this functionality would be using evolutionary algorithms. However we will not elaborate any further on this since this is outside the scope of this thesis.

To summarize, neural networks learn by modifying the connecting weights according to rules formulated as algorithms. Therefore a learning procedure is always an algorithm that can easily be implemented by means of a programming language.

### Training patterns, teaching input, desired output and error vector

**Definition 16.** (Desired output). The expected result of the output layer when presented with a training pattern.

**Definition 17.** (Teaching input). For an output neuron  $j$  the teaching input  $t_j$  is the desired and correct value  $j$  should have after the input of a training pattern. The teaching inputs  $t_1, t_2, \dots, t_n$  can be combined into a vector  $t$ .  $t$  always refers to a specific training pattern  $p$ . this correlation is contained in the training set.

**Definition 18.** (Training pattern). A training pattern is an input vector  $p$  with the components  $p_1, p_2, \dots, p_n$  whose desired output is known. By entering a training pattern into the network we receive an output that can be compared with the desired output.

**Definition 19.** (Training set). A training set (named  $P$ ) is a finite set of pairs of training patterns ( $p$ ) and teaching inputs ( $t$ ). These finite ordered pairs  $(p, t)$  of training patterns and teaching inputs are used to train our neural network.

**Definition 20.** (Error vector). The error vector (also called the difference vector) for output neurons  $\Omega_1, \Omega_2, \dots, \Omega_n$  is the difference between the actual output vector and the teaching input for a training input  $p$ . Depending on whether you are learning offline or online, the difference vector refers to a specific training pattern, or to the error of a set of training patterns which is normalized in a certain way.

$$E_p = \begin{pmatrix} t_1 - y_1 \\ t_2 - y_2 \\ \vdots \\ t_n - y_n \end{pmatrix}$$

**Unsupervised learning.** Unsupervised learning is the type of learning closest related to how biological neural networks operate. This type of learning is not suitable for all problems as only the training patterns are given and the network itself is taxed with trying to identify and classify similar patterns.

**Definition 21.** (Unsupervised learning). The training set only consists of input patterns, the network tries by itself to detect similarities and to generate pattern classes.

**Reinforcement learning.** Reinforcement learning uses only the training patterns, but after each cycle a value is returned to the network. This value indicates whether the output was correct or not.

**Definition 22.** (Reinforcement learning). The training set consists of input patterns, after completion of a sequence a value is returned to the network indicating whether the result was right or wrong and, possibly, how right or wrong it was.

**Supervised learning.** Supervised learning uses a training set  $P$  consisting of both the training pattern ( $p$ ) and teaching input ( $t$ ) in form of the precise activation or output of all output neurons  $\Omega$ . The network is fed with the training patterns, and the connection weights then are adjusted based on the error vector obtained from subtracting the actual output with the teaching input. The goal is to change the weights in such a manner that the network cannot only associate input and output patterns independently after the training, but that it can also provide plausible results to similar but untrained input patterns. The network should be able to generalise after being successfully trained.

**Definition 23.** (Supervised learning). The training set consists of training patterns and teaching inputs so that the network can calculate a precise error vector to adjust the network with.

**Offline learning or online learning.** Offline learning is a method that stands in stark contrast with online learning. In offline learning you present all training patterns at once and the total error is accumulated or calculated with an error function. The weights will only be changed after all patterns have been presented. Offline training procedures are also called batch training procedures, since a batch of results is corrected in one go. Such an iteration of a batch training process is called epoch.

In online learning the weights are changed after every pattern presented.

Each method has advantages and disadvantages, and it isn't always clear what type is best to be used.

**Definition 24.** (Offline learning). Several training patterns are presented to the network at once, the errors are accumulated and it learns for all patterns at the same time.

**Definition 25.** (Online learning). The network learns directly from the errors of each training sample.

**Training samples** The network should be tested after training to see whether it has merely memorized the training data (the network produce the right output for our trained inputs but provides wrong answers for all other problems of the same class), or whether it was successfully trained to generalize the problem.

A network only has memorizing the training data can be sign of the fact that the network is oversized and has too much free storage capacity. On the other it can happen that the network has insufficient storage capacity, this in turn leads to bad generalization.

A training set is often divided into a set used for training the network, and a set used for verification to check for possible memorization instead of generalization. This splitting up of course only happens when there are enough training samples not to affect the initial quality of training. The usual used ratio is 70% for training data and 30% for verification. These sets should be randomly chosen. We can conclude the training of the network when both the results of the training data and the verification data are good. The verification data is included in the training, even though it is not directly used in training the network. This means that you do not modify the network to accommodate for poor results with the verification data. You accumulated when you modify the network. You should keep in mind that you run the risk of worsening the learning performance by withholding information from the network for means of verification.

**Learning curve and error measurement** A learning curve indicates the variations in the error. The motivation for creating a learning curve is to see whether the network is progressing or not. The error must first be normalized before we can use it to plot out the learning curve. A normalized error represents the distance between the correct and current output of the network. Four ways of calculating the normalized error exist. In the following sections  $\Omega$  will be the output neurons and  $O$  the set of output neurons.

**Specific error.** The specific error  $Err_p$  is based on single training sample  $p$ . This means it is generated during online learning.

$$Err_p = \frac{1}{2} \sum_{\Omega \in O} (t_{\Omega} - y_{\Omega})^2 \quad (4.9)$$

**Root mean square.** The root mean square (also referred to as RMS) of two vectors  $t$  and  $y$  is an error normalization method that can be used both for online and offline.

The online formula:

$$Err_p = \sqrt{\frac{\sum_{\Omega \in O} (t_{\Omega} - y_{\Omega})^2}{|O|}} \quad (4.10)$$

The offline formula:

$$Err = \sum_{p \in P} \sqrt{\frac{\sum_{\Omega \in O} (t_{\Omega} - y_{\Omega})^2}{|O|}} \quad (4.11)$$

**Euclidean distance.** Just like the root mean square the Euclidean distance calculates the error based on the two vectors  $t$  and  $y$ , and can be used both online and offline.

The online formula:

$$Err_p = \sqrt{\sum_{\Omega \in O} (t_{\Omega} - y_{\Omega})^2} \quad (4.12)$$

The offline formula:

$$Err = \sum_{p \in P} \sqrt{\sum_{\Omega \in O} (t_{\Omega} - y_{\Omega})^2} \quad (4.13)$$

These four methods only represent a small grasp out of the most used error measuring methods in existence. If you are interested in finding out more about error measurement please read the technical report of Prechelt [Prechelt, 1994]

Our learning curve will look different depending on the method used. Normally a perfect learning curve will look like a negative exponential function. This means that a learning curve will be proportional to  $e^{-t}$ .

**When to stop learning?** Generally learning is stopped when the user "thinks" the error became small enough. There is no clear-cut answer for this question. Analysing and comparing learning curves can bring more clarity.

A learning curve descending fast can be overtaken by another curve. This can indicate that either the learning rate of the worse curve was too high or the worse curve itself simply got stuck in a local minimum, but was the first to find it. It is also important to plot the verification data on a second learning curve as this will often provide worse results, but with good generalization this curve can decrease too.

Preventing memorization of the sample data can be achieved by early stopping. Early stopping might be beneficial when the learning curve of the verification samples suddenly rise rapidly while the learning curve of the training data is continuously falling

**Hebbian rule** The Hebbian learning rule acts as basis for most other complex learning rules. It was first described by *Donnald O. Hebb* [Hebb, 1949] and currently exists in two forms, the original rule and the generalized form.

#### The original rule.

**Definition 26.** (Hebbian rule). If neuron  $j$  receives an input from neuron  $i$  and if both neurons are strongly active at the same time, then increase the weight  $w_{i,j}$ , strengthening the connection.

$$\Delta w_{i,j} \sim \eta o_i a_j \quad (4.14)$$

With  $\Delta w_{i,j}$  being the change in weight from  $i$  to  $j$ . This change in weight is simply added to the weight  $w_{i,j}$ .  $\Delta w_{i,j}$  is dependent on following factors.

- the output  $o_i$  of predecessor neuron  $i$
- the activation function of successor neuron  $j$
- the constant  $\eta$ , the learning rate

Notice that the definitions speaks about the activation twice, while in the formula the activation is only used once. Instead of the activation  $a_i$  of neuron  $i$  and the activation  $a_j$  of neuron  $j$ , the formula uses the activation  $a_j$  of neuron  $j$  and the output  $o_j$  of neuron  $i$ . This is possible because the identity or activation of a neuron is often used as the output ( 4.2.2). Therefore  $a_i$  and  $o_i$  can be considerate as being the same value.

**Generalized form.**

**Definition 27.** (Hebbian rule, generalized). The generalized form of the Hebbian Rule only specifies the proportion of change in the weight based on the product of two undefined functions, but with defined input values.

$$\Delta w_{i,j} = \eta \cdot (h(o_i, w_{i,j}) \cdot g(a_j, t_j)) \quad (4.15)$$

With  $\Delta w_{i,j}$  being the change in weight from  $i$  to  $j$ . The change in weight is simply added to the weight  $w_{i,j}$ .  $\Delta w_{i,j}$  is dependent on following factors.

- $g(a_j, t_j)$ : this function uses the actual activation  $a_j$  and the expected activation found in the teaching input  $t_j$
- $h(o_i, w_{i,j})$ : this function uses the output of the predecessor neuron  $o_i$  and the connection weight between the two neurons  $w_{i,j}$
- the constant  $\eta$ , the learning rate

Since this is a generalized form, the functions  $g$  and  $h$  are not specified.

**The learning rate.** The speed and accuracy of a learning procedure are be controlled by and are always proportional to the learning rate ( $\eta$ ). When this value is chosen too big, it is possible to miss a favourable narrow local minima in the error function. Setting  $\eta$  very small is not a good solution either as this would dramatically increase the time and computing cost needed for training a network. Experience has shown that learning rates between 0.01 and 0.9 are mostly optimal. A popular method for finding an optimal  $\eta$  is starting out high and slowly decreasing it.

**Definition 28.** (Learning rate). The learning rate ( $\eta$ ) dictates the speed and accuracy of a learning procedure.

## 4.3 Neural networks for pattern recognition

We will take a look at several neural network implementations now that we have a clear understanding of how neural networks work and can be trained. This thesis will focus on the network implementations most frequently found in the literature to be used for pattern recognition.



### 4.3.1 Perceptron networks

This is the classic example of neural network, often when the term "neural network" is used a perceptron network (or variation thereof) is meant. Perceptron networks were first described by Frank Rosenblatt in 1958 [Rosenblatt, 1958].

While there is no set definition of a perceptron network, they are often described as being a feedforward network with shortcut connections. The input layer of the the perceptron network is called the retina, and is only used for data input. No processing happens in the input layer as the weights for the input layer are static.

#### Single layer perceptron network

A single layer perceptron (or SLP for short) is a perceptron network that has only one layer (except the input neurons). This means that the trainable connection weights going from the input layer go straight towards one or more output neurons  $\Omega$ . A single layer perceptrons with multiple outputs can simply be considered as several separate single layer perceptrons with the same input. The single layer perceptrons is because of the limitation to a single layer only able to represent linear data.

**Single layer perceptrons and the delta rule.** The delta rule is an extension on the Hebbian rule. It has the advantage that it can be used with networks without binary activation functions, and that the network will automatically learn faster when it is farther away from the target. I refer you to the book "Neural networks" page 90-95 [Kriesel, 2013] in case you want to read the whole mathematical and logical construction of the delta rule.

**Definition 29.** (Delta rule). The delta rule (also called the Widrow-Hoff rule) is based on the generalized Hebbian rule ( 4.2.4), where the function  $h$  only uses the output  $o_i$  of the predecessor neuron  $i$ , and the function  $g$  is the difference between the desired activation  $t_\Omega$  and the actual activation  $a_\Omega$  of the output neuron  $\Omega$ . If we substitute this into the Hebbian rule we get he following:

$$\Delta w_{i,\Omega} = \eta \cdot o_i \cdot (t_\Omega - a_\Omega) = \eta o_i \delta_\Omega \quad (4.16)$$

If we were to use the desired output instead of the desired activation the formula would look like this:

$$\Delta w_{i,\Omega} = \eta \cdot o_i \cdot (t_\Omega - o_\Omega) = \eta o_i \delta_\Omega \quad (4.17)$$

These formulas can be used for online learning, in the case of batch processing you simply let the error accumulate

$$\Delta w_{i,\Omega} = \eta \cdot \sum_p o_{p,i} \cdot (t_{p,\Omega} - o_{p,\Omega}) = \eta \cdot \sum_p o_{p,i} \cdot \delta_{p,\Omega} \quad (4.18)$$

### Multilayer perceptron network

The multilayer perceptron (MLP for short) can, contrary to single layer perceptron, represent functions with a higher than linear dimension. This is achieved by taking a single layered perceptron with multiple output neurons and attaching other single layer perceptrons to these outputs. These new single layered perceptrons will further process the already processed output from the output neurons. Each added layer supposedly adds the capability to process an extra dimension.

When describing a multilayer perceptron comprising 7 input neurons, a hidden layer of 10 neurons, another hidden layer of 5 neurons and an output layer of 6 neurons the following abbreviation is often used: 7-10-5-6.

The numbers indicate the amount of neurons for that layer. The first layer is the input layer (retina) and the last is the output layer. The layers in between are the hidden layers, these are always order going from closest to the input layer (and farthest from the output) to farthest from the input layer (and closest to the output).

Word of warning for reading through the literature. Not all sources use the same way of counting the network layers, some include the retina, some count the trainable weight layers and others exclude the output neuron layer.

**Multilayer perceptrons backpropagation.** The backpropagation of error learning rule is used to train multi-staged neural perceptrons. Backpropagation is a gradient descent procedure with error function  $Err(W)$  receiving all the connection weights as arguments and links them to the output error. This system is an extension of the delta rule. The output function is kept the same for each neuron, but the variable  $\delta_i$  for each neuron  $i$  needs to be generalized from one trainable weight to several ones. Again, if you want to read the whole mathematical and logical construction, please read [Kriesel, 2013] pages 104-108.

**Definition 30.** (Backpropagation). Backpropagation is based on the generalized Hebbian rule ( 4.2.4), where the function  $h$  only uses the output  $o_i$  of the predecessor neuron  $i$ , and where the function  $g$  is the difference between the desired activation  $t_\Omega$  and the actual activation  $a_\Omega$  of the output neuron  $\Omega$ . If we substitute

this into the Hebbian rule we get the following:

$$\Delta w_{i,j} : \eta \cdot o_i \cdot \delta_j \text{ with} \quad (4.19)$$

$$\delta_j = \begin{cases} f'_{act}(net_j) \cdot (t_j - y_j) & \text{(when } j \text{ is an output neuron)} \\ f'_{act}(net_j) \cdot \sum_{l \in L} (\delta_l w_{h,l}) & \text{(when } j \text{ is a hidden neuron)} \end{cases}$$

**Backpropagation and the learning rate.** The learning rate greatly influences the effect of the backpropagation during network training. The training would benefit from a variable learning rate. Large learning rate might work fine at the start to learn more quickly, but we would like to be able to increase the learning accuracy near the end of the learning procedure by decreasing the size of the learning rate. It is a bad idea to reduce the learning rate continuously, as the algorithm will get stuck when the learning rate is reduced faster than the error is reduced. The solution is to reduce the error gradually over time.

A different learning rate per layer would also be a good idea, as the layers farthest away from the output layer will be least affected by the backpropagation. The layers closest to the input layer should therefore be assigned a larger learning rate than the layers closer to the output layer.

**Eliminating the learning rate.** The two previously described problems for the learning rate when using backpropagation has led to the implementation of resilient backpropagation. Resilient backpropagation is often used, but that does not mean that resilient backpropagation is always better. We will not go deeper into comparing normal backpropagation with resilient backpropagation. Just keep in mind that resilient backpropagation is based on these two principles:

- Weight changes are not proportional to the gradient of the error function.
- Many dynamically adjusted learning rates instead of one static learning rate.

### 4.3.2 Hopfield networks

A Hopfield network is a completely linked network developed by John Hopfield [Hopfield, 1982]. He was inspired by how the magnetic particles organize themselves into a state of least energy. You can compare the magnetic forces exerted by the particles with the connection weights between neurons and position or rotation of the particles with the activation of a neuron. To simulate this, all neurons must affect each other, but not itself, hence the choice for a completely linked network.

A Hopfield network is made up out of a set of  $H$  completely linked neurons.

each of these neurons can only have two possible states, namely  $\{-1, 1\}$ . The state of the network of  $|H|$  neurons can then be expressed as a string value  $x \in \{-1, 1\}^{|H|}$ . The whole network state is set and returned as it is virtually impossible to determine a set of input and output neurons for a completely linked network.

**Training a Hopfield network.** A set  $P$  of training patterns  $p \in \{1, 1\}^{|H|}$  will be used in training. Each pattern represents a minimum for out network. In contrast to other networks, we already know the minima for the error function, so we don't need to look for them anymore. Instead we can just define them, so that the network will converge to the closest minimum when input is presented.

Hopfield network training happens offline and is done in a single iteration. this means that all neurons will be trained at once.

$$w_{i,j} = \sum_{p \in P} p_i \cdot p_j \quad (4.20)$$

After the pattern has been trained the weights  $w_{i,j}$  are processed. For each weight we check if the neurons are in the same state or not. If they are in the same state, add 1 to the weight. Are they in a different state, subtract 1 from the weight. This algorithm then is repeated for each training pattern  $p \in P$ . The more neurons  $i$  and  $j$  are in the same state the higher the total connection weight  $w_{i,j}$  will be and the more the connected neurons will be forced to attain the same state. The same is true for the opposite.

**Maximum trainable patterns.** Thanks to this training we can store a number of patterns  $p$ , however this amount is limited. The maximum number of reconstructible patterns  $p$  is limited to  $|P|_{MAX} \approx 0.139 \cdot |H|$ . This was shown by complex mathematical analysis not covered in this thesis.

**The influence of connection weights.** A neuron can change their state in function of the current states of the other neurons and in function of the connection weights. This means that the connection weights are able to influence the change of the whole network. There are three possibilities for the connection weights

- $w_{i,j} > 0$ : This will force the two connected neurons to become equal in state. The bigger this value is, the harder the two neurons will try to get in the same state.
- $w_{i,j} < 0$ : This will force the two connected neurons to become opposite in state. The bigger this value is, the harder the two neurons will try to get in the opposite state.

- $w_{i,j} = 0$ : This means that the two neurons do not influence each other.

**Converging a Hopfield network.** The network will start to change once it has been trained and has received an input pattern. The change in the individual neurons  $k$  can be expressed as

$$x_h(t) = f_{act} \left( \sum_{j \in H} w_{j,h} \cdot x_j(t-1) \right) \quad (4.21)$$

Where  $f_{act}$  generally is the binary activation function with threshold value 0. This means that neuron  $h$  adds the weights from the previous step of all neurons connected to it. As said before, the sign of this value will determine in which direction the neuron will change, and the size how strongly it is forced to change.

**Input and output of a Hopfield network.** A Hopfield network automatically looks for a minimum for the state it is in. We will use this characteristic, and define an input pattern as being a state of a Hopfield network. When we use the binary string representation again, then a Hopfield input pattern can be described as string  $x \in \{1, 1\}^{|H|}$ .

The network will converge to a minimum state after the network has been initialized with the input pattern. But how can we now that such a minimum has been reached? Simple, the network stops. Regular completely linked (hence only zeros on the diagonal) Hopfield network will always converge to a minimum when presented by an input pattern. We can then build another binary string from the reached equilibrium,  $y \in \{1, 1\}^{|H|}$  is then the Hopfield network output.

### 4.3.3 Self-organizing maps

Self Organizing maps (also called Kohonen networks, or shorted to SOM) were developed by Tuevo Kohonen in the Eighties [Kohonen, 1982]. Just like Hopfield networks, the state of the network will be the output, but Kohonen networks learn completely unsupervised. It also is not needed to know what the neurons calculate, we only need to know which neuron was active at a certain time. In short, Kohonen networks are trained without supervision, we not interested in the exact output but rather in which neuron provides the output.

**Definition 31.** (Self-organizing map). A self-organizing map (or Kohonen network) is a set  $K$  of self organizing map neurons. When an input is presented exactly neuron  $k \in K$  that is closest to the input pattern will be activated.

**Structure of a self organizing map.** Kohonen networks are used to map a high-dimensional input onto a low-dimensional grid of neurons, so a map of the high-dimensional space can be drawn. For this purpose the network obtains many points of the high-dimensional input space where every point in that space will be assigned to a neuron. There essentially are two spaces the Kohonen networks are working in, namely the high-dimensional input space and the low-dimensional grid of neurons. Even when the the dimensions of the two spaces would be the same, the spaces would not be equal. It would only mean that in this case they only have the same dimension.

In a one-dimensional grid, the neurons can have a maximum of 2 connections. So it could be organized as a string of pearls where every neuron has exactly 2 neighbours (except the start and end neurons of course). In a two-dimensional each connection can have a maximum of 4 connections. This means they could be organized as a square array or in a of honey-comb structure. Higher-dimension grids can be used, however the amount of maximum connections increases exponentially, and these high-complexity networks often lack a visual representation. Therefore they are not used very often. Irregular topologies are possible too, but theses are also not used very often.

**Training a self organized map.** Training a Kohonen network makes it cover the input space. This is actually rather straightforward and can be described in five steps.

1. Initialization. The network starts with random neuron centers  $c_k \in \mathbb{R}^N$  of the neuron space.
2. Create an input pattern. A point  $p$  is selected from the input space  $\mathbb{R}^N$ . This is the stimulus that will be entered into the network.
3. Measure distance. The distance  $\|p - c_j\|$  is then calculated for every neuron  $j$ .
4. Winner takes it all. The neuron which has the smallest distance to  $p$  wins. This means that a neuron will be randomly selected from all the neurons fulfilling this condition:

$$\|p - c_i\| \leq \|p - c_j\| \forall k \neq i$$

5. Adapt the centers. Move the neuron centers within the input space based on this rule:

$$\Delta c_j = \eta(t) \cdot h(i, j, t) \cdot (p - c_j)$$

The value of  $\Delta c_k$  is added to the existing neuron centers. This formula shows that the change of position is dependent on the distance to the input pattern. The topology influences the change based on the function  $h(i, j, t)$ .

**Learning rule** The learning rule is the following:

$$\begin{aligned} \Delta c_j &= \eta(t) \cdot h(i, j, t) \cdot (p - c_j) \\ c_j(t+1) &= c_j(t) + \Delta c_j \end{aligned} \tag{4.22}$$

**The topology function  $h$ .** The topology function  $h(i, j, t)$  is defined in the grid and represents the relationship between neighbouring neurons. This means that this function will determine the topology of the trained network. It often is time dependent, but it is not necessary so. This function will take a large value if  $j$  is the the neighbour of, or even the winning neuron itself. Otherwise it will a small value. The topology function  $h(i, j, t)$  has only one maximum. This maximum is found next to winner neuron  $i$ , where the distance is 0.

**Introducing the distance.** For the topology function to be able to return large values next to the winner neuron and small values for non neighbouring neurons, it needs to know these distances. It needs to know how far  $i$  and  $j$  are away from each other. In a one-dimensional network the amount of connections between  $i$  and  $j$  can be used, in a two-dimensional the euclidean distance can be used.

**Common distance and topology functions.** One of common distance functions is the Gaussian bell function. The width of the bell function can be changed by applying the parameter  $\sigma$ . This parameter can be used to monotonically reduce the size of the neighbourhoods over time.

$$h(i, j, t) = e^{\left(-\frac{\|g_i - c_j\|^2}{e \cdot \sigma(t)^2}\right)} \tag{4.23}$$

$g_i$  and  $g_j$  represent the positions of the neurons on the grid. There are several other functions that can be used, but we will not elaborate much further on this.

**Learning rate.** Reducing the learning rate  $\eta$  in time in conjunction with the neighbourhood size  $\sigma$  is used to prevent the later training phases from pulling the entire map towards a new pattern. A  $\eta$  between 0.01 and 0.6 typically works well, but this is very dependant on the network topology and the neighbourhood size.

The advantage of a large learning rate and neighbourhood size at the start is that the network can unfold fast when starting out from the randomly initialized network. In the end when the network becomes more rigid, it will be able to fine-tune itself because of the smaller learning rate and neighbourhoods. However,  $h \cdot \eta$  should always be bigger than 1, otherwise the neurons will start missing the training sample.

## 4.4 Implementation

After all this theoretical research, an attempt was made to implement what had been learned. The goal was to try and solve a CAPTCHA by using neural networks for pattern recognition. Access to hundreds if not thousands of CAPTCHA images was needed. On top of that was it necessary to know the correct answer for each one of those CAPTCHAs. That is why it was decided to build a custom CAPTCHA generating framework.

### 4.4.1 Captcha builder

The main objective in creating this was to be able to precisely control the type of CAPTCHA image generated, and that a lot of images (with the respective answers) could be generated quickly. An attempt was made to find a suitable framework, but none were up to the task. As a result the development of a custom build framework was started. The resulting CAPTCHA builder is capable of

- rendering backgrounds range from transparent to two-coloured gradients.
- rendering text in
  - Character sets ranging from a reduced alphanumeric set to a set comprising all letters, numbers and punctuation.
  - Arabic font
  - Chinese characters
- rendering several styles of fonts, letter colouring or letter outlining
- rendering line noise onto the image
- generating distortions
  - fish-eye



- shear
- ripple
- stretch
- ...
- rendering a border around the image.

This all is done via an elaborate sequence of parsers and builder patterns<sup>8</sup>. The element defining what type of CAPTCHA image is created is the "build-String". The buildString is a line of text passed to the CaptchaBuilder, the builder then parses the text and sets up the different element builders, ready to create a CAPTCHA test. A few examples of buildStrings:

Background:

```
:BACKGROUND!TWO COLOR GRADIENT#COLORS1@RANGE*FF0000*000000
#COLORS2@LIST*00FF00*0000FF:
```

Border:

```
:BORDER!SOLID#COLORS@RANGE*000000*ffffff#THICKNESS@4:
```

Noise:

```
:NOISE!CURVED LINE#COLORS@RANGE*000000*ffffff#THICKNESS@2.3:
```

Gimp:

```
:GIMP!FISHEYE#DOUBLE1@3.7#DOUBLE2@7.6#
COLORS1@RANGE*000000*ffffff#COLORS2@LIST*a5f9b3*1586df*de48af
```

The parsing of the buildString happens in two stages, the long parse is when the whole string is analysed to build the ElementCreatorBuilders (again made based on the builder design pattern). These element creators builders are stored in an ArrayDeque in the CaptchaBuilder. After the long parse (named like this because it can take some time to parse a long buildString), the CaptchaBuilder is ready to generate actual CAPTCHA tests. When the method buildCaptcha() is called the Builder will perform the short parsing of the buildString, where it checks whether the order of the element creators builders inside the ArrayDeque is correct or not. When everything checks out, each of the element creators builders is called upon to make a fresh element creator that is used to add elements to the CAPTCHA or distort it. This whole process of short-parsing is essential for maintaining randomness. If you want to delve through the code, or use it yourself, feel free as

---

<sup>8</sup>As of writing, it can be noted that some of the parts of the builder patterns should be implemented more efficiently with extensibility in mind.

the CaptchaBuilder framework is licensed under the MIT license. A small piece of code worth looking at, is the ColorRangeContainer class. The CaptchaBuilder used to become real slow when using a lot of, or big color ranges as the option for one or more of the elements. Colours used to be stored as an ArrayList of integers and obviously all these integers needed to be generated. This is accomplished by looping through all the possibilities and storing them. Needless to say this took up a huge amount of time and memory, so a better way of storing these colour ranges was thought of. (Code A.1, page 55) Whenever a class stores a color, be it a single colour, multiple colours or a whole range of colours, it will do this with this utility class. It has a lot of different constructors, this is to account for the different options of representing colours in a programming environment. Once an object of the class has been instantiated, it can return a random colour from all the stored possibilities, it doesn't matter whether the possible colours is only 1 or all 4228250625 colours<sup>9</sup>. It returns this colour when its function getRandomColorInRange() is called. Here again there are several other methods available with the same functionality, that return a colour in most of the ways a colour can be represented grammatically.

As a final point of interest, the code for the CAPTCHA tests generated. These are objects instantiations of the Captcha class. This class is implemented as an immutable data object, where all the variables cannot be changed once the class has been instantiated. The Captcha objects themselves are safe to pass to other applications as their implementation resists change and manipulation, making them secure for use in the "outside world"<sup>10</sup>. (Code A.2, page 57). It can be concluded, all things are considered, that the framework does its job rather well. It can generate all different kinds of CAPTCHA images quickly and every CAPTCHA image will always have the same characteristics when same buildString is used.

#### 4.4.2 Neural networks

When trying to implement the neural networks one of the questions was how to start building the networks. Would it be opportune to write the neural networks from scratch, or would it be better to use one of the many neural network frameworks out there? The first idea was to create a custom neural network framework (much like the CaptchaBuilder framework). This option was quickly dropped, as it was a task of gargantuan proportions, and the framework would not be finished in time. Instead the focus was placed on finding a neural network framework that was able to accommodate the needs of this thesis, yet was still easy enough to

---

<sup>9</sup>The maximum amount of colours representable in the RGBa format

<sup>10</sup>The Framework could be adapted to work inside a website for example.

quickly implement the different types of neural networks. Three candidate neural network frameworks were found: Neuroph<sup>11</sup>, Encog<sup>12</sup> and Snipe<sup>13</sup>.

- Neuroph: Neuroph has a useful graphical interface that is easy to use. however it has the downside of quickly becoming slow. When testing Hopfield networks, Neuroph started slowing down quickly as the dimensions increased.
- Encog: Encog is a well documented framework with lots of examples available, it was able to maintain its speed much throughout the initial testing of Hopfield and multilayered perceptron networks.
- Snipe: Snipe is like the Swiss army knife of the neural network frameworks, it can do anything. Snipe is adaptable and customizable to your needs, the trade off is that due to all this potential the framework is very complex to hard understand and use.

Encog was chosen as the framework to use, as it came out of the test as having good performance under stress and easy to implement.

### Implementing the networks

The hardest thing to implement was the conversion from an image with a letter to an input set the network could use. For this a class was made that iterates over the image pixels and converts the image from an array of pixels to an array of double values. When a pixel has a colour value that falls between the range specified, the corresponding double will be set to +1. If the color is not in the specified range, the double value will become -1. This could be changed to increase when needed, but so far I have not seen any beneficial effects of increasing the gap between those two values. (Code A.3, page 60)

**Creating the Hopfield network.** The first network implemented was the Hopfield network (Code A.5, page 65). This was straightforward to implement thanks to the fact that the Encog framework has default implementation of a Hopfield network. The functionalities (build, train and evaluate) have been implemented so that all the needed steps are executed in one method call. A state pattern might prove a useful extra extension to prevent the execution of one function when it cannot be executed yet (e.g. training the network when it hasn't been build yet).

---

<sup>11</sup><http://neuroph.sourceforge.net>

<sup>12</sup><http://www.heatonresearch.com/encog>

<sup>13</sup><http://www.dkriesel.com/en/tech/snipe>

**Testing out the network** Images of letters weren't immediately used when first testing out this network, instead generic 10X10 patterns were used to see whether the network was capable of recognizing patterns. The first results were promising so the implementation was adjusted to accommodate for using the input from the letters. The network was expanded to receive input for each pixel of the image containing one character (40X50) (Figure B.7, page 78) resulting in a massive network of 2000 neurons. The training of the network was very easy, again thanks to the Encog framework. However, when evaluating the network it quickly became apparent that the implementation of the Hopfield network was flawed. The patterns trained were all combined into a single pattern (Figure B.8, page 79). As stated before, a Hopfield network is only capable of storing a maximum of  $\approx 0.139 \cdot |H|$  patterns, with  $H$  being the total amount of neurons. This was a problem that was not identified quickly, seeing that the network should be able to theoretically store a maximum of 278 reconstructible patterns, being far more than the 45 patterns being trained. A more detailed study has shown that the theoretical maximum amount of reconstructible patterns overshoots the real value by a magnitude 10. The storage capacity of a Hopfield network decreases significantly if the patterns are highly correlated (which is the case when training the network for character recognition). A more realistic storage capacity for Hopfield networks training character patterns using the Hebbian rule is  $\approx 0.012$  [Gang and Zheyuan]. Gang Wei and Zheyuan Yu also propose a better learning rule, namely the Pseudo-inverse Rule. The pseudo-inverse rule has a technical maximum capacity of  $p = 2|H|$ , but again, realistically the pseudo-inverse rule has a storage capacity of  $\approx 0.064$  when training for character recognition. Using the pseudo-inverse rule the network would need to consist out of at least 704 neurons to store 45 character patterns. The network as implemented should be able to store and recognize the patterns correctly once the pseudo-inverse learning rule is implemented. Unfortunately Encog did not allow to quickly change the learning rule, as only the Hebbian rule is default implemented and the API for writing a custom learning rule did not bring much clarity.

**Creating the multi layered perceptron.** The second network implemented was the MLP network (Code A.6, page 68). This network was just as straightforward to implement as the Hopfield network thanks to the Encog framework. Just like the Hopfield network the functionalities have been implemented so that they can be called with a single method call. The building of the network is capable of adding several hidden layers. The training method can be specified during the initialization of the network. once the training has started, the method cannot be changed any more.

**Testing out the network** This network was immediately implemented to use the generated letter images (Figure B.7, page 78) as input. For this purpose the retina layer was made up out of 2000 input neurons, one neuron for each pixel in the letter image. By utilizing the previously mentioned ImageToArray utility class this was easily done. The output was chosen to be made up out of 8 neurons, each representing a bit value (the neurons should use the binary threshold function, or the output should be normalized to a range of  $\{0, 1\}$ ).

An batch of tests has been taken trying to find an optimal configuration for the perceptron network<sup>14</sup> The network topology was varied from not using an hidden layer to using a hidden layer with up to 4500 neurons in 500 step increments. Each topology was then trained to an accuracy of 10%, 1% and 0.1%. This resulted in 30 different configurations to test. for each configuration the network was trained with the images found on page 78. The network then was evaluated against 108 character images randomly taken from the training set and against 500 newly generated characters without and with distortion. The results of the are collected in table C.1 on page 83. The time required for training perceptron networks increases linearly in function of the amount of neurons to train. The network operates very fast once it has finished training. The time required to process an input pattern varies from  $4.5414e^{-05}$  seconds and 0.01536286 seconds, averaging a processing time of 0.005196 seconds.<sup>15</sup>

There are some network configurations that perform better than others. While the results are incomplete, they do hint towards a configuration containing a hidden layer. Browning and Kolas provide a detailed overview of the influence each variable component has on the perceptron network [Browning and Kolas, 2007]. For further research into implementing and training perceptron network. Jiang and colleagues provide an interesting alternative method of training feedforward networks based on data clustering [Jiang and Kam Sie Wah, 2003], a more modern approach that according to their research paper has demonstrated its superiority over competing methods.

**Creating the Kohonen network.** Again, implementing this network was very easy thanks to the Encog framework. The functionalities are, just like with the previous two network implementations, programmed so that they can be called with a single method call. The difference here is that we cannot know what character the network recognizes. The only feedback we get is knowing what neuron is

---

<sup>14</sup>Time measurements are the results using a computer with an Intel i7 870 with 16 Gb 1333 MHz RAM and 1Gb Nvidia GeForce GTX 460.

<sup>15</sup>This is the average time taken from the test samples comprising 17632 images.

firing. We have to map the output neuron firing to the input pattern triggering it to overcome this problem. This mapping is done right after the network has been trained.

**Testing out the network** Using the network gave no significant result, the total error of the network would not drop below 14%. Whichever configuration was tried, varying the learning rate, the learning rule, the radius all did not yield any significant improvements. The gradient of the error curve consistently neared zero after a 3 to 4 iterations. To my surprise increasing the learning rate and neighbourhood sizes, in an attempt to keep  $h \cdot \eta$  above 1 did not result in a significant change in the error output. A possible cause of this plateau might be that the network is not unfolding right. This could mean that the network has topological defects. A quick solution for these possible defects has not been found so far. The implementation makes it easy to implement a multi-dimensional neighbourhood, but since the network did not perform as planned a more detailed look into the use and effects of multi-dimensional neighbourhoods was not possible. Törmä [Törmä, 1995] provides us with some additional information about the effects of 1D vs 2D neighbourhoods. Unfortunately verifying these findings in our set-up was not possible.

# Chapter 5

## Conclusion

It can be said that the results of the experimental phase were rather disappointing. Two of the implemented neural networks failed to either complete the training or recognize any pattern correctly. The implemented perceptron network yielded better results. The network in its various configurations was able to recognize 19% of the undistorted single character images. When a distortion was introduced, the recognition rate dropped to 16%. While characters were recognized, these results are not significant enough to speak of a successful implementation. The problems encountered could be solved when more research and development time would be put into these problems.

- Kohonen network: The error here is probably due to a knot in the network topology. Kazushi Murakoshi and Yuichi Sato have written an article about how to reduce topological defects in self organizing maps [Murakoshi and Sato, 2006]. This article might be a good starting point for further investigation.
- Hopfield network: The problem here is due to the fact that Hopfield networks have a low maximum storage capacity when dealing with highly related patterns such as letters and numbers. The implementation of the pseudo-inverse learning rule instead of the Hebbian rule might already be enough to fix this problem.
- Perceptron: The biggest challenge here is trying to find an optimal network topology, learning rule and propagation type for dealing with the problems of pattern recognition. It would have been interesting to further investigate the subject of an generalized optimal perceptron set-up for pattern recognition, but this is a whole new research project on it's own.
- Input: One of the last suspects is that the input layer has too much input neurons making the network prone to 'overlearning'. This problem can be

solved by trimming of the whitespace and then resampling the image to a lower resolution. When implementing the pseudo inverse learning rule a Hopfield network would need to consist out of atleast 704 neurons, for that reason would might it be a good choice to choose a resampling rate so that there are still at least 750 pixels left in the image.

Character recognition in general has dramatically improved of the last years, so much that computers are now outperforming humans when challenged with single characters [Chellapilla and Larson, 2005]. So how do the neural networks hold up after the research performed in this thesis?

We can conclude that neural networks are a valid option for pattern recognition in general. Neural networks have been used for pattern recognition purposes like the detection of defect objects in the production, medical diagnosis, navigation and guidance systems, target recognition systems, even license plate recognition, fingerprint analysis and document processing [Egmont-Petersen et al., 2002]. It must be said that solving CAPTCHAs with neural networks is not the most optimal solution, despite the fact that theoretically they are a plausible candidate. The main problem lies in the fact that CAPCTHAs are a very dynamic thing. Once it is found out that a certain CAPTCHA style is no longer effective at keeping the spammers at bay, that system will be adapted slightly, so that the task of configuring and training the neural networks needs to be done again. On top of that, one must be in possession of a large enough sample with correct answers to be able to successfully train the network. There also is an economic incentive for spammers not to work with neural networks. There are far easier and cheaper methods of circumventing the CAPTCHAs.

One of these methods is paying a service for service like "Death by Captcha"<sup>1</sup> where you pay for having your CAPTCHAs solved. The rate is as low as \$0.5/1.000 CAPTCHAs solved correctly. If the theoretical cost of creating a fully functional CAPTCHA solver is placed at \$10.000 then such a system would need to solve 20 million CAPTCHAs successfully before it would become profitable. A strong argument for using paid labour to solve CAPTCHAs is that it has a big succes rate, and that is is futureproof, the CAPTCHAs are not "broken". They are solved by humans in normal operation, so the reverse Turing test is not violated.

Academically there is a great deal to be learned from trying to solve the pattern recognition problems, but in a real world economical context it no longer is a valid option.

---

<sup>1</sup><http://deathbycaptcha.com/>, accesed on 2310-06-05



# Appendix A

## Code

### A.1 ColorRangeContainer.java

**Listing A.1:** The ColorRangeContainerClass, used for storing the colors used in the elements and randomly getting one of the available colors.

```
/*
 * The MIT License
 *
 * Copyright 2013 Pieter Van Eeckhout.
 *
 * Permission is hereby granted, free of charge, to any person obtaining a
 * copy
 * of this software and associated documentation files (the "Software"), to
 * deal
 * in the Software without restriction, including without limitation the
 * rights
 * to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
 * copies of the Software, and to permit persons to whom the Software is
 * furnished to do so, subject to the following conditions:
 *
 * The above copyright notice and this permission notice shall be included in
 * all copies or substantial portions of the Software.
 *
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
 * IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
 * FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
 * AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
 * LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
 * FROM,
 * OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN
 * THE SOFTWARE.
 */
package bachelorthesis.captchabuilder.util;

import bachelorthesis.captchabuilder.util.enums.CaptchaConstants;
import java.awt.Color;
import java.util.List;
import java.util.Random;

/**
```

```

* ColorRangeContainer.java (UTF-8)
*
* usage and functionality here
*
* 2013/04/19
*
* @author Pieter Van Eeckhout <vaneeckhout.pieter@gmail.com>
* @author Pieter Van Eeckhout <pieter.vaneeckhout.q1295@student.hogent.be>
* @author Hogent StudentID <2000901295>
* @since 1.1.0
* @version 1.1.0
*/
public class ColorRangeContainer {

    private final int startR;
    private final int endR;
    private final int startG;
    private final int endG;
    private final int startB;
    private final int endB;
    private final int startA;
    private final int endA;
    private Random random;
    private boolean listMode;
    private List<String> hexList;

    /**
     * Constructor
     * <p/>
     * @param MSa a colour in MSaccess format
     */
    public ColorRangeContainer(int MSa) {
        this(MSa, MSa);
    }

    /**
     * constructor
     * <p/>
     * @param hexList a list of colours in hexadecimal form
     */
    public ColorRangeContainer(List<String> hexList) {
        this(0);
        this.listMode = true;
        this.hexList = hexList;
    }

    /**
     * constructor
     * <p/>
     * @param rgba a collection of colours in RGBA format
     */
    public ColorRangeContainer(int[] rgba) {
        this(rgba, rgba);
    }

    /**
     * Constructor
     * <p/>
     * @param r a colour's red value
     * @param g a colour's green value
     * @param b a colour's blue value
     */

```

```
public ColorRangeContainer(int r, int g, int b) {
    this(r, g, b, 0);
}

/**
 * constructor
 * <p/>
 * @param r a colour's red value
 * @param g a colour's green value
 * @param b a colour's blue value
 * @param a a colour's alpha value
 */
public ColorRangeContainer(int r, int b, int g, int a) {
    this(r, r, g, g, g, a, a);
}

/**
 * constructor
 * <p/>
 * @param startRGBa the start of a colour range in RGBa format
 * @param endRGBa the end of a colour range in RGBa format
 */
public ColorRangeContainer(int[] startRGBa, int[] endRGBa) {
    this(startRGBa[0], endRGBa[0], startRGBa[1], endRGBa[1], startRGBa[2],
        endRGBa[2], startRGBa[3], endRGBa[3]);
}

/**
 * constructor
 * <p/>
 * @param startMSa the start of a colour range in MSaAcces format
 * @param endMSa the start of a colour range in MSaAcces format
 */
public ColorRangeContainer(int startMSa, int endMSa) {
    this(ImageUtil.msAccesToRGBa(startMSa), ImageUtil.msAccesToRGBa(endMSa));
}

/**
 * constructor
 * <p/>
 * @param startR the start of a colour range red value
 * @param endR the end of a colour range red value
 * @param startG the start of a colour range green value
 * @param endG the end of a colour range green value
 * @param startB the start of a colour range blue value
 * @param endB the end of a colour range blue value
 * @param startA the start of a colour range alpha value
 * @param endA the end of a colour range red value
 */
public ColorRangeContainer(int startR, int endR, int startG, int endG, int
    startB,
                           int endB, int startA, int endA) {
    this.random = CaptchaConstants.RANDOM;
    this.startR = startR;
    this.endR = endR;
    this.startG = startG;
    this.endG = endG;
    this.startB = startB;
    this.endB = endB;
    this.startA = startA;
    this.endA = endA;
}
```

```

        this.listMode = false;
    }

    /**
     * picks a random colour in the range and returns it
     * <p/>
     * @return a colour object
     */
    public Color getRandomColorInRange() {
        return new Color(getRandomInRangeR(), getRandomInRangeG(),
            getRandomInRangeB(), getRandomInRangeA());
    }

    /**
     * picks a random colour in the range and returns it
     * <p/>
     * @return a colour in MSAcces format
     */
    public int getRandomMSAccesInRange() {
        return ImageUtil.rgbToMsAcces(getRandomInRangeR(), getRandomInRangeG(),
            getRandomInRangeB());
    }

    private int getRandomInRangeR() {
        if (listMode) {
            return ImageUtil.hexadecimalToRGBa(hexList.get(random.nextInt(
                hexList.size())))[0];
        } else {
            return random8bitNumber(startR, endR);
        }
    }

    private int getRandomInRangeG() {
        if (listMode) {
            return ImageUtil.hexadecimalToRGBa(hexList.get(random.nextInt(
                hexList.size())))[1];
        } else {
            return random8bitNumber(startG, endG);
        }
    }

    private int getRandomInRangeB() {
        if (listMode) {
            return ImageUtil.hexadecimalToRGBa(hexList.get(random.nextInt(
                hexList.size())))[2];
        } else {
            return random8bitNumber(startB, endB);
        }
    }

    private int getRandomInRangeA() {
        if (listMode) {
            return ImageUtil.hexadecimalToRGBa(hexList.get(random.nextInt(
                hexList.size())))[3];
        } else {
            return random8bitNumber(startA, endA);
        }
    }

    private int random8bitNumber(int start, int end) {
        if (start > end) {

```

```
        if (random.nextBoolean()) {
            return random8bitNumber(0, end);
        } else {
            return random8bitNumber(start, 256);
        }
    }
    if (start == end) {
        return start;
    } else {
        return random.nextInt(end - start) + start;
    }
}
```

## A.2 Captcha.java

**Listing A.2:** The Captcha test.

```
/*
 * The MIT License
 *
 * Copyright 2013 Pieter Van Eeckhout.
 *
 * Permission is hereby granted, free of charge, to any person obtaining a
 * copy
 * of this software and associated documentation files (the "Software"), to
 * deal
 * in the Software without restriction, including without limitation the
 * rights
 * to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
 * copies of the Software, and to permit persons to whom the Software is
 * furnished to do so, subject to the following conditions:
 *
 * The above copyright notice and this permission notice shall be included in
 * all copies or substantial portions of the Software.
 *
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
 * IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
 * FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
 * AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
 * LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
 * FROM,
 * OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN
 * THE SOFTWARE.
 */
package bachelorthesis.captchabuilder;

import java.io.Serializable;
import java.awt.image.BufferedImage;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.util.Date;
import javax.imageio.ImageIO;

/**
 * Captcha.java (UTF-8)
```

```

*
* Captcha object, contains the image, the answer, buildstring used to create
* the image and the date the captcha was create.
*
* 2013/04/17
*
* @author Pieter Van Eeckhout <vaneeckhout.pieter@gmail.com>
* @author Pieter Van Eeckhout <pieter.vaneeckhout.q1295@student.hogent.be>
* @author Hogent StudentID <2000901295>
* @since 1.0.7
* @version 1.0.7
*/
public class Captcha implements Serializable {

    private static final long serialVersionUID = 617954136L;
    private String answer;
    private String buildSequence;
    private boolean caseSensitive;
    private Date timestamp;
    private BufferedImage captchImage;

    /**
     * Constructor
     * <p/>
     * @param buildSequence
     * @param answer
     * @param caseSensitive
     * @param captchImage
     * @param timestamp
     */
    public Captcha(String buildSequence, String answer, boolean caseSensitive,
        BufferedImage captchImage, Date timestamp) {
        this.buildSequence = buildSequence;
        this.answer = answer;
        this.captchImage = captchImage;
        this.timestamp = timestamp;
        this.caseSensitive = caseSensitive;
    }

    /**
     * Validates if the string passed matches the answer stored
     * <p/>
     * @param response the response given by the user
     * <p/>
     * @return true or false
     */
    public boolean isCorrect(String response) {
        if (caseSensitive) {
            return answer.equals(response);
        } else {
            return answer.equalsIgnoreCase(response);
        }
    }

    public String getAnswer() {
        return answer;
    }

    public BufferedImage getImage() {
        return captchImage;
    }
}

```

```
public Date getTimeStamp() {
    return timestamp;
}

public String getBuildSequence() {
    return buildSequence;
}

public boolean isCaseSensitive() {
    return caseSensitive;
}

public Date getTimeStamp() {
    return timestamp;
}

@Override
public String toString() {
    return new StringBuilder()
        .append("[Answer: ")
        .append(answer)
        .append("][CaseSensitive: ")
        .append(caseSensitive)
        .append("][Timestamp: ")
        .append(timestamp)
        .append("][Image: ")
        .append(captchaImage)
        .append("][BuildSequence: ")
        .append(buildSequence)
        .append("]")
        .toString();
}

private void writeObject(ObjectOutputStream out) throws IOException {
    out.writeObject(buildSequence);
    out.writeObject(answer);
    out.writeObject(caseSensitive);
    out.writeObject(timestamp);
    ImageIO.write(captchaImage, "png", ImageIO.createImageOutputStream(out));
}

private void readObject(ObjectInputStream in) throws IOException,
    ClassNotFoundException {
    buildSequence = (String) in.readObject();
    answer = (String) in.readObject();
    caseSensitive = (Boolean) in.readObject();
    timestamp = (Date) in.readObject();
    captchaImage = ImageIO.read(ImageIO.createImageInputStream(in));
}
}
```

## A.3 ImageToArray.java

**Listing A.3:** Utility class to convert images to array representations.

```
/*
```

```

* The MIT License
*
* Copyright 2013 Pieter Van Eeckhout.
*
* Permission is hereby granted, free of charge, to any person obtaining a
* copy
* of this software and associated documentation files (the "Software"), to
* deal
* in the Software without restriction, including without limitation the
* rights
* to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
* copies of the Software, and to permit persons to whom the Software is
* furnished to do so, subject to the following conditions:
*
* The above copyright notice and this permission notice shall be included in
* all copies or substantial portions of the Software.
*
* THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
* IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
* FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
* AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
* LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
* FROM,
* OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN
* THE SOFTWARE.
*/
package bachelorthesis.captchacleanup;

import java.awt.image.BufferedImage;

/**
 * ImageToArray.java (UTF-8)
 *
 * Utility class images
 *
 * 2013/05/20
 *
 * @author Pieter Van Eeckhout <vaneeckhout.pieter@gmail.com>
 * @author Pieter Van Eeckhout <pieter.vaneeckhout.q1295@student.hogent.be>
 * @author Hogent StudentID <2000901295>
 * @since 1.0.0
 * @version 1.0.0
 */
public class ImageToArray {

    /**
     * extracts the image data, all pixels within the colour range return true
     *
     * @param image the image to be analysed
     * @param startRange the start colour in MSA format
     * @param endRange the end colour in MSA format
     * <p/>
     * @return an array with the boolean data
     */
    public static boolean[][] colorRangeToBooleanArray(BufferedImage image,
                                                         int startRange,
                                                         int endRange) {
        boolean[][] array = new boolean[image.getWidth()][image.getHeight()];
        int startR = (startRange >> 16) & 0x000000FF;
        int startG = (startRange >> 8) & 0x000000FF;
        int startB = (startRange) & 0x000000FF;
        int endR = (endRange >> 16) & 0x000000FF;

```



```
int endG = (endRange >> 8) & 0x000000FF;
int endB = (endRange) & 0x000000FF;

for (int y = 0; y < image.getHeight(); y++) {
    for (int x = 0; x < image.getWidth(); x++) {
        int RGB = image.getRGB(x, y);
        int alpha = (RGB >> 24) & 0x000000FF;
        boolean inRange = false;
        if (alpha != 0) {
            int R = (startRange >> 16) & 0x000000FF;
            int G = (startRange >> 8) & 0x000000FF;
            int B = (startRange) & 0x000000FF;
            if (startR <= R && R <= endR && startG <= G && G <= endG
                &&
                startB <= B && B <= endB) {
                inRange = true;
            }
        }
        array[x][y] = inRange;
    }
}
return array;
}

/**
 * extracts the image data, all pixels within the colour range return 1,
 * the
 * others return 0
 *
 * @param image the image to be analysed
 * @param startRange the start colour in MSA format
 * @param endRange the end colour in MSA format
 * <p/>
 * @return an array with the double data
 */
public static double [][] colorRangeToDoubleArray(BufferedImage image,
                                                    int startRange,
                                                    int endRange, double
                                                    inRangeValue, double
                                                    outRangeValue) {
    double [][] array = new double[image.getWidth()][image.getHeight()];
    int startR = (startRange >> 16) & 0x000000FF;
    int startG = (startRange >> 8) & 0x000000FF;
    int startB = (startRange) & 0x000000FF;
    int endR = (endRange >> 16) & 0x000000FF;
    int endG = (endRange >> 8) & 0x000000FF;
    int endB = (endRange) & 0x000000FF;

    for (int y = 0; y < image.getHeight(); y++) {
        for (int x = 0; x < image.getWidth(); x++) {
            int RGB = image.getRGB(x, y);
            int alpha = (RGB >> 24) & 0x000000FF;
            if (alpha != 0) {
                int R = (startRange >> 16) & 0x000000FF;
                int G = (startRange >> 8) & 0x000000FF;
                int B = (startRange) & 0x000000FF;
                if (startR <= R && R <= endR && startG <= G && G <= endG
                    &&
                    startB <= B && B <= endB) {
                    array[x][y] = inRangeValue;
                } else {

```

```

        array[x][y] = outRangeValue;
    }
}
}
return array;
}
}

```

## A.4 TrainingSet.java

**Listing A.4:** Utility class to the training patterns.

```

/*
 * The MIT License
 *
 * Copyright 2013 Pieter Van Eeckhout.
 *
 * Permission is hereby granted, free of charge, to any person obtaining a
 * copy
 * of this software and associated documentation files (the "Software"), to
 * deal
 * in the Software without restriction, including without limitation the
 * rights
 * to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
 * copies of the Software, and to permit persons to whom the Software is
 * furnished to do so, subject to the following conditions:
 *
 * The above copyright notice and this permission notice shall be included in
 * all copies or substantial portions of the Software.
 *
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
 * IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
 * FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
 * AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
 * LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
 * FROM,
 * OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN
 * THE SOFTWARE.
 */
package bachelorthesis.neuralnetworks.util;

/**
 * TrainingSet.java (UTF-8)
 *
 * class that stores the input and output training patterns for neural network
 * training.
 *
 * 2013/06/04
 *
 * @author Pieter Van Eeckhout <vaneeckhout.pieter@gmail.com>
 * @author Pieter Van Eeckhout <pieter.vaneeckhout.q1295@student.hogent.be>
 * @author Hogent StudentID <2000901295>
 * @since 1.0.0
 * @version 1.0.0
 */
public class TrainingSet {

```

```
private final int inputCount;
private final int outputCount;
private double input[][];
private double output[][];
private int trainingSetCount;

public TrainingSet(int inputCount, int outputCount) {
    this.inputCount = inputCount;
    this.outputCount = outputCount;
    trainingSetCount = 0;
}

public int getInputCount() {
    return inputCount;
}

public int getOutputCount() {
    return outputCount;
}

public void setTrainingSetCount(int trainingSetCount) {
    this.trainingSetCount = trainingSetCount;
    input = new double[trainingSetCount][inputCount];
    output = new double[trainingSetCount][outputCount];
}

public int getTrainingSetCount() {
    return trainingSetCount;
}

public void setInput(int set, int index, double value)
    throws RuntimeException {
    if ((set < 0) || (set >= trainingSetCount)) {
        throw (new RuntimeException("Training_set_out_of_range:" + set));
    }
    if ((index < 0) || (index >= inputCount)) {
        throw (new RuntimeException("Training_input_index_out_of_range:" +
            index));
    }
    input[set][index] = value;
}

public void setOutput(int set, int index, double value)
    throws RuntimeException {
    if ((set < 0) || (set >= trainingSetCount)) {
        throw (new RuntimeException("Training_set_out_of_range:" + set));
    }
    if ((index < 0) || (set >= outputCount)) {
        throw (new RuntimeException("Training_input_index_out_of_range:" +
            index));
    }
    output[set][index] = value;
}

public double getInput(int set, int index)
    throws RuntimeException {
    if ((set < 0) || (set >= trainingSetCount)) {
        throw (new RuntimeException("Training_set_out_of_range:" + set));
    }
    if ((index < 0) || (index >= inputCount)) {
```

```

        throw (new RuntimeException(" Training_input_index_out_of_range:" +
            index));
    }
    return input[set][index];
}

public double getOutput(int set, int index)
    throws RuntimeException {
    if ((set < 0) || (set >= trainingSetCount)) {
        throw (new RuntimeException(" Training_set_out_of_range:" + set));
    }
    if ((index < 0) || (set >= outputCount)) {
        throw (new RuntimeException(" Training_input_index_out_of_range:" +
            index));
    }
    return output[set][index];
}

public double[] getOutputSet(int set)
    throws RuntimeException {
    if ((set < 0) || (set >= trainingSetCount)) {
        throw (new RuntimeException(" Training_set_out_of_range:" + set));
    }
    return output[set];
}

public double[] getInputSet(int set)
    throws RuntimeException {
    if ((set < 0) || (set >= trainingSetCount)) {
        throw (new RuntimeException(" Training_set_out_of_range:" + set));
    }
    return input[set];
}

public void setInputSet(int set, double[] input) throws RuntimeException {
    if ((set < 0) || (set >= trainingSetCount)) {
        throw (new RuntimeException(" Training_set_out_of_range:" + set));
    }
    this.input[set] = input;
}

public void setOutputSet(int set, double[] output) throws RuntimeException
{
    if ((set < 0) || (set >= trainingSetCount)) {
        throw (new RuntimeException(" Training_set_out_of_range:" + set));
    }
    this.output[set] = output;
}

public double[][] getInput() {
    return input;
}

public void setInput(double[][] input) {
    this.input = input;
}

public double[][] getOutput() {
    return output;
}

public void setOutput(double[][] output) {
    this.output = output;
}

```

```
}  
}
```

## A.5 EncogHopfieldNetwork.java

**Listing A.5:** The Hopfield network.

```
/*  
 * The MIT License  
 *  
 * Copyright 2013 Pieter Van Eeckhout.  
 *  
 * Permission is hereby granted, free of charge, to any person obtaining a  
 *   copy  
 * of this software and associated documentation files (the "Software"), to  
 *   deal  
 * in the Software without restriction, including without limitation the  
 *   rights  
 * to use, copy, modify, merge, publish, distribute, sublicense, and/or sell  
 * copies of the Software, and to permit persons to whom the Software is  
 * furnished to do so, subject to the following conditions:  
 *  
 * The above copyright notice and this permission notice shall be included in  
 * all copies or substantial portions of the Software.  
 *  
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR  
 * IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,  
 * FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE  
 * AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER  
 * LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING  
 * FROM,  
 * OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN  
 * THE SOFTWARE.  
 */  
package bachelorthesis.neuralnetworks.network.encog.hopfield;  
  
import bachelorthesis.neuralnetworks.network.NeuralNetwork;  
import bachelorthesis.neuralnetworks.util.TrainingSet;  
import org.encog.ml.data.specific.BiPolarNeuralData;  
import org.encog.neural.thermal.HopfieldNetwork;  
  
/**  
 * EncogHopfieldNetwork.java (UTF-8)  
 *  
 * Provides a configurable Encog HopfieldNetwork  
 *  
 * 2013/05/19  
 *  
 * @author Pieter Van Eeckhout <vaneeckhout.pieter@gmail.com>  
 * @author Pieter Van Eeckhout <pieter.vaneeckhout.q1295@student.hogent.be>  
 * @author Hogent StudentID <2000901295>  
 * @since 1.0.0  
 * @version 1.0.0  
 */  
public class EncogHopfieldNetwork extends NeuralNetwork {  
    private HopfieldNetwork network;
```

```

private int neuroncount;

/**
 * Constructor
 * <p/>
 * @param id the network id
 */
protected EncogHopfieldNetwork(int id) {
    super(id);
}

@Override
public void buildAndTrainNetwork(TrainingSet trainingSet) {
    neuroncount = trainingSet.getInputCount();
    System.out.println(" Building_hopfield_network");
    network = new HopfieldNetwork(trainingSet.getInputCount());

    network.reset();
    System.out.println(" Training_hopfield_network");
    long startTimeLong = System.nanoTime();
    for (double[] ds : trainingSet.getInput()) {
        network.addPattern(doubleArrayToBiPolarNeuralData(ds));
    }
    long endTimeLong = System.nanoTime();
    double durationInSec = (double) ((endTimeLong - startTimeLong) / Math
        .pow(10, 9));
    System.out.println(" Finished_training_network_in:" + durationInSec);
}

private BiPolarNeuralData doubleArrayToBiPolarNeuralData(double[] data) {
    BiPolarNeuralData patternData = new BiPolarNeuralData(neuroncount);
    if (data.length != neuroncount) {
        IndexOutOfBoundsException e = new IndexOutOfBoundsException(
            " the_size_of_the_trainingsinputs_is_different_from_the_"
            + "amount_"
            + "of_input_neurons");
        throw e;
    }
    patternData.setData(data);
    return patternData;
}

@Override
public double[] evaluate(double[] input, int maxIterations) {
    System.out.println(" hopfield_network_evaluating_with_max_iterations:"
        + maxIterations);
    BiPolarNeuralData inputPattern = doubleArrayToBiPolarNeuralData(input);
    ;
    network.setCurrentState(inputPattern);
    int cycles = network.runUntilStable(maxIterations);
    System.out.println(" Cycles_until_stable(max_" + maxIterations + "):_"
        + cycles);
    BiPolarNeuralData outputPattern = (BiPolarNeuralData) network
        .getCurrentState();
    return outputPattern.getData();
}

public static String convertForDisplay(double[] inputPattern, double[]
    outputPattern, int width, int height) {
    int index1 = 0;
    int index2 = 0;
    StringBuilder block = new StringBuilder();

```

```
        for (int row = 0; row < height; row++) {

            for (int col = 0; col < width; col++) {
                if (inputPattern[index1++] == 1) {
                    block.append('O');
                } else {
                    block.append(' ');
                }
            }

            block.append(" --->---");

            for (int col = 0; col < width; col++) {
                if (outputPattern[index2++] == 1) {
                    block.append('O');
                } else {
                    block.append(' ');
                }
            }

            block.append("\n");
        }

        return block.toString();
    }

    @Override
    public String getLayerLayout() {
        return "[" + neuroncount + " ]";
    }
}
```

## A.6 EncogPerceptronNetwork.java

**Listing A.6:** The multilayered perceptron network.

```
/*
 * The MIT License
 *
 * Copyright 2013 Pieter Van Eeckhout.
 *
 * Permission is hereby granted, free of charge, to any person obtaining a
 * copy
 * of this software and associated documentation files (the "Software"), to
 * deal
 * in the Software without restriction, including without limitation the
 * rights
 * to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
 * copies of the Software, and to permit persons to whom the Software is
 * furnished to do so, subject to the following conditions:
 *
 * The above copyright notice and this permission notice shall be included in
 * all copies or substantial portions of the Software.
 *
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
```

```

* IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
* FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
* AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
* LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
  FROM,
* OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN
* THE SOFTWARE.
*/
package bachelorthesis.neuralnetworks.network.encog.perceptron;

import bachelorthesis.neuralnetworks.network.NeuralNetwork;
import static bachelorthesis.neuralnetworks.network.encog.perceptron.
    PropagationType.ManhattanPropagation;
import bachelorthesis.neuralnetworks.util.TrainingSet;
import java.util.List;
import org.encog.engine.network.activation.ActivationSigmoid;
import org.encog.ml.data.MLDataSet;
import org.encog.ml.data.basic.BasicMLDataSet;
import org.encog.ml.train.MLTrain;
import org.encog.ml.train.strategy.Strategy;
import org.encog.neural.networks.BasicNetwork;
import org.encog.neural.networks.layers.BasicLayer;
import org.encog.neural.networks.training.propagation.back.Backpropagation;
import org.encog.neural.networks.training.propagation.manhattan.
    ManhattanPropagation;
import org.encog.neural.networks.training.propagation.resilient.
    ResilientPropagation;
import org.encog.neural.networks.training.propagation.scg.
    ScaledConjugateGradient;
import org.encog.util.simple.EncogUtility;

/**
 * EncogPerceptronNetwork.java (UTF-8)
 *
 * Provides a configurable Encog BasicNetwork
 *
 * 2013/05/19
 *
 * @author Pieter Van Eeckhout <vaneeckhout.pieter@gmail.com>
 * @author Pieter Van Eeckhout <pieter.vaneeckhout.q1295@student.hogent.be>
 * @author Hogent StudentID <2000901295>
 * @since 1.0.0
 * @version 1.1.0
 */
public class EncogPerceptronNetwork extends NeuralNetwork {

    private BasicNetwork network;
    private int[] hiddenLayers;
    private double accuracy;
    private double learningRate;
    private List<Strategy> trainingStrategies;
    private PropagationType propagationType;
    private int outputCount;

    /**
     * Constructor
     *
     * @param id the id of the network
     * @param hiddenLayers the amount of neuron each hidden layer has (
     *     in
     *     order)
     * @param accuracy the desired accuracy

```



```
* @param learningRate      the learning rate (only used with
*                           ManhattanPropagation)
* @param trainingStrategies the training strategies to be used
*/
protected EncogPerceptronNetwork(int id, int[] hiddenLayers,
                                   double accuracy, double learningRate,
                                   List<Strategy> trainingStrategies,
                                   PropagationType propagationType) {

    super(id);
    this.hiddenLayers = hiddenLayers;
    this.accuracy = accuracy;
    this.learningRate = learningRate;
    this.trainingStrategies = trainingStrategies;
    this.propagationType = propagationType;
}

@Override
public void buildAndTrainNetwork(TrainingSet trainingSet) {
    System.out.println("Building basic network");
    this.network = new BasicNetwork();

    this.outputCount = trainingSet.getOutputCount();

    System.out.println("Adding layers to network");
    network.addLayer(new BasicLayer(null, true, trainingSet.getInputCount()
    ));
    if (hiddenLayers != null) {
        for (int i : hiddenLayers) {
            network.addLayer(
                new BasicLayer(new ActivationSigmoid(), true, i));
        }
    }
    network.addLayer(new BasicLayer(new ActivationSigmoid(), true,
    trainingSet.getOutputCount()));

    network.getStructure().finalizeStructure();
    network.reset();

    System.out.println("initializing network training system");
    MLDataSet trainingData = new BasicMLDataSet(trainingSet.getInput(),
    trainingSet.getOutput());
    final MLTrain training;

    switch (propagationType) {
        case Backpropagation:
            training = new Backpropagation(network, trainingData);
            break;
        case ManhattanPropagation:
            training = new ManhattanPropagation(network, trainingData,
            learningRate);
            break;
        case ResilientPropagation:
            training = new ResilientPropagation(network, trainingData);
            break;
        case ScaledConjugateGradient:
            training = new ScaledConjugateGradient(network, trainingData);
            break;
        default:
            IllegalArgumentException e = new IllegalArgumentException(
                "Unknown propagationType");
            throw e;
    }
}
```

```

        System.out.println(" Propagation:␣" + propagationType.name());

        System.out.println("adding␣training␣strategies");

        for (Strategy strategy : trainingStrategies) {
            training.addStrategy(strategy);
        }

        System.out.println(" Start␣training␣to␣accuracy:␣" + accuracy);
        int layers = network.getLayerCount();
        System.out.println("#Layer:␣" + layers);
        for (int i = 0; i < layers; i++) {
            System.out.println("Layer␣" + i + "␣#neurons:␣" + network
                .getLayerTotalNeuronCount(i));
        }

        long startTimeLong = System.nanoTime();
        EncogUtility.trainToError(training, accuracy);
        long endTimeLong = System.nanoTime();
        double durationInSec = (double) ((endTimeLong - startTimeLong) / Math
            .pow(10, 9));
        System.out.println(" Finished␣training␣network␣in:␣" + durationInSec);
    }

    @Override
    public double[] evaluate(double[] input, int maxIterations) {
        double[] output = new double[outputCount];
        System.out.println(" Evaluating␣input");
        long startTimeLong = System.nanoTime();
        network.compute(input, output);
        long endTimeLong = System.nanoTime();
        double durationInSec = (double) ((endTimeLong - startTimeLong) / Math
            .pow(10, 9));
        System.out.println(" Finished␣evaluating␣in:␣" + durationInSec);

        return output;
    }

    @Override
    public String getLayerLayout() {
        StringBuilder strBuilder = new StringBuilder();
        strBuilder.append("␣");
        int layers = network.getLayerCount();
        for (int i = 0; i < layers; i++) {
            strBuilder.append(network.getLayerTotalNeuronCount(i) - 1).append(
                "␣");
        }

        return strBuilder.append("]").toString();
    }
}

```

## A.7 EncogKohonenNetwork.java

**Listing A.7:** The self organizing map (Kohonen network).

## APPENDIX A. CODE

---

```
/*
 * The MIT License
 *
 * Copyright 2013 Pieter Van Eeckhout.
 *
 * Permission is hereby granted, free of charge, to any person obtaining a
 * copy
 * of this software and associated documentation files (the "Software"), to
 * deal
 * in the Software without restriction, including without limitation the
 * rights
 * to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
 * copies of the Software, and to permit persons to whom the Software is
 * furnished to do so, subject to the following conditions:
 *
 * The above copyright notice and this permission notice shall be included in
 * all copies or substantial portions of the Software.
 *
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
 * IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
 * FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
 * AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
 * LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
 * FROM,
 * OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN
 * THE SOFTWARE.
 */
package bachelorthesis.neuralnetworks.network.encog.kohonen;

import bachelorthesis.neuralnetworks.network.NeuralNetwork;
import static bachelorthesis.neuralnetworks.network.encog.kohonen.
    NeighborhoodFunctionType.BUBBLE;
import static bachelorthesis.neuralnetworks.network.encog.kohonen.
    NeighborhoodFunctionType.RBF;
import static bachelorthesis.neuralnetworks.network.encog.kohonen.
    NeighborhoodFunctionType.RBF1D;
import static bachelorthesis.neuralnetworks.network.encog.kohonen.
    NeighborhoodFunctionType.SINGLE;
import bachelorthesis.neuralnetworks.util.TrainingSet;
import java.util.HashMap;
import java.util.Map;
import org.encog.mathutil.rbf.RBFEnum;
import org.encog.ml.data.MLData;
import org.encog.ml.data.MLDataSet;
import org.encog.ml.data.basic.BasicMLData;
import org.encog.ml.data.basic.BasicMLDataSet;
import org.encog.neural.som.SOM;
import org.encog.neural.som.training.basic.BasicTrainSOM;
import org.encog.neural.som.training.basic.neighborhood.NeighborhoodBubble;
import org.encog.neural.som.training.basic.neighborhood.NeighborhoodFunction;
import org.encog.neural.som.training.basic.neighborhood.NeighborhoodRBF;
import org.encog.neural.som.training.basic.neighborhood.NeighborhoodRBF1D;
import org.encog.neural.som.training.basic.neighborhood.NeighborhoodSingle;

/**
 * EncogKohonenNetwork.java (UTF-8)
 *
 * Provides a configurable Encog SOM
 *
 * 2013/05/19
 *
 * @author Pieter Van Eeckhout <vaneeckhout.pieter@gmail.com>
 */
```

```

* @author Pieter Van Eeckhout <pieter.vaneeckhout.q1295@student.hogent.be>
* @author Hogent StudentID <2000901295>
* @since 1.0.0
* @version 1.0.0
*/
public class EncogKohonenNetwork extends NeuralNetwork {

    private Map<Integer, Double[]> output;
    private SOM network;
    private NeighborhoodFunctionType neighborhoodFunctionType;
    private RBFEnum radialBiasFunction;
    private int radius;
    private double learningRate;
    private int[] neighborhoodSize;
    private double error;
    private boolean forceWinner;

    protected EncogKohonenNetwork(NeighborhoodFunctionType
        neighborhoodFunction,
        RBFEnum radialBiasFunction, int radius, double learningRate,
        double error, boolean forceWinner, int id) {
        super(id);
        this.neighborhoodFunctionType = neighborhoodFunction;
        this.radialBiasFunction = radialBiasFunction;
        this.radius = radius;
        this.learningRate = learningRate;
        this.forceWinner = forceWinner;
        this.error = error;
    }

    @Override
    public String getLayerLayout() {
        return "[" + network.getInputCount() + "|" + network.getOutputCount()
            + "]";
    }

    @Override
    public void buildAndTrainNetwork(TrainingSet trainingSet) {
        network = new SOM(trainingSet.getInputCount(), trainingSet.
            getTrainingSetCount());
        network.reset();

        double[][] trainingInput = trainingSet.getInput();

        MLDataSet training = new BasicMLDataSet(trainingInput, null);

        NeighborhoodFunction neighborhoodFunction;

        switch (neighborhoodFunctionType) {
            case BUBLLE:
                neighborhoodFunction = new NeighborhoodBubble(radius);
                break;
            case RBF:
                neighborhoodFunction =
                    new NeighborhoodRBF(neighborhoodSize,
                        radialBiasFunction);
                break;
            case RBF1D:
                neighborhoodFunction = new NeighborhoodRBF1D(
                    radialBiasFunction);
                break;
            case SINGLE:

```

```
        neighborhoodFunction = new NeighborhoodSingle();
        break;
    default:
        neighborhoodFunction = new NeighborhoodSingle();
    }

    BasicTrainSOM train = new BasicTrainSOM(network, learningRate,
        training,
        neighborhoodFunction);

    train.setForceWinner(forceWinner);

    double err = 1;
    int iteration = 0;

    while (err > error) {
        train.iteration();
        err = train.getError();
        System.out.println("Iteration:\u0020" + iteration + ",\u0020Error:" + err +
            ",\u0020target\u0020error:\u0020" + error);
        iteration++;
    }

    //couple each output neuron to a trained pattern #output neurons == #
    trained patterns
    output = new HashMap<>();
    MLData data;
    for (int i = 0; i < trainingInput.length; i++) {
        double[] input = trainingInput[i];
        data = new BasicMLData(input);
        int outneuron = network.winner(data);

        //JAVA PRIMITIVE TYPES.... JUST WHY???
        Double[] outPattern = new Double[input.length];
        for (int j = 0; j < input.length; j++) {
            outPattern[j] = new Double(input[j]);
        }

        output.put(outneuron, outPattern);
    }
}

@Override
public double[] evaluate(double[] input, int maxIterations) {
    MLData data = new BasicMLData(input);
    Double[] outPattern = output.get(network.winner(data));
    double[] out = new double[outPattern.length];

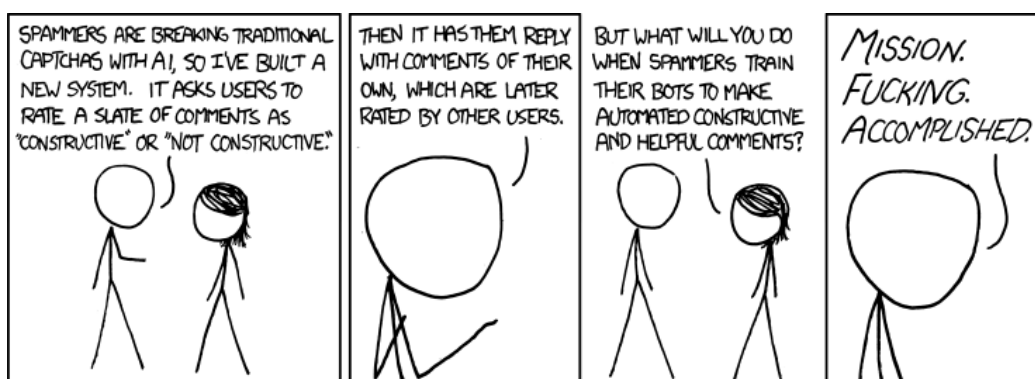
    //JAVA PRIMITIVE TYPES.... JUST WHY???
    for (int i = 0; i < outPattern.length; i++) {
        out[i] = outPattern[i];
    }

    return out;
}
}
```



## Appendix B

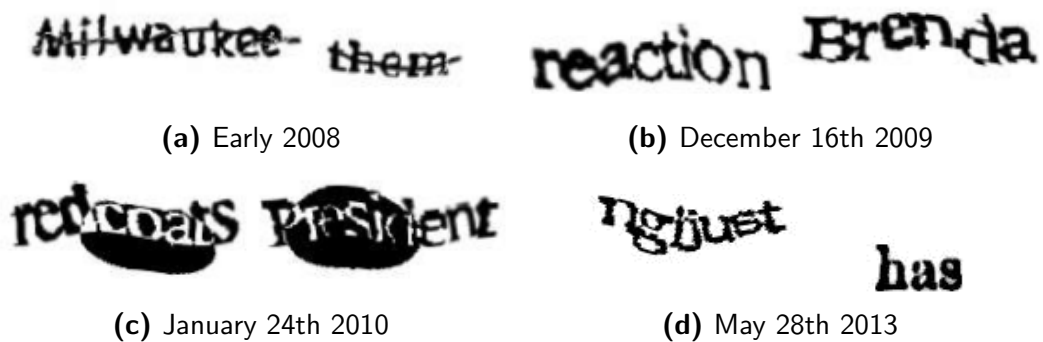
### Images



**Figure B.1:** xkcd on the future of CAPTCHA (Source: <http://www.xkcd.com/810/>, accessed on 2013/05/28)

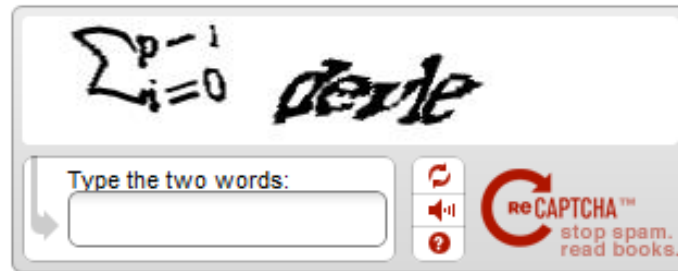


**Figure B.2:** The reCAPTCHA system (Source: [municipal cooperation.org, unknown])

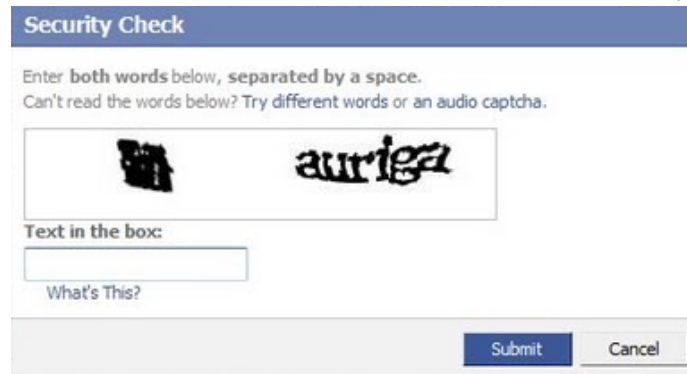


**Figure B.3:** Examples of CAPTCHAs directly Downloaded from reCAPTCHA (Source: [Motoyama et al., 2010] and [reCAPTCHA, 2013])

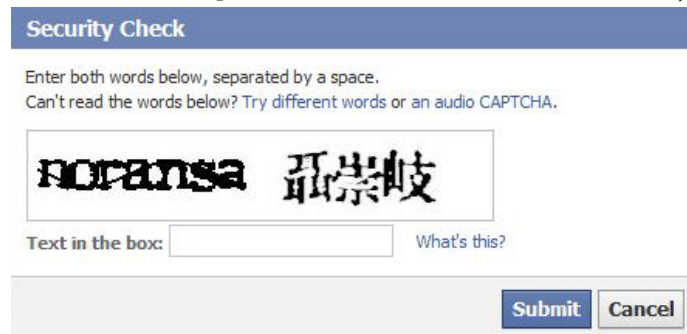




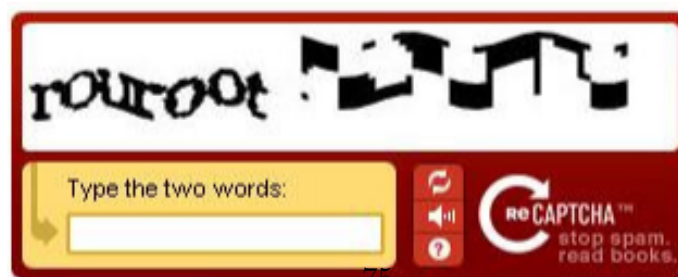
- (a) Impossible CAPTCHA (Source: <http://asmallpieceofgodspan.blogspot.be/2012/04/captchas.html> Accessed on 2013-0528)



- (b) Impossible CAPTCHA (Source: <http://asmallpieceofgodspan.blogspot.be/2012/04/captchas.html> Accessed on 2013-0528)

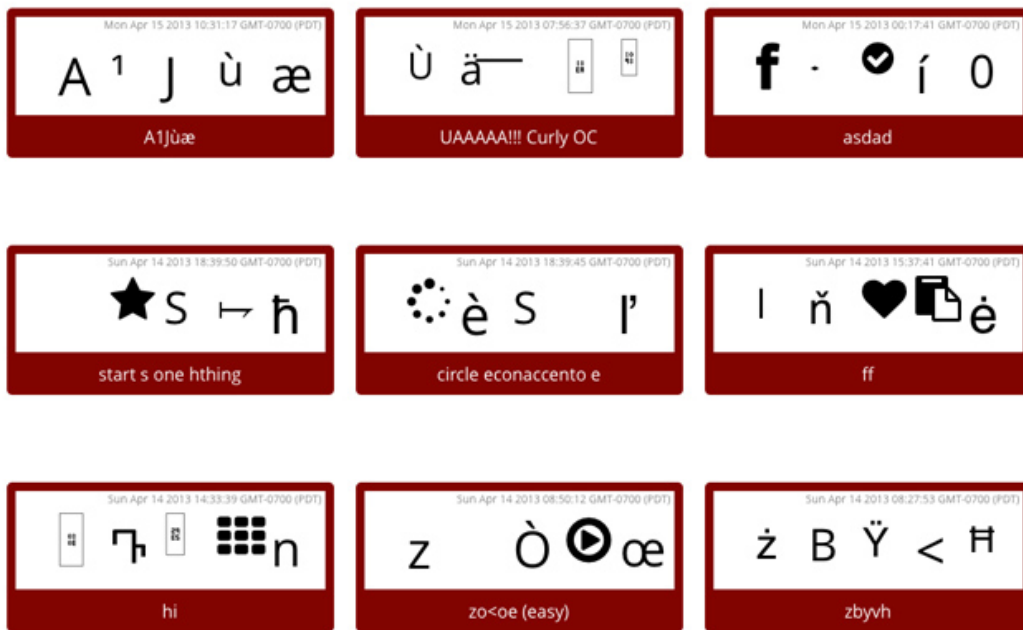


- (c) Impossible CAPTCHA (Source: <http://oactechnology.com/it-blog/blog/2012/08/31/captcha/> Accessed on 2013-0528)

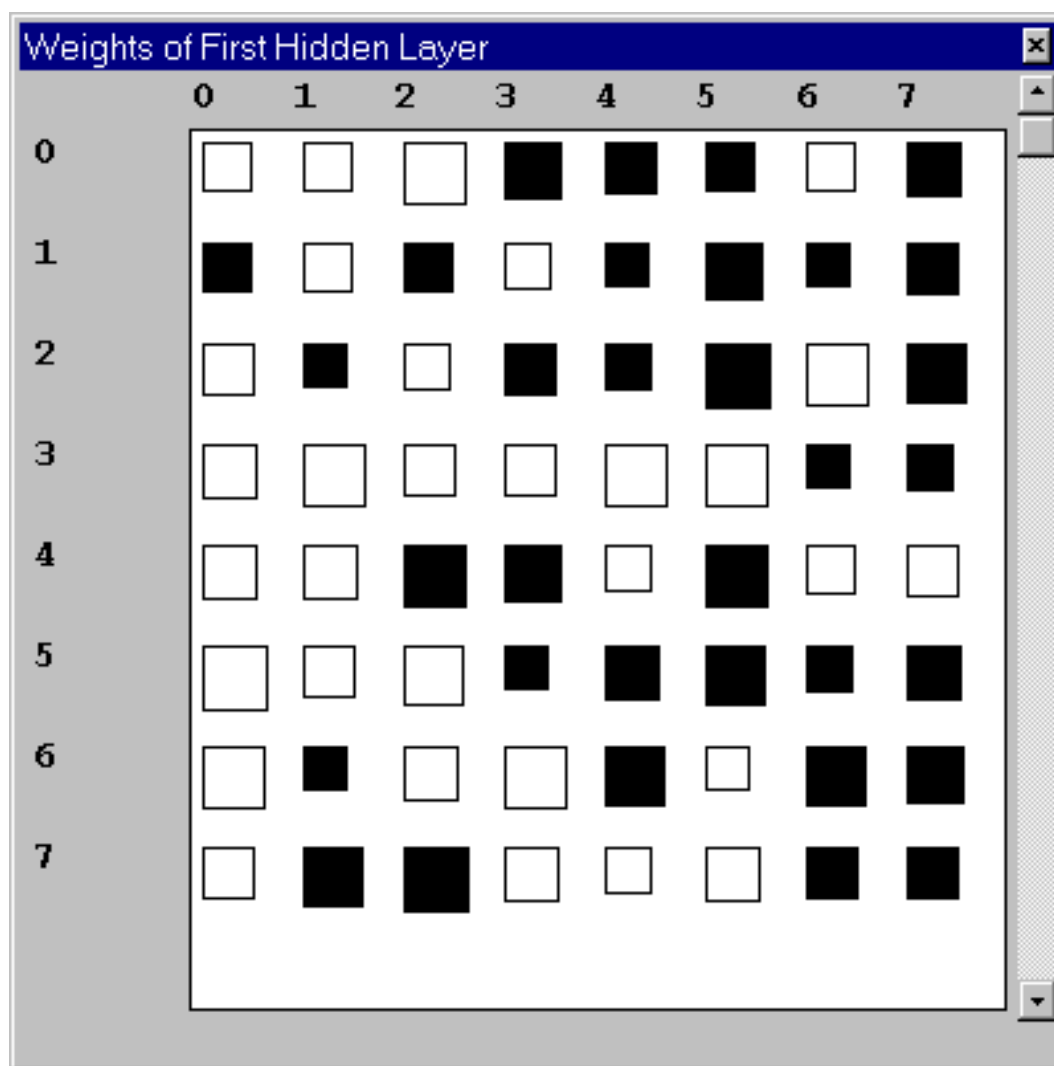


- (d) Impossible CAPTCHA (Source: <http://ragegenerator.com/pages/comic/39222> Accessed on 2013-0528)

Figure B.4: Examples of CAPTCHAs nearly impossible to solve



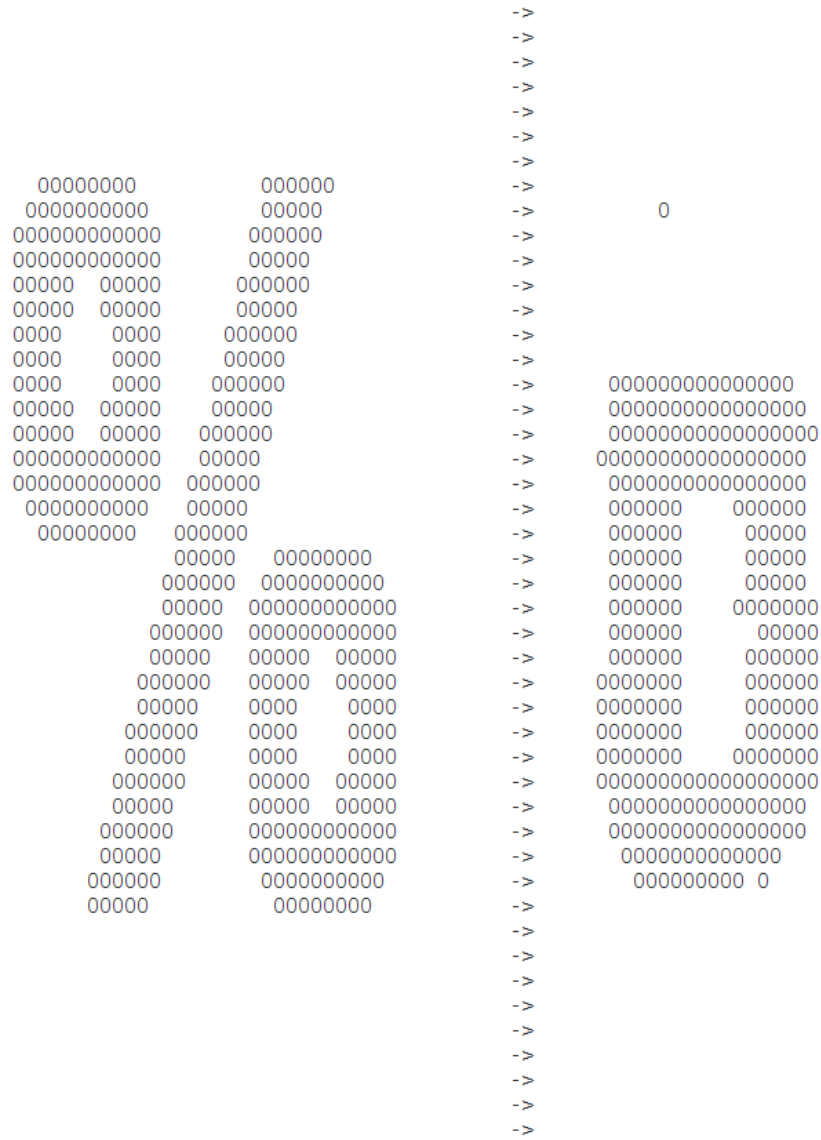
**Figure B.5:** Test generated by the CRAPCHA system (Source: [laughingsquid.com/])



**Figure B.6:** An example Hinton diagram (Source: <http://www.nd.com/products/nsv30/hinton.htm>)

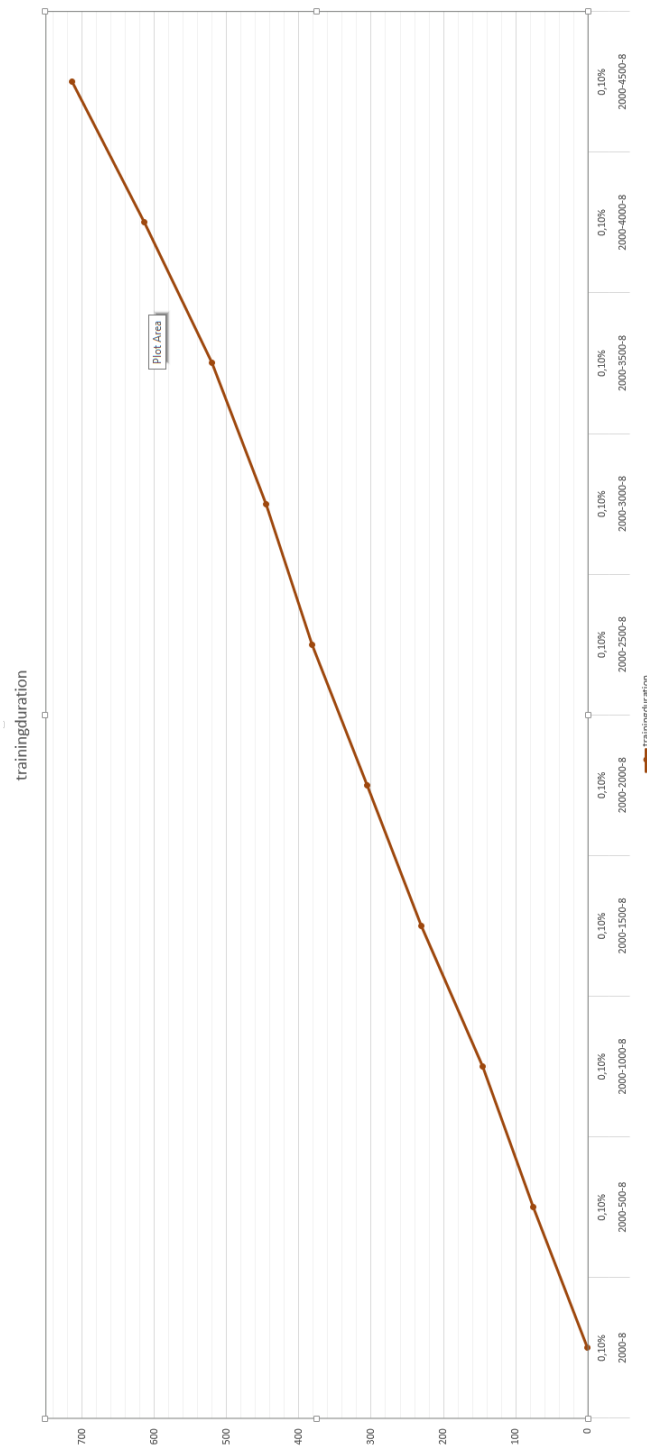
**a b c d e f g h i**  
**j k l m n o p q r**  
**s t u v w x y z 1**  
**2 3 4 5 6 7 8 9 0**  
**@ \$ = ! & # % + ?**

Figure B.7: The set of training images



**Figure B.8:** example of pattern overwriting when overtraining a Hopfield network





**Figure B.10:** Graphical representation of the increasing time requirement for training larger networks

# **Appendix C**

## **Tables**



**Table C.1:** Overview of the success rate for perceptron configurations

| Configuration | Accuracy | evaluation | evaluation % | normal | normal % | distorted | distorted % | trainingduration |
|---------------|----------|------------|--------------|--------|----------|-----------|-------------|------------------|
| 2000-8        | 10%      | 50         | 46,30%       | 69     | 13,80%   | 74        | 14,80%      | 0,22             |
|               | 1%       | 105        | 97,22%       | 93     | 18,60%   | 79        | 15,80%      | 0,34             |
|               | 0,10%    | 108        | 100,00%      | 32     | 6,40%    | 90        | 18,00%      | 0,47             |
| 2000-500-8    | 10%      | 48         | 44,44%       | 46     | 9,20%    | 28        | 5,60%       | 42,45            |
|               | 1%       | 102        | 94,44%       | 75     | 15,00%   | 43        | 8,60%       | 63,8             |
|               | 0,10%    | 108        | 100,00%      | 59     | 11,80%   | 54        | 10,80%      | 76,22            |
| 2000-1000-8   | 10%      | 32         | 29,63%       | 73     | 14,60%   | 80        | 16,00%      | 86,98            |
|               | 1%       | 103        | 95,37%       | 24     | 4,80%    | 36        | 7,20%       | 120,96           |
|               | 0,10%    | 108        | 100,00%      | 87     | 17,40%   | 50        | 10,00%      | 145,82           |
| 2000-1500-8   | 10%      | 45         | 41,67%       | 30     | 6,00%    | 27        | 5,40%       | 137,53           |
|               | 1%       | 105        | 97,22%       | 18     | 3,60%    | 36        | 7,20%       | 207,33           |
|               | 0,10%    | 108        | 100,00%      | 103    | 20,60%   | 35        | 7,00%       | 230,98           |
| 2000-2000-8   | 10%      | 49         | 45,37%       | 75     | 15,00%   | 11        | 2,20%       | 189,4            |
|               | 1%       | 104        | 96,30%       | 34     | 6,80%    | 46        | 9,20%       | 262,35           |
|               | 0,10%    | 108        | 100,00%      | 110    | 22,00%   | 27        | 5,40%       | 305,6            |
| 2000-2500-8   | 10%      | 41         | 37,96%       | 40     | 8,00%    | 29        | 5,80%       | 238,29           |
|               | 1%       | 100        | 92,59%       | 62     | 12,40%   | 41        | 8,20%       | 343,44           |
|               | 0,10%    | 108        | 100,00%      | 54     | 10,80%   | 43        | 8,60%       | 381,68           |
| 2000-3000-8   | 10%      | 43         | 39,81%       | 44     | 8,80%    | 22        | 4,40%       | 301,65           |
|               | 1%       | 105        | 97,22%       | 61     | 12,20%   | 50        | 10,00%      | 412,99           |
|               | 0,10%    | 108        | 100,00%      | 39     | 7,80%    | 17        | 3,40%       | 445,79           |
| 2000-3500-8   | 10%      | 38         | 35,19%       | 17     | 3,40%    | 40        | 8,00%       | 334,87           |
|               | 1%       | 103        | 95,37%       | 45     | 9,00%    | 54        | 10,80%      | 461,98           |
|               | 0,10%    | 108        | 100,00%      | 91     | 18,20%   | 24        | 4,80%       | 520,07           |
| 2000-4000-8   | 10%      | 30         | 27,78%       | 24     | 4,80%    | 34        | 6,80%       | 380,71           |
|               | 1%       | 103        | 95,37%       | 64     | 12,80%   | 39        | 7,80%       | 531,18           |
|               | 0,10%    | 108        | 100,00%      | 51     | 10,20%   | 11        | 2,20%       | 613,68           |
| 2000-4500-8   | 10%      | 58         | 53,70%       | 73     | 14,60%   | 0         | 0,00%       | 498,38           |
|               | 1%       | 102        | 94,44%       | 41     | 8,20%    | 7         | 1,40%       | 621,92           |
|               | 0,10%    | 106        | 98,15%       | 76     | 15,20%   | 29        | 5,80%       | 713,27           |



# Bibliography

- M Tariq Banday and NA Shah. A Study of CAPTCHAs for Securing Web Services. *IJSDIA International Journal of Secure Digital Information Age*, 1(2):66–74, 2011. URL <http://adsabs.harvard.edu/abs/2011arXiv1112.5605T>.
- Rodney Beede. *Analysis of reCAPTCHA effectiveness*. PhD thesis, University of Colorado at Boulder, 2010. URL <https://www.google.be/url?sa=t&rct=j&q=&esrc=s&source=web&cd=2&cad=rja&sqi=2&ved=0CD0QFjAB&url=http%3A%2F%2Fwww.rodneybeede.com%2Fdownloads%2FCSCI5722%2520-%2520Computer%2520Vision%2C%2520Final%2520Paper%2C%2520Rodney%2520Beede%2C%2520Fall%25202010.pdf&ei=po0GUebeNsan08LsgIAE&usg=AFQjCNHg9HPwDNUm7Qvv1tYXZ7YLhUToqg&sig2=-AYq-OR-Qnr6Qu6rXpVGNw&bvm=bv.45960087,d.d2k>.
- Adam Browning and Dave Kolas. Defeating CAPTCHAs : Applying Neural Networks. Technical report, 2007. URL <http://filebox.vt.edu/users/brownad/CS5804/abrowning-dkolas-project.pdf>.
- David Bushell. In Search Of The Perfect CAPTCHA, 2011. URL <http://coding.smashingmagazine.com/2011/03/04/in-search-of-the-perfect-captcha/>. accesed 2013-05-28.
- Tianhui Cai. CAPTCHA Solving With Neural Networks. Technical report, TJHSST Computer Systems Lab, 2008.
- Kumar Chellapilla and Kevin Larson. Computers beat humans at single character recognition in reading based human interaction proofs (HIPs). *Proceedings of the ...*, 98053, 2005. URL [http://www.researchgate.net/publication/220271810\\_Computers\\_beat\\_Humans\\_at\\_Single\\_Character\\_Recognition\\_in\\_Reading\\_based\\_Human\\_Interaction\\_Proofs\\_\(HIPs\)/file/79e4150d119fa42dff.pdf](http://www.researchgate.net/publication/220271810_Computers_beat_Humans_at_Single_Character_Recognition_in_Reading_based_Human_Interaction_Proofs_(HIPs)/file/79e4150d119fa42dff.pdf).
- Stephen Cobb. The Economics of Spam, 2003. URL [http://spamhelp.whybot.com/articles/economics\\_of\\_spam.pdf](http://spamhelp.whybot.com/articles/economics_of_spam.pdf).

- Dennis Egen. A Proposal For Improvements of Image Based CAPTCHA. Technical report, Rutgers University, Camden, 2009.
- M. Egmont-Petersen, D. de Ridder, and H. Handels. Image processing with neural networks—a review. *Pattern Recognition*, 35(10):2279–2301, October 2002. ISSN 00313203. doi: 10.1016/S0031-3203(01)00178-9. URL <http://linkinghub.elsevier.com/retrieve/pii/S0031320301001789>.
- Wei Gang and Yu Zheyuan. Storage Capacity of Letter Recognition in Hopfield Networks. Technical report. URL FacultyofComputerScience, DalhousieUniversity, Halifax, N.S., Canada B3H1W5.
- Jeff Heaton. *Introduction to neural networks with Java*. 1 edition, 2005. ISBN 097732060X. URL <http://tocs.ulb.tu-darmstadt.de/185177875.pdf>.
- D O Hebb. *The Organization of Behavior: A Neuropsychological Theory*, volume 44 of *A Wiley book in clinical psychology*. Wiley, 1949. ISBN 0805843000. doi: 10.2307/1418888. URL <http://www.amazon.com/dp/0805843000>.
- J. J. Hopfield. Neural Networks and Physical Systems with Emergent Collective Computational Abilities. *Proceedings of the National Academy of Sciences*, 79(8):2554–2558, April 1982. ISSN 0027-8424. doi: 10.1073/pnas.79.8.2554. URL <http://www.pnas.org/cgi/content/long/79/8/2554>.
- SY Huang, YK Lee, Graeme Bell, and Zhan-he Ou. *An efficient segmentation algorithm for CAPTCHAs with line cluttering and character warping*. PhD thesis, Ming Chuan University, 2010. URL <http://link.springer.com/article/10.1007/s11042-009-0341-5>.
- Xudong Jiang and Alvin Harvey Kam Sie Wah. Constructing and training feed-forward neural networks for pattern classification. *Pattern Recognition*, 36:853–867, 2003. URL <http://www.sciencedirect.com/science/article/pii/S0031320302000870>.
- Dennis W K Khong. An Economic Analysis of Spam Law. *Erasmus Law and Economics Review*, 1(February):23–45, 2004. URL <http://www.eier.org/viewarticle.php?id=2>.
- Teuvo Kohonen. Self-organized formation of topologically correct feature maps. *Biological Cybernetics*, 43(1):59–69, 1982. ISSN 0340-1200. doi: 10.1007/BF00337288. URL <http://dx.doi.org/10.1007/BF00337288>.
- David Kriesel. *Neural Networks*. 2013 edition, 2013. URL [http://www.dkriesel.com/en/science/neural\\_networks](http://www.dkriesel.com/en/science/neural_networks).

laughingsquid.com/. Crapcha, impossible captcha tests that pranks web users. <http://laughingsquid.com/crapcha-impossible-captcha-tests-that-prank-web-users/>.

Marti Motoyama, Kirill Levchenko, Chris Kanich, Damon Mccoy, Geoffrey M Voelker, and Stefan Savage. *Understanding CAPTCHA-Solving Services in an Economic Context*. PhD thesis, University of California, San Diego, 2010. URL [https://www.google.be/url?sa=t&rct=j&q=&esrc=s&source=web&cd=1&cad=rja&sqi=2&ved=0CEEQFjAA&url=http%3A%2F%2Fwww.usenix.org%2Fevent%2Fsec10%2Ftech%2Ffull\\_papers%2FMotoyama.pdf&ei=8I6GUa-YA6ar0QWAvoDoDg&usg=AFQjCNGw19XqLpU9H07GzrwXiZzp7AUSHw&sig2=n\\_TgEyAkII33doIB1VJFDw&bvm=bv.45960087,d.d2k](https://www.google.be/url?sa=t&rct=j&q=&esrc=s&source=web&cd=1&cad=rja&sqi=2&ved=0CEEQFjAA&url=http%3A%2F%2Fwww.usenix.org%2Fevent%2Fsec10%2Ftech%2Ffull_papers%2FMotoyama.pdf&ei=8I6GUa-YA6ar0QWAvoDoDg&usg=AFQjCNGw19XqLpU9H07GzrwXiZzp7AUSHw&sig2=n_TgEyAkII33doIB1VJFDw&bvm=bv.45960087,d.d2k).

municipal cooperation.org. Help:adding content. [http://www.municipal-cooperation.org/index.php?title=Help:Adding\\_content](http://www.municipal-cooperation.org/index.php?title=Help:Adding_content), unknown. Accessed: 2013-05-28.

Kazushi Murakoshi and Yuichi Sato. Reducing topological defects in self-organizing maps using multiple scale neighborhood functions. *Bio Systems*, 90(1):101–4, 2006. ISSN 0303-2647. doi: 10.1016/j.biosystems.2006.07.004. URL <http://www.ncbi.nlm.nih.gov/pubmed/16962231>.

Moni Naor. Verification of a human in the loop or Identification via the Turing Test. .... *wisdom. weizmann. ac. il/~ naor/PAPERS/human ...*, 1996. URL <http://www.wisdom.weizmann.ac.il/~naor/PAPERS/human.pdf>.

Lutz Prechelt. Proben1: A set of neural network benchmark problems and benchmarking rules. *Technicak Report*, 21(19/94):94, 1994. URL <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.41.3666&rep=rep1&type=pdf>.

Google reCAPTCHA. recaptcha. <http://www.google.com/recaptcha>, 2013. Accessed: 2013-05-28.

Frank Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6):386–408, 1958.

Graig Sauer and Harry Hochheiser. Towards a universally usable CAPTCHA. ... *the 4th Symp. On Usable ...*, pages 2–5, 2008. URL [http://www.researchgate.net/publication/228957237\\_Towards\\_a\\_universally\\_usable\\_CAPTCHA/file/d912f50d10c2235ccc.pdf](http://www.researchgate.net/publication/228957237_Towards_a_universally_usable_CAPTCHA/file/d912f50d10c2235ccc.pdf).

Markus Törmä. *Kohonen self-organizing featuremap in pattern recognition*. PhD thesis, Helsinki University of Technology, 1995.

Yasuharu Ukai and Toshihiko Takemura. Spam mails impede economic growth. *The Review of Socionetwork Strategies*, 1(1):14–22, March 2007. ISSN 1867-3236. doi: 10.1007/BF02981628. URL <http://link.springer.com/10.1007/BF02981628>.

Jeff Yan and Ahmad Salah El Ahmad. A low-cost attack on a Microsoft captcha. *Proceedings of the 15th ACM conference on Computer and communications security - CCS '08*, page 543, 2008. doi: 10.1145/1455770.1455839. URL <http://portal.acm.org/citation.cfm?doid=1455770.1455839>.

Roger Stephen Young. *How Computers Work Processor and Main Memory*. 2001. URL <http://www.fastchip.net/howcomputerswork/bookbpdf.pdf>.

# List of Figures

|      |  |    |
|------|--|----|
| 4.1  | Neuron Data processing (Source: [Kriesel, 2013] page 39, Figure 3.1)   | 18 |
| 4.2  | Standard feedforward network; Source:[Kriesel, 2013]   | 24 |
| 4.3  | Shortcut feedforward network; Source:[Kriesel, 2013]   | 25 |
| 4.4  | Direct recurrent network   | 25 |
| 4.5  | Indirect recurrent network   | 26 |
| 4.6  | Lateral recurrent network  | 27 |
| 4.7  | Completely linked network  | 27 |
| B.1  | xkcd on the future of CAPTCHA (Source: <a href="http://www.xkcd.com/810/">http://www.xkcd.com/810/</a> , accessed on 2013/05/28)           | 73 |
| B.2  | The reCAPTCHA system (Source: [municipal cooperation.org, unknown])  | 74 |
| B.3  | Examples of CAPTCHAs directly Downloaded from reCAPTCHA (Source: [Motoyama et al., 2010] and [reCAPTCHA, 2013])                            | 74 |
| B.4  | Examples of CAPTCHAs nearly impossible to solve  | 75 |
| B.5  | Test generated by the CRAPCHA system (Source: [laughingsquid.com/])  | 76 |
| B.6  | An example Hinton diagram (Source: <a href="http://www.nd.com/products/nsv30/hinton.htm">http://www.nd.com/products/nsv30/hinton.htm</a> ) | 77 |
| B.7  | The set of training images   | 78 |
| B.8  | example of pattern overwriting when overtraining a Hopfield network  | 79 |
| B.9  | Graphical representation of the success rate for perceptron configurations   | 80 |
| B.10 | Graphical representation of the increasing time requirement for training larger networks   | 81 |

# List of Tables

|     |  |    |
|-----|--|----|
| C.1 | Overview of the success rate for perceptron configurations . . . . . | 83 |
|-----|--|----|



# Listings

|     |  |    |
|-----|--|----|
| A.1 | The ColorRangeContainerClass, used for storing the colors used in the elements and randomly getting one of the available colors. . . . | 51 |
| A.2 | The Captcha test. . . . .  | 55 |
| A.3 | Utility class to convert images to array representations. . . . .  | 57 |
| A.4 | Utility class to the training patterns. . . . .  | 60 |
| A.5 | The Hopfield network. . . . .  | 63 |
| A.6 | The multilayered perceptron network. . . . .   | 65 |
| A.7 | The self organizing map (Kohonen network). . . . .   | 68 |



# **Appendix D**

## **CD**