MASTER'S THESIS

Spiking Neural Networks for Pattern Recognition

Ketil Vestbøstad

supervised by
Terje Kristensen

June 13, 2017

Acknowledgements

First of all, I would like to thank my supervisor Terje Kristensen and also my co-supervisor Lars Michael Kristensen. I have profited much from their competence, they have been available for my needs, and they have shown a lot of interest in my work and progress. Terje Kristensen has also allowed me to use part of his earlier works as inspiration and base class for some of my development. Furthermore, I would like to thank the mathematicians Jon Birger Drange and Jon Eivind Vatne for their contributions.

I would also like to thank my family and especially my wife Kari for her support through a demanding period of time.

Last, but not least, as a Christian I want to thank God for his presence and provisions.

Contents

1 Introduction

Spiking neural networks is no longer a new notion, as it has been around since the later 1990-ties. Still the development has not been as fast as the pioneers probably envisioned, and the actual usage today is not as wide as they may have believed at that time. Nevertheless, there has been important breakthroughs in the later years, especially when it comes to hardware-implementation of spiking neurons. A series of questions have triggered my own interest for the field of spiking neural networks: How can computers become better at accomplishing tasks and problems that the human brain solves all the time? What can we learn from how how biological neurons interact, and how can these insights help us find new models for programming? And the other way around, how can computer simulation of spiking neurons help us better understand the biological processes in the brain?

We find it natural to start by giving an outline of the biological neuron, as its structure and way of functioning to a great extent is the model for the artificial neural networks that are the theme of this thesis.

1.1 The Biological Neuron

The notion "neural network" reveals of course the main inspiration of its origin, namely the neural system of the brain. The brain of humans and of animals is a computing unit that from sensory inputs can produce a wide variety of reactions, thoughts and emotions (to say the least). In humans the brain consists of close to 10^{11} neuron cells, which are the actual processing units of the brain. (It should be noted that brain tissue consists also of lots of other cells, called glia cells. But as they are not involved in information exchange, they will not be treated in this thesis.) Since the function of the biological neuron, as well as the terminology related to it, is to a great extent used also in artificial neural networks, we find it worthwhile describing them closer.

1.1.1 The Structure of the Neuron

Even though there are different kinds of neurons with their own properties, they all have the same basic structure. Three functionally distinct parts of the neuron can be found: The soma (or cell body), the dendrites and the axon. (See Figure 1) The dendrites could be called an input device, because

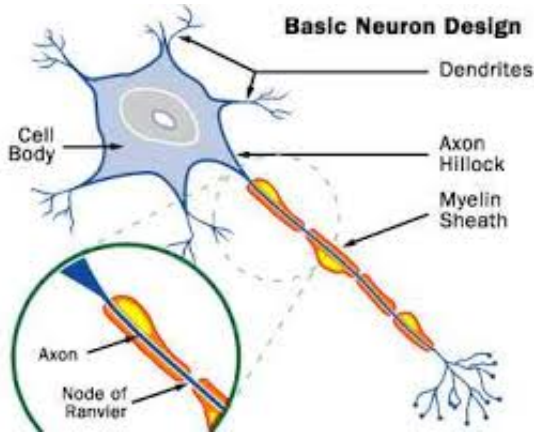

Figure 1: The neuron of the human brain. (Illustration from [1])

they receive signals from other neurons, which they in turn transmit to the soma. The soma then performs the central non-linear processing: Whenever enough input signals are received, it generates an electrical pulse which serves as the output signal. This pulse is then transmitted through the axon to other neurons. (There will be more on these dynamics in section 1.1.3.)

1.1.2 Synapses

The junction connecting an axon and a dendrite is called a synapse. Most synapses are chemical, in that an electrical signal from the sending neuron leads to a release of certain molecules called neurotransmitters, which in turn are caught by receptors at the receiving side of the synaptic cleft. (See Figure 2.) They lead to an ion influx which again changes the electric potential of the membrane of the receiving neuron cell. Other synapses are known to be electrical, in which specialized membrane proteins make a direct electrical connection between the two neurons.

Synapses can also be divided in two groups by the effect they have on the neuron that is receiving their signal. If the neuron's potential is heightened, then we have an excitatory synapse, and we get an excitatory post-synaptic potential. If it is lowered, we have an inhibitory synapse, and we get an inhibitory post-synaptic potential.

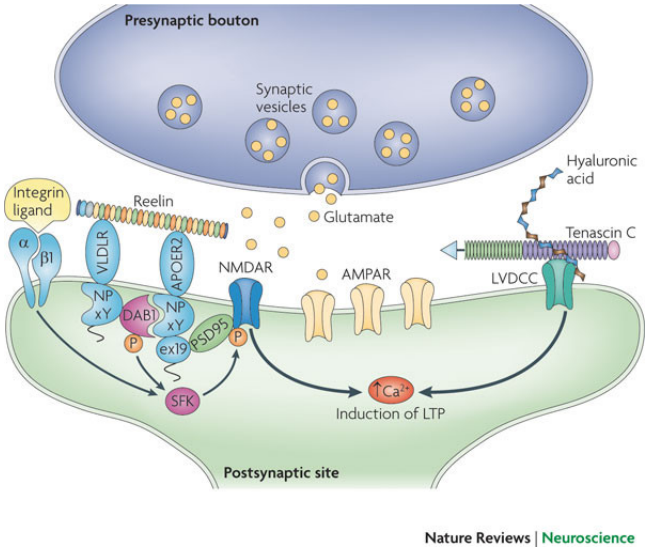

Figure 2: Close-up of the synaptic junction. (Illustration from [2])

1.1.3 The Neuron and its Action Potential

The potential difference between the interior of the cell and its surroundings is called the membrane potential. Without any activity, that is to say that no signals are received, this membrane potential will have a constant negative value of about $-65mV$. After the arrival of an electric pulse, the potential changes. If the pulse has passed through an excitatory synapse, the potential change is positive. In the opposite case, the change will be negative. A negative potential change will after some time decay back to the resting potential. With a positive potential change (following a signal from an excitatory synapse) there are two possibilities. If no or only a few more pulses are received during a short time span, the potential that has been built up will also in this case eventually decay back to the resting potential. But if enough excitatory signals arrive within this short time, the membrane potential will reach a critical value known as the firing threshold. Then the membrane potential exhibits a pulse-like excursion with an amplitude of about 100 mV and duration of 1-2 milliseconds, as illustrated in Figure 3. This is called an *action potential* or simply a *spike*, and in this thesis we will hereafter mostly use the term *spike* about this phenomenon. The actual process of cell membrane charge going from negative to positive is also known as *depolarization*.

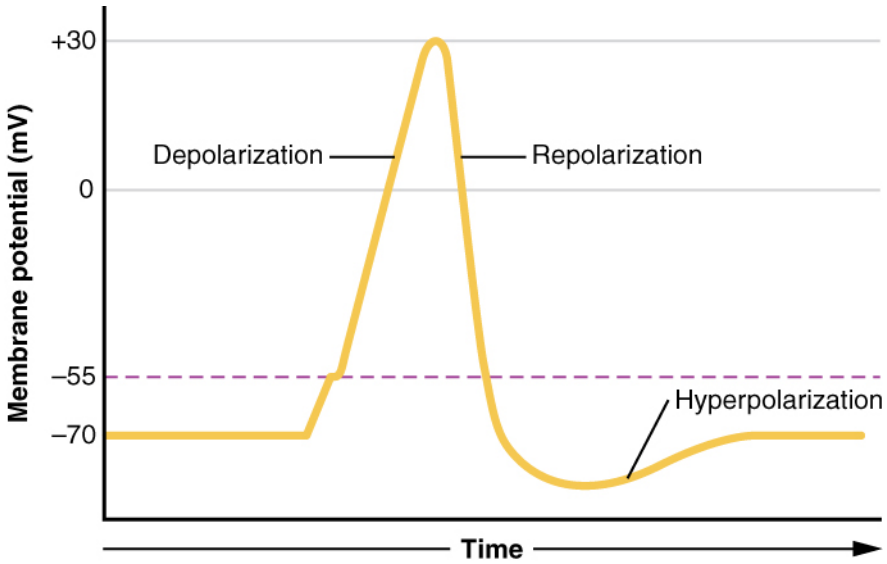

Figure 3: Action potential. (Illustration from [3])

This action potential will propagate along the axon of the neuron to the synapses of other neurons. After the action potential, the voltage goes through a short period of hyper-polarization, in which the voltage drops below resting potential. This is called the *refractory* period of the neuron, and it marks the minimal temporal distance between two spikes. This can be further divided into a period of total refractoriness, followed by a relative refractory time window. In the first of these it is impossible to excite new spikes, even with strong inputs. In the second one it is hard, but still not impossible to excite a spike. Throughout this thesis (as is done in [4]), the moment when a given neuron emits an action potential is called the spike time (or firing time) of this neuron.

1.1.4 Synaptic Plasticity and Hebbian Learning

It has been observed that even though an action potential is a quite uniform unit (as described above), one incoming spike can have a much larger effect on the change of potential of a post-synaptic neuron than another. It was found that it is the synapses that is the cause of this difference. Each synapse between pre-synaptic neuron i and post-synaptic neuron j can be attributed to a synaptic weight w_{ij}. It was also early suspected that this synaptic

weight was not a static unit, but rather dynamic. Already in 1949 Donald Hebb formulated his famous postulate [7]:

> "When an axon of cell A is near enough to excite cell B or repeatedly or persistently takes part in firing it, some growth process or metabolic change takes place in one or both cells such that As efficiency, as one of the cells firing B, is increased."

It is interesting to note that at the time, Hebb had only theoretical assumptions as basis for his postulate. But later on, experiments have confirmed it. Today, this principle is often rephrased in the meaning that changes in the synaptic transmission activity are influenced by correlations in the spiking activity of the pre- and post-synaptic neurons [4, p. 361]. It is found that certain stimulations can systematically induce modifications in the post-synaptic response that last for longer periods, such as hours or days. If the change in synaptic efficacy is positive, then we call it long-term potentiation of synapses. If the change is negative, we call it long-term depression. It is a common view that these persistent modifications are the neuronal correlate of *learning* and *memory*. The term *synaptic plasticity* (or more precisely *Spike-Timing Dependent Plasticity*) is often used for this neuronal behaviour.

In the formal theory of neural networks, the synaptic weight is considered as a parameter that can be modified in order to optimize the performance of an (artificial) neural network for a certain task. This process is called *learning* (in its widest sense), and the procedure for updating the weights is termed a *learning rule*. Since this process is inspired by Hebb's principle, it is often called *Hebbian learning*.

1.1.5 Spike Coding

As we have seen, quite a lot is known about individual neurons, synapses and how spikes are transmitted. However, we have far less knowledge when it comes to how these spikes should be interpreted. What is the information contained in this system of electrical pulses, and how can other neurons decode the signals? These questions concern the problem of neuronal coding, which is a fundamental issue in neuroscience. It has been the subject of lots of research for decades, yet it remains to a great extent unsolved. We cannot really treat this matter in any depth here, and so we just point to the main theories.

The traditional view has been that most (or even all) the information is contained in the neuron's mean firing rate. Already in 1926 it was shown by Edgar Adrian [6] that the firing rate of stretch receptor neurons in a muscle has some relationship to the measure of force applied to the muscle. This is often referred to as *rate coding*.

Still, more recent research has questioned whether this theory based on temporal average is too simplistic. It has been shown by behavioural experiments that reaction times are often too short to allow for useful temporal averages [8]. There is evidence that the precise timing of spikes can also contain important information. Naturally, this opens for several new questions and theories: Is it the first spike after a given stimulus that contains the main information, as proposed in the *time-to-first-spike* theory [4, p. 29]? Or is it rather global oscillations in the brain that form the reference point against which spikes should be measured? Could there be found a mechanism based on synchrony and correlations? A considerable amount of current research is dealing with these and other questions. A summary of this can be found in [4, p. 37].

1.2 Research Question

When it comes to formulating the research question, it may sometimes need to be negotiated between what would be the most interesting challenge from a scientific point of view on the one hand, and what is still realistic to achieve on the other hand. With these considerations in mind, our research question will be the following:

> We want to build a spiking neural network (SNN) from scratch. After validating this SNN, we will compare its performance to a more conventional kind of neural network. For this study we will use the problem of automated hyphenation of words of the Norwegian language. The two kinds of neural networks will be compared both on efficiency and accuracy on this problem.

1.3 Thesis Outline

The following is a brief summary of the content of the chapters that constitute the rest of this thesis.

Chapter 2: Artificial Neural Networks The section first gives an overview of the historical development in this field of research. Then we describe some of the motivation driving the development, before introducing a certain theoretical neuron model, The Spike Response Model. Following this comes a look at architectures for artificial neural networks, and then we explore the different possibilities of how information can be coded into spikes.

Chapter 3: Learning Algorithms for Spiking Neural Networks This chapter shows a theoretical derivation of a learning rule for a spiking neural network, first for one layer and then for multiple layers of neurons.

Chapter 4: Software Implementation This is a description of the structure of the software we have created.

Chapter 5: Validation In this chapter we perform simple tests on the spiking neural network that we have developed. For this purpose we use boolean functions and randomly generated spike trains, and they are tested on networks of different depths.

Chapter 6: The Hyphenation Problem This section consists of experiments where the performance of a spiking neural network is compared to a conventional neural network based on the backpropagation algorithm.

Chapter 7: Discussion, Conclusion and Future Directions In this chapter we reflect on the results that we have found, and try to explain some of the difficulties that we have encountered. We also give an outlook at an IBM research project based on spiking neural networks. We conclude the thesis, and we take a look at what could be natural follow-up challenges for the future.

2 Artificial Neural Networks

In this section we introduce the basic concepts of artificial neural networks (ANNs) and also give a brief account of the historical development. In addition we describe their architecture, functional features and some fields of actual use. We also point out the differences between network based on sigmoid neurons and networks of spiking neurons.

2.1 Historical Development

The idea to take inspiration from biological processes of the brain to create a computational model appeared quite early in the history of computer science. The following presentation of the historical development is largely based on [10], where Wolfgang Maass distinguishes three generations within the development of artificial neural networks. Common for all ANNs is that the neurons are the computational units, and it is the type of neuron that constitutes the basis for this kind of classification into generations.

2.1.1 First Generation

The first generation of ANNs was based on McCulloch-Pitts neurons [29], and are also referred to as perceptrons. The neural network models that have been developed from this base include multilayer perceptrons (MLP), Hopfield nets [18] and Boltzmann machines. A common characteristic of these models is that they can only have digital output. They are still universal for computations with digital input and output, and every boolean function can be computed by some two-layer perceptron.

2.1.2 Second Generation

The second generation of neural networks is based on units that make use of an activation function (also called a transfer function). This function takes a weighted sum of inputs, and generates a continuous set of possible output values. By far the most popular activation function is the sigmoid function:

$$\frac{1}{1+e^{-x}} \tag{2.1}$$

(A graphic illustration is provided in Figure 4.)

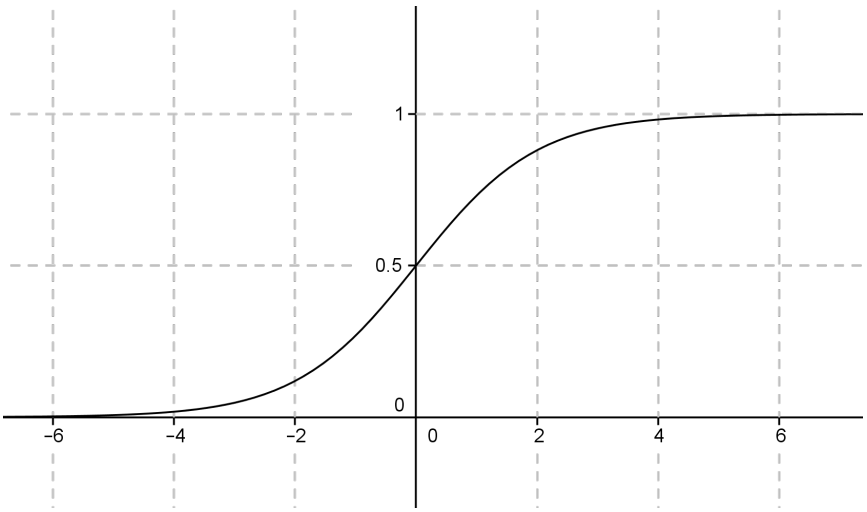

Figure 4: The sigmoid function

This function is found to be mathematically convenient since it is differentiable, and the derivative of the function can be easily described in terms of the original function:

$$\frac{d}{dx}f(x) = \frac{e^x}{(1+e^x)^2} = f(x)(1-f(x))$$

Other activation functions have also been used, an overview can be found at [11]. In contrast to the first generation, this kind of neural networks can handle analogue inputs and also give analogue output. They also support learning algorithms that are based on gradient descent [1], out of which *backpropagation* [19] probably is the best known. An important milestone was reached when a network finally was able to learn functions that are not linearly separable. In Euclidean geometry, linear separability is a property that can be found (or alternatively, that is lacking) in a pair of sets of data. The general definition states that the property is fulfilled if the sets can be separated by a straight line (in a 2-dimensional plane, see Figure 5 for an illustration), by a plane (in a 3-dimensional space) or by a hyperplane (in a multidimensional space). The boolean OR-function is an example of a function that is linearly separable, while the "exclusive OR-function" (also called

[1] We give a short description of the gradient descent method in section 3.1

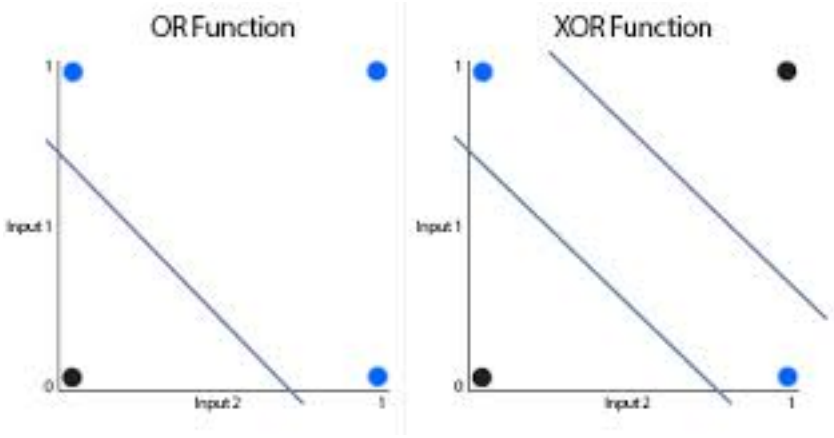

Figure 5: Linearly vs. not-linearly separable function. The blue points represent the boolean value *true* and the black points represent *false*.

"XOR") is usually mentioned as a basic non-linearly separable function, as illustrated in Figure 5.

These second generation networks are also biologically more realistic than the first generation, in that the output of a sigmoid (or corresponding function) unit can be interpreted as a representation of the current firing rate of a brain neuron.

2.1.3 Third Generation

Then, can this biological inspiration be taken any further? What results can we get if we mimic the brain neurons even closer? This may have been some of the questions that motivated the development of the third generation of neural networks. Here the computational units are spiking neurons (also called *integrate and fire-neurons*), thereby the common term *Spiking neural networks* (SNN). In this model, both the input and the output of a neuron is actually a series of chronologically ordered action potentials, which is usually called a spike train. Other aspects of a spike, like for instance the amplitude and the duration, are not taken into account - it is the time when an electric pulse is emitted that is defined to be relevant. Again, this seems to correspond quite closely to the biological model, in which the spikes are found to be quite uniform in character. Thus, as already stated, the neuro-scientists assume that the real information somehow lies in the spike times.

The state of a neuron is described by its *potential* , which is modelled by a dynamic variable, and the neuron works as a leaky integrator of the spikes it is receiving. Later spikes thus contribute more to the potential than earlier ones. Then, if the potential reaches a predefined threshold, the neuron fires a spike. The refractory period of a neuron is also modelled, so that there will be a minimum temporal distance between two following spikes. Even the synaptic delay is taken into account, although at this point (as we shall see shortly) our network model is deviating somewhat from its biological origin.

In contrast to the earlier generations, the spiking neural networks is called a dynamic system. Since the aspect of time (real or simulated) is so central, it may be better suited to do computations on temporal patterns. The term *temporal pattern* is not easily defined, but informally, we talk about phenomena of which a development along the time line is an important feature. We note that temporal data patterns often will be in the form of video or audio data.

Another interesting feature of this third generation of neural networks is found when we consider the two main theories on spike coding from section 1.1.5. As the earlier generations corresponded closely to the theory of rate coding, these neural networks will correspond to the idea that the precise timing of spikes contains important information.

2.2 Why Artificial Neural Networks?

The reasons for putting an effort into developing artificial neural networks can broadly be divided into two groups, each one described in the following two subsections.

2.2.1 Using Brain Architecture to Develop Computer Programs

Here the aim is to ameliorate the performance of computers in problem fields that have traditionally been challenging for programmers. Examples are pattern recognition, symbol interpretation and different kinds of classifications and clusterings. A number of approaches have been used:

Auto-association: The network is trained by a set of patterns. It will adjust its synaptic weights (see section 1.1.4) by means of the learning algorithm, and this way the training set will be stored in the memory of the network. When given a new pattern, the most similar training pattern will be reproduced.

Pattern association: Here the training set consists of pairs of patterns, and the network learns a mapping between the input and output patterns. When presented with an unknown input pattern, the network should pick an output pattern that corresponds to this mapping.

Clustering: No a priori classification is known when using this approach. The network should rather discover certain important features that enables it to classify the data.

Common real-life applications in which artificial neural networks often play a (more or less central) role, include the following:

- System identification and control: Self-driving vehicles, trajectory prediction, and natural resources management.
- Game-playing and decision making.
- Pattern recognition as used in radar systems, face identification, object recognition.
- Sequence recognition used for motion and speech recognition.
- Medical diagnosis.
- Financial applications: Automated trading systems, stock trading.
- Data mining: Discovering patterns in and extracting useful information from large data sets.

2.2.2 Programming Computers to Model Brain Activity

Above we have seen how the architecture of the brain has inspired digital applications. But an important motivation for using artificial neural networks is actually the other way around: They are used to simulate brain activity, and from these simulations our knowledge of biological neural systems can be expanded. Computational neuroscience has become an important research field in later years. Notorious is the Human Brain Project [20] which was initiated in October 2013, involving 113 institutions across Europe. Its aim is nothing short of creating a whole brain model within its 10 years funding period. This will in turn hopefully facilitate medical research related to healing and brain development.

2.3 The Spike Response Model

In this section, we will outline the formal neuron model that we will use in our further investigations. It is called the Spike Response Model (abbreviated SRM), and was first introduced by Gerstner [5]. This model describes how a single neuron processes incoming spikes in order to produce output-spikes. We should note that this is originally a purely biological model. As our aim is not to describe the biological processes as closely as possible, but rather to make an effective computational model, we find it appropriate to use a modified and somewhat simplified version of it.

In the Spike Response Model, the state of a neuron j is characterized by its potential $u_j(t)$, in which t represents the time line. Any neuron from which neuron j can receive spikes is called a *presynaptic* neuron with regard to j. Likewise, any neuron that neuron j may send spikes to, is called a *postsynaptic* neuron. Incoming spikes will either increase or reduce the potential, depending on whether they have been passing through an excitatory or inhibitory synapse. If a rising potential reaches a given threshold ϑ, the neuron fires a spike. As already mentioned, it is the time of the spike (hereafter called spike-time) that contains the interesting information. Therefore, the output of neuron j will be an array (possibly empty) of spike-times, which can be characterized as follows:

$$\mathcal{F}_j = \{t_j^{(f)}; 1 \leq f \leq n\} = \{t, | \, u_j(t) = \vartheta\} \tag{2.2}$$

in which n is the number of spikes emitted by the neuron. $\mathcal{F}_j$ is also called a spike train. In such a spike train, all the spikes will be chronologically ordered. Thus it will have the form of an array of strictly rising numbers. If f and g are indexes in this array, and $1 \leq f < g \leq n$, then $t_j^{(f)}$ is an earlier spike than $t_j^{(g)}$.

Now, if the presynaptic neuron i fires a spike at $t_i^g \in \mathcal{F}_j$, then the postsynaptic potential $u_j(t)$ will in this model be changed by $w_{ij}\epsilon(t - t_i^g - d_{ji})$. Here the weight of the connection between neurons i and$j$ is denoted with the variable w_{ji}, while d_{ji} denotes the delay. We see that the spike response function ϵ mathematically describes the effect of the incoming spike on the potential of the postsynaptic neuron. Different mathematical expressions have been used for this function, but the function always has a short rising part followed by a longer decaying period, as illustrated in Figure 6. We will use the following:

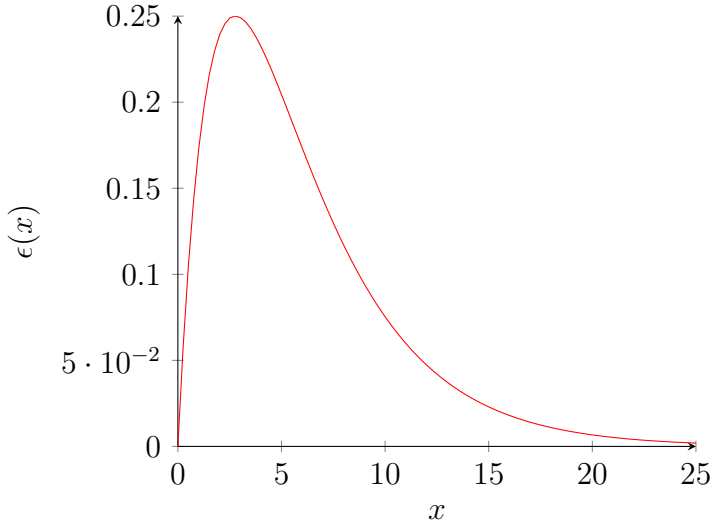

Figure 6: The ϵ kernel of the Spike Response function illustrated.

$$\epsilon(s) = \Big[\exp(-s/\tau_m) - \exp(-s/\tau_s)\Big]\mathcal{H}(s) \tag{2.3}$$

in which $\mathcal{H}(s)$ is the Heaviside step function: $\mathcal{H}(s) = 0$ for $s \leq 0$ and $\mathcal{H}(s) = 1$ for $s > 0$. There are two time-constants τ_m and τ_s (with $\tau_m > \tau_s > 0$) that determine how fast the function rises and decays, and they determine also its top point. It is hard to find theoretical justifications for these (and other) constants that is used in this model. We assume that they are mainly chosen experimentally, and inspired by [9] we set τ_m to 4.0 and τ_s to 2.0.

Refractoriness is also influencing the potential of a neuron. This term is modelled by a function η. As described in section 2.3, if a neuron j emits a spike at $t_j^{(f)}$, its potential at (the later) time t is reduced with $\eta(t - t_j^{(f)})$. This function is not given a fixed form in the Spike Response Model, but it is normal to use a non-positive simple exponential decay, and again we use [9] as our inspiration:

$$\eta(s) = -\vartheta \exp(-s/\tau_r)\mathcal{H}(s) \tag{2.4}$$

where ϑ is the threshold constant, $\mathcal{H}(s)$ is the Heaviside function (as above) and τ_r is a new time-constant that can be given different values. In most of our simulations we used $\tau_r = 20.0$. See Figure 7 for an illustration.

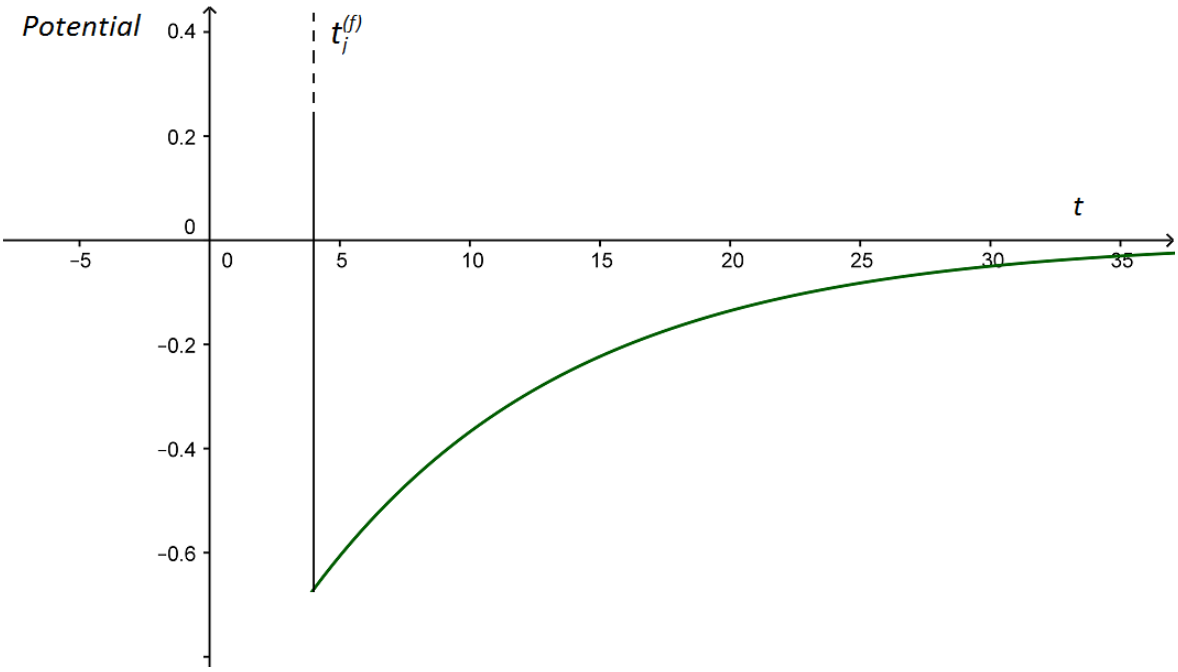

Figure 7: The function $\eta(t - t_j^{(f)})$ modelling refractoriness. When neuron j fires a spike, its potential drops immediately to $-\vartheta$, and then gradually goes back to resting potential.

The formula for the potential of a neuron j as a function of time, $u_j(t)$, is made up of the refractoriness term and the sum of the inputs from all its presynaptic neurons:

$$u_j(t) = \eta(t - t^f) + \sum_{i \in \Gamma_j} w_{ji}^k \sum_{t_i^{(g)} \in \mathcal{F}_i} \epsilon_{ij}(t - t_i^g - d_{ji}) \tag{2.5}$$

Here Γ_j is the set of all presynaptic neurons.

We note also that the refractoriness term in [9] is written as the sum over all the previous spikes of the neuron:

$$\sum_{t_j^{(g)} \in \mathcal{F}_j} \eta(t - t^f)$$

But Gerstner and Kistler [4, p. 122] say that "In realistic spike trains, the interval between two spikes is typically much longer than the time constant τ_m. Hence, the sum over the η terms are usually dominated by the most recent firing time $t_i^{(f)} > t$ of neuron i. We therefore truncate the sum over f and neglect the effect of earlier spikes." They conclude their argument stating that "Loosely speaking, the neuron remembers only its most recent firing." This supports our intuition that reducing the term to only include the most recent spike will have virtually no effect on the output of the function. Still it will be a significant simplification, first when it comes to our implementation, but of course also computationally. For this reason, we will stick to the formula 2.5.

As a conclusion, the equations 2.2 - 2.5 describe the behaviour of a single neuron within the Spike Response Model.

2.4 Network Architecture

So far we have concentrated on single neurons, but as already stated, we will need a network of neurons in order to perform computations. Then the question of which architecture to use needs also to be answered.

From the world of conventional artificial neural networks, we know that a variety of network architectures have been explored. The most basic architecture has only a set of input neurons and a set of output neurons. It is fully connected in the sense that an input neuron is connected to every output neuron, but not to any other input neuron. Likewise, every output

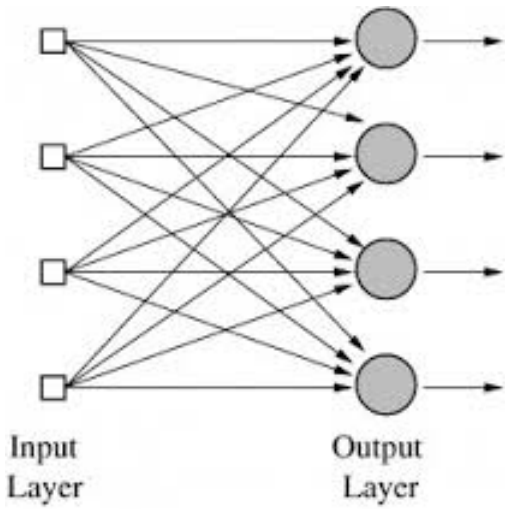

Figure 8: A 1-layered neural network

neuron is connected to every input neuron, but not to any other output neuron. We say that the input neurons form an *input layer*, and the output neurons form an *output layer*. It can be argued that the input layer has no real neurons, since there is no actual processing involved in them. They are rather forced to give a certain output.

This architecture is most often referred to as one-layered networks, as shown in Figure 8. What is actually counted is not the neuron layers, but rather the layer of adjustable weights between them. Examples of this architecture are the first generation's perceptron and adaline networks.

Then it is possible to put a set of neurons between the input and output layers. This set can be organized in one or more layers of neurons. These are usually called hidden layers, since they are not visible from the input and output layers at the outside. Figure 9 illustrates an ANN with hidden neuron layers. We should note that a network with one hidden layer most often will be referred to as a two-layered network. As already stated, it is then the layers of weights that are counted.

Furthermore, there are *feed-forward* and *recurrent* neural networks, of which in the latter connections among neurons can form directed cycles. In a feed-forward network, there are no cycles or loops, the information always goes in the same direction: From the input layer, possibly through hidden layer(s), and ending in the output layer. This is the most basic and well-

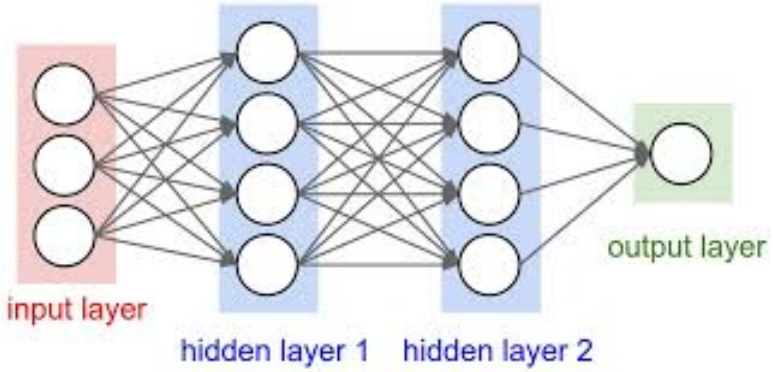

Figure 9: A 3-layered neural network. Note that what is counted is not the neuron layers, but the layers of connections (synaptic weights) between the neurons.

known out of the two. A recurrent network will usually involve a higher degree of complexity, and implementing them, as well as controlling their behaviour, seems to demanding at our level. Therefore we have chosen to use a feed-forward architecture as base for our spiking neural network. Most artificial neural networks are analogous to their biological counterparts in that any two connected neurons share only one single connection channel, and then only one synapse. With inspiration from [12] we are deviating somewhat from this model, in that we use multiple parallel sub-connections between any two connected neurons in our architecture. Each of these sub-connections has its own distinct synaptic weight and also a certain temporal delay. The delay is defined as the difference between the firing time of the pre-synaptic neuron, and the time the post-synaptic potential starts rising. This is illustrated in Figure 10. This architecture should help us simulate the time aspect in our implementation, and the number of parallel sub-connections is usually the same as the size of the *time window* (see section 4.2.3) we use.

2.5 Spike Coding in SNN

As we have already seen, both input and output to a s piking neural network are in the form of spike trains. But the data on which we want to do computations may be of different forms: Images, sounds, video recordings, and symbols of different kinds. This is data that by nature (although maybe not in their actual representation) is analogue. An important question then

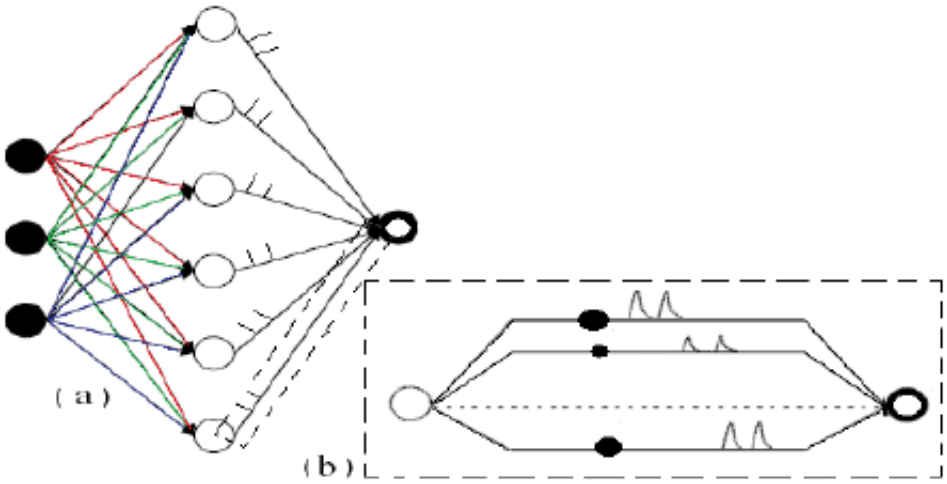

Figure 10: (a) A two-layer SNN, giving (b) a close-up of one of the connections between two neurons, showing how it is subdivided into multiple synaptic terminals, each of which has its own distinct weight and delay associated with it. (Image from [13])

is how this data can be transformed or encoded into spike trains. We may again turn to neurobiology to search for answers, but as we have observed (in section 1.1.5), even the few proposals that have been made are by no means definitive. Naturally, the data have to be preprocessed in some way. For images, we need light- and colour-values for each pixel. For sound waves, we need to break it down into smaller units, each with its value for strength (amplitude) and frequency. Motions may have to be divided into frames. Whenever this is done, we find that several different approaches are possible.

The simplest solution may be to make the input-neuron's firing time proportional to the (above mentioned) data unit value. This is often called "Time-to-first-spike coding", and it is a parallel to one of the two main theories of spike coding that we know from neurobiology (section 1.1.5. It does only allow one spike within each time window, which may of course be a limitation.

Another approach is to let all the data values go through a thresholding function. Then the resulting stream of bits can be interpreted as a spike train. The thresholding will of course lead to a large loss of data, since every value will be reduced to a bit. But there may also be advantages to it, as the method can give simple and compact spike trains, which may turn out

useful in algorithms and applications that allow for simplifications.

It is also possible to distribute data from one variable over several neurons. This is called population coding, with the idea that a value is represented by a population of neurons. Different ways of doing this have been shown. One example is found in [12], where eight separate neurons are used to encode one analogue value. Every neuron is represented by a gaussian kernel with a given mean and variance, and they also have a certain overlap with their neighbours. The height of the gaussian kernel as a function of the input value then determines the firing time of the neuron, see Figure 11. The biological inspiration for this kind of data encoding is taken form the theory of *receptive fields*. The receptive field of a sensory neuron is the particular region of the sensory space (for example the body surface, or the field of vision) in which a stimulus will affect the firing of that neuron.

The choice of method will often depend on the nature of the data to be encoded. If there are relatively few values, each with a high level of information density, then population coding may be well suited. On the other hand, when there's a high number of values, like in video samples, then thresholding may be a more natural choice.

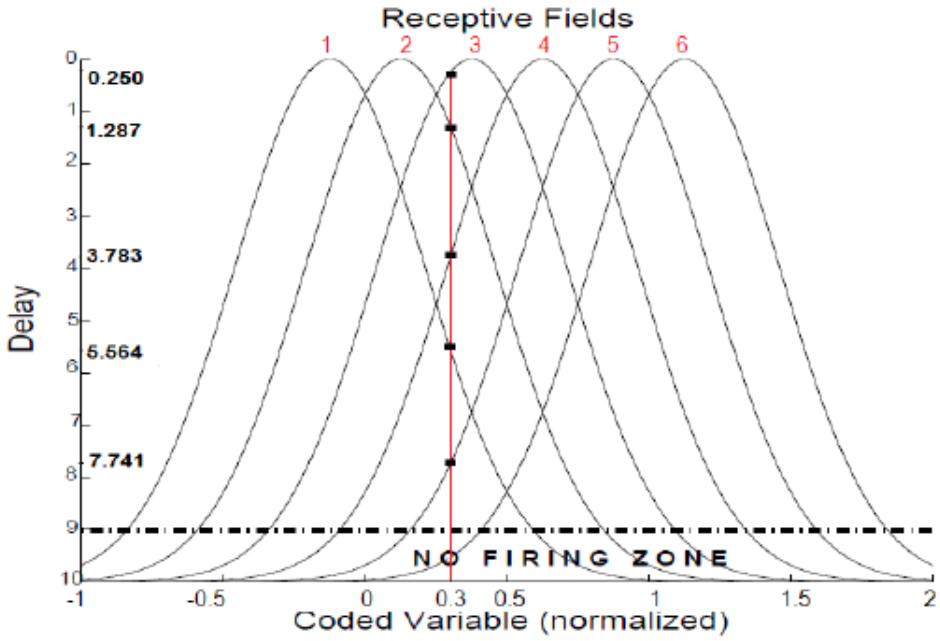

Figure 11: Population coding. An input value is encoded by means of (in this case) 6 gaussian activation functions. For the real-valued input 0.3 five of the neurons will fire: Neuron 1 fires at $t = 6$ ($5.564 \approx 6$), neuron 2 fires at $t = 1$ ($1.287 \approx 1$) neuron 3 fires at $t = 0$ ($0.250 \approx 0$) neuron 4 fires at $t = 4$ ($3.793 \approx 4$ neuron 5 fires at $t = 8$ ($7.741 \approx 8$), while neuron 6 does not fire at all, since it falls in the *no firing zone*. (Figure from [14].)

3 Learning Algorithms for Spiking Neural Networks

In a number of studies, different learning algorithms for spiking neural networks have been developed. Kasinski and Ponulak [30] deliver an excellent comparative analysis of the body of literature for spiking neural networks. In this section we show how a learning rule can be formally derived. The rule will then be implemented in a spiking neural network that is learning automatic word hyphenation (Chapter 6).

Moreover, we find it natural to start deriving a learning rule for a network without hidden layers, which is also known as a one-layered network. Then we will proceed to show how the rule can be extended to networks with one or more hidden layer(s).

3.1 Learning Rule for a One-layered Network

Our strategy is to tune the weights iteratively in order to minimize a given error function. To achieve this goal we will use the *gradient descent* method, which is a first-order optimization algorithm. Informally, this can be described as taking steps proportionally to the negative of the gradient of the function at the current point. This will then give a local minimum of the function as a result.

It should also be noted that our learning rule is heavily inspired by the *backpropagation* algorithm [19], and it could be characterized as a modification of it. As in this algorithm, the error measure is determined by the difference between the desired output and the output from a forward run of a program cycle.[2] An important element is to decide how the output should be coded. In conventional backpropagation, the output is represented as a real number, typically in the range from 0 to 1. But in a spiking neural network, the output is a spike train. If the whole train should be used to calculate the network error, it would be hard to determine which actual output spike should be paired to which desired output spike. We could of course arrive at a situation in which the actual output spike train has firing times $f_j^1, f_j^2 ... f_j^n$ and the desired output spike train has firing times $d_j^1, d_j^2 ... d_j^n$, that is to say that the two trains have the same numbers of spikes. Then it would be easy

[2]Since learning is taking place in discrete steps, the program activity resulting in such a step can be called a program cycle.

to find an error measure by subtracting $f_j^1 - d_j^1, f_j^2 - d_j^2 ... f_j^n - d_j^n$ But the numbers of spikes in the two trains could just as well be different (f.ex. one train consisting of two spikes, the other one of five spikes), then the way of calculating the error measure would be less straightforward. But we find that it is possible to make a simplification, since we do not usually need a very large representational power in the output neurons. The reason for this is that there is most often a limited number of classes that the output can belong to. The simplification we make is that, instead of using the entire spike train, we take only the first spike into consideration. To determine the error of the network, we will use the sum of the squared differences between the desired spike time d and the actual spike time t:

$$E_{net} = 1/2 \sum_{j \in J} (t_j^1 - \hat{d}_j^1)^2 \tag{3.1}$$

Thus $\hat{d}_j^1$ represents the desired first spike time and t_j^1 the actual first spike time of neuron j, while J denotes the output layer. (In t_j^n, the n is the spike number in the spike train of neuron j. So then t_j^3 would denote the time of the third firing of this neuron.)

Other error functions are of course possible. Still this one is quite simple, and it has been shown to serve its purpose through wide usage. It is also mathematically convenient in that taking its partial derivative with respect to the spike time: $\frac{\partial E_{net}}{\partial t_j^1}$, we get simply $t_j^1 - \hat{d}_j^1$, as shown in equation 3.4. We note that the factor $1/2$ of the error function in 3.1 cancels out the exponent when the function is differentiated.

Then, to minimize the network error, each weight should be changed proportionally to the derivative of the network error with respect to this weight. We should keep in mind that in our architecture, each connection between any two neurons is subdivided into k synaptic terminals, as illustrated in Figure 10.

The weight-change for a synaptic terminal between neuron i and neuron j can then be expressed as:

$$\Delta w_{ij}^k = -\eta \frac{\partial E_{net}}{\partial w_{ij}^k} \tag{3.2}$$

in which η is a (usually quite small) constant called the learning rate, and w_{ij}^k is the weight of the k'th synaptic terminal between neuron i and output neuron j.

Now, in order to get a more useful expression, the last factor in the above formula can by means of the chain rule be expanded to:

$$\frac{\partial E_{net}}{\partial w_{ij}^k} = \frac{\partial E_{net}}{\partial t_j^1} \frac{\partial t_j^1}{\partial w_{ij}^k} \tag{3.3}$$

We will now look at the two factors on the right-hand side of the above formula separately. We find that, from the definition of the error function (Eq. 3.1), the first one can be expressed quite simply as follows:

$$\frac{\partial E_{net}}{\partial t_j^1} = t_j^1 - \hat{d}_j^1 \tag{3.4}$$

Computing the second factor of the right-hand side of equation 3.3 is somewhat less straightforward. This factor expresses that there is a relation between the spike time and a change in the synaptic weight. Even though we know that a weight change will influence the spike time, we have no formula that describes this relation. Several solutions have been proposed to this problem. In [12], it is proposed that for a small temporal interval around a spike, the relationship between the (simulated) neuron potential and the spike time can be approximated as a linear function. Then it will also be possible to calculate its derivative, which is essential in the gradient descent method. Other approaches are presented in [17] and [16]. All of these would be very interesting for our purpose, but they exhibit mathematical complexity at a level that we have found too difficult to follow and implement.

We have therefore investigated alternative methods and found that the work of Olaf Booji [9] is simpler and also appropriate for our purposes. The rest of this chapter is as a whole taken from this work, but with certain modifications of our own, which will be clearly pointed out at their place. We have found it necessary to present a mathematically precise derivation of a learning rule, in order to make our solution comprehensible, and to explain the basis of our implementation. Our thesis should be self-contained, which justifies this extensive use of a single source. It should also be noted that developing a new theoretical model for a spiking neural network is not at all a common achievement, even at the scientific research level within this field.

The factor that we are currently looking at is:

$$\frac{\partial t_j^1}{\partial w_{ij}^k} \tag{3.5}$$

We start by noting that the spike time t_j^1 can be expressed as a function of the synaptic weight w_{ij}^k:

$$t_j^1 = t_j^1(w_{ij}^k) \tag{3.6}$$

The following equations (3.7 - 3.9) deviate a bit from their counterparts in [9], because we find our notation to be formally more correct. But these equations lead up to eq. 3.10, which is the same as in our source. The potential u_j of neuron j is a function of both the synaptic weight w_{ij}^k and the spike time t_j^1, so then we get

$$u_j = u_j(w_{ij}^k, t_j^1(w_{ij}^k)) \tag{3.7}$$

If we at time t_j^1 get a spike from neuron j, it is because the neuronal potential u_j reached the threshold ϑ at this point in time, which can be expressed as

$$u_j(w_{ij}^k, t_j^1(w_{ij}^k)) = \vartheta \tag{3.8}$$

Since the potential is ϑ, which is a constant, for every t_j^1, then we must have

$$du_j(w_{ij}^k, t_j^1(w_{ij}^k)) = 0. \tag{3.9}$$

where d is the differential of u_j.

Then the above equation (3.9) can be expanded to:

$$\frac{\partial u_j}{\partial w_{ij}^k} dw_{ij}^k + \frac{\partial u_j}{\partial t_j^1} \frac{\partial t_j^1}{\partial w_{ij}^k} dw_{ij}^k = 0 \tag{3.10}$$

We find that the above equation can be simplified by what we can think of as a division by dw_{ij}^k. (Although this operation may not be acceptable in a strictly formal mathematical sense, it will still yield correct results.) Then we get:

$$\frac{\partial u_j}{\partial w_{ij}^k} + \frac{\partial u_j}{\partial t_j^1} \frac{\partial t_j^1}{\partial w_{ij}^k} = 0 \tag{3.11}$$

Looking closely at the above formula, we find that the second factor of the last term is actually the same as the second factor on the right-hand side of equation 3.3. If we can compute the other two terms, we would be closer to a solution. To see how these derivatives can be calculated, we take a new look at the formula for the potential of a neuron:

$$u_j(t) = \eta(t - t_j^f) + \sum_{i \in \Gamma_j} w_{ij}^k \sum_{t_i^g \in \mathcal{F}_i} \sum_{k=1}^{l} \epsilon(t - t_i^g - d^k) \tag{3.12}$$

Then we find that the first term, that is the partial derivative with respect to the weight, is described by the equation:

$$\frac{\partial u_j(t_j^1)}{\partial w_{ij}^k} = \sum_{t_i^g \in \mathcal{F}_i} \epsilon(t_j^1 - t_i^g - d^k) \tag{3.13}$$

We move on to the second term of 3.11, the partial derivative of the potential with respect to the first spike time. Since we are so far only looking at the first spike time of the output neuron, we do not need to take the refractoriness term into account. Then we get

$$\frac{\partial u_j(t_j^1)}{\partial t_j^1} = \sum_{i,k} \sum_{t_i^g \in \mathcal{F}_i} w_{ij}^k \epsilon'(t - t_i^g - d^k) \tag{3.14}$$

The equations 3.13 and 3.14 can now be filled into 3.11:

$$\frac{\partial t_j^1}{\partial w_{ij}^k} = \frac{-\sum_{t_i^g \in \mathcal{F}_i} \epsilon(t - t_i^g - d^k)}{\sum_{i,k} \sum_{t_i^g \in \mathcal{F}_i} w_{ij}^k \epsilon'(t - t_i^g - d^k)} \tag{3.15}$$

If we combine these results, we get a formula that can adequately express the weight-change (Eq. 3.2) for the weights of a single-layered network:

$$\Delta w_{ij}^k = -\eta \frac{-\sum_{t_i^g \in \mathcal{F}_i} \epsilon(t - t_i^g - d^k)}{\sum_{i,k} \sum_{t_i^g \in \mathcal{F}_i} w_{ij}^k \epsilon'(t - t_i^g - d^k)} (t_j^1 - \hat{d}_j^1) \tag{3.16}$$

3.2 Learning Rule for Networks with Hidden Layer(s)

We will now demonstrate how the learning rule can be extended to comprise more than one layer of adjustable weights. In our derivation we will use a network with one hidden layer of neurons, that is to say that we have two layers of weights to be tuned. But we will also show how this can be generalized to an arbitrary number of layers. In Figure 12 we find the basic units of a (very) simplified two-layer spiking neural network. It is clear that both the weights going from the input layer H to the hidden layer I and the weights from the hidden layer to the output layer J need to be adjusted. We have already derived the necessary weight change for the weights w_{ij}^k (leading to the output layer) in the previous subsection. It is still the same basic idea

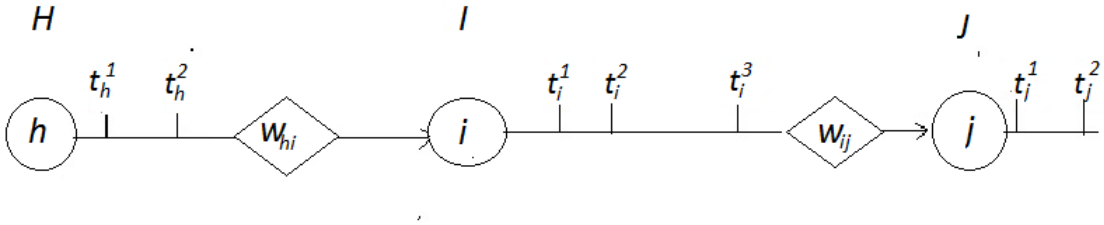

Figure 12: A simplified SNN with one hidden layer. Here neuron h belongs to input layer H, neuron i belongs to hidden layer I and neuron j belongs to output layer J. The small vertical bars represent spikes.

that will be used to derive the rule for adjusting the weights w_{hi}^k between the input and hidden layer. The gradient descent method remains central:

$$\Delta w_{hi}^k = -\eta \frac{\partial E_{net}}{\partial w_{hi}^k} \tag{3.17}$$

We see that once again we need to compute the network error with respect to the weight for every synaptic terminal of the layer. But this time we must keep in mind that a neuron of the hidden layer can fire several times, as opposed to an output neuron, where only the first spike will be reckoned. The network error then depends on all the spikes t_i^g of the hidden-neuron. as demonstrated by the following formula:

$$\frac{\partial E_{net}}{\partial w_{hi}^k} = \sum_{t_i^g \in \mathcal{F}_)} \frac{\partial t_i^g}{\partial w_{hi}^k} \frac{\partial E_{net}}{\partial t_i^g} \tag{3.18}$$

We will first look at the last factor on the right-hand side of the above equation, the partial derivative of the network error with respect to the spikes of a hidden-neuron. This is depending on the derivatives of the errors with respect to all the spikes of the neurons of the following (succeeding) layer. These neurons are denoted by Γ_i, and can be defined as:

$$\Gamma_i = \{j | j \text{ is postsynaptic to } i\}$$

It is possible to expand this factor (the derivative of the network error with respect to the spike t_i^g of a hidden-neuron) as follows:

$$\frac{\partial E_{net}}{\partial t_i^g} = \sum_{j \in \Gamma_i} \frac{\partial t_j^1}{t_i^g} \frac{\partial E_{net}}{\partial t_j^1} \tag{3.19}$$

It turns out that the last factor on the right-hand side of the above equation is already given an expression in equation 3.4. What is left is then to calculate the partial derivative of the first spike of an output neuron with respect to the spike of a hidden-neuron. Here we can use the same idea as in deriving equation 3.11. Then we get

$$\frac{\partial u_j(t_j^1)}{\partial t_i^g} + \frac{\partial u_j(t_j^1)}{\partial t_j^1}\frac{\partial t_j^1}{\partial t_i^g} = 0 \tag{3.20}$$

The first factor of the second term on the left-hand side of the above equation was computed in equation 3.14. Since we are searching for an expression for the second factor of this term, we need to calculate the first term. Again using the formula for the potential (3.12), we find the following:

$$\frac{\partial u_j(t_j^1)}{\partial t_i^g} = -\sum_{k=1}^{l} w_{ij}^k \epsilon'(t_j^1 - t_i^g - d^k) \tag{3.21}$$

We can now combine the above equation with 3.14 to get:

$$\frac{\partial t_j^1}{\partial t_i^g} = \frac{\sum_k w_{ij}^k \epsilon'(t_j^1 - t_i^g - d^k)}{\sum_{i,k}\sum_{t_i^f \in \mathcal{F}_i} w_{ij}^k \epsilon'(t_j^1 - t_i^f - d^k)} \tag{3.22}$$

The above formula along with equation 3.4 can now be filled into equation 3.19, which then reads:

$$\frac{\partial E_{net}}{\partial t_i^g} = \sum_{j \in \Gamma_i} \frac{\sum_k w_{ij}^k \epsilon'(t_j^1 - t_i^g - d^k)}{\sum_{i,k}\sum_{t_i^f \in \mathcal{F}_i} w_{ij}^k \epsilon'(t_j^1 - t_i^f - d^k)}(t_j^1 - \hat{d}_j^1) \tag{3.23}$$

We then turn to the first factor of the left-hand side of equation 3.18 , which is the partial derivative of a hidden-layer neuron with regard to a synaptic weight leading to that neuron. We use essentially the same method as we used between the hidden layer and the output layer (equations 3.8 to 3.11). But here we also have to remember that the spike t_i^g is not necessarily the first spike of the neuron i, and therefore we must also take the refractoriness-term into account. As justified in section 2.3, we do the simplification of just using the previous spike in our equation,as opposed to using the neuron's entire spike train. As we showed there, this will make our computations lighter (i.e. decrease their level of complexity) while at the same

time not affecting the results in a significant way. Then, departing from equation 3.11, we now get:

$$\frac{\partial u_i(t_i^g)}{\partial w_{hi}^k} + \frac{\partial u_i(t_i^g)}{\partial t_i^g}\frac{dt_i^g}{dw_{hi}^k} + \frac{\partial u_i(t_i^g)}{\partial t_i^f}\frac{\partial t_i^f}{\partial w_{hi}^k} = 0, \tag{3.24}$$

with the relation $f < g$ for the index letters f and g. At this point we deviate from [9]. We also note that this deviation is not only formal, but that it will affect the computations and thus also our implementation.

The first term of the above equation is once again found using the formula for the potential of the neuron (3.12) in the same way as was done in eq. 3.13:

$$\frac{\partial u_i(t_i^g)}{\partial w_{hi}^k} = \sum_{t_h^p \in \mathcal{F}_h} \epsilon(t_i^g - t_h^p - d^k) \tag{3.25}$$

Then we come to the first factor of the second term. This resembles the case we just had in equation 3.14, but again we have to extend it a bit to include the refractoriness term:

$$\frac{\partial u_i(t_i^g)}{\partial t_i^g} = \eta'(t_i^g - t_i^f) + \sum_{h,k}\sum_{t_h^p \in \mathcal{F}_h} w_{hi}^k \epsilon'(t_i^g - t_h^p - d^k), \tag{3.26}$$

where t_i^f denotes the spike preceding t_i^g in the spike train $\mathcal{F}_i$ of neuron i.

Finally, we need an expression for the first factor of the third term of 3.24, which is the derivative the potential as a function of the spike time t_i^g with respect to a preceding spike t_i^f :

$$\frac{\partial u_i(t_i^g)}{\partial t_i^f} = \eta'(t_i^g - t_i^f) \tag{3.27}$$

Combining the equations 3.24 to 3.27, we get an expression for the first factor of the right-hand side of equation 3.18:

$$\frac{\partial t_i^g}{\partial w_{hi}^k} = \frac{-\sum_{t_h^p \in \mathcal{F}_h} \epsilon(t_i^g - t_h^p - d^k) + \eta'(t_i^g - t_i^f)\frac{\partial t_i^f}{\partial w_{hi}^k}}{\eta\prime(t_i^g - t_i^f) + \sum_{h,k}\sum_{t_h^p \in \mathcal{F}_h} w_{hi}^k \epsilon'(t_i^g - t_h^p - d^k)}, \tag{3.28}$$

in which the relation between t_i^f and t_i^g is still the same as above stated (after equation 3.26). Furthermore it can be noted that this equation is recursive,

in that the partial derivative $\frac{\partial t_i^g}{\partial w_{hi}^k}$ is formulated in terms of $\frac{\partial t_i^f}{\partial w_{hi}^k}$, in which t_i^f is the spike preceding t_i^g in the spike train of neuron i. For this reason it is necessary to start computing this expression with the first spike in the train, and then continue in the given order until the last one is computed.

Then, to conclude this section, the expression given in 3.28 together with equation 3.23 can now be filled into the right-hand side of equation 3.18. This gives us a formula to compute the partial derivative of the network error with respect to the weight between a hidden neuron and its synaptic predecessors. This will in turn help us adjusting the synaptic weights in a manner that will reduce the network error.

4 Software Implementation

First a few words about the process. Initially, we read some papers on the SyNAPSE project, which is currently being carried out by IBM Research. This is a project in which spiking neural networks play a crucial role. At the time we were hoping to be able to profit from the software environment that they have created as part of the project. But IBM did not respond to our requests, so we had to choose a different approach. We then decided to try to implement from scratch our own spiking neural network, based on the theoretical model that was presented in chapter 3.

4.1 Tools

We have used Java as our programming language, mainly because of our own prior experience and personal preference. Many other options were of course available, but we have no indication that any other language would have served our purposes better.

As IDE (Integrated Development Environment) we have used Eclipse.

4.2 Representation

Without going into all the details, we will now have a look at how major elements of our neural network are represented in the implementation.

4.2.1 Neurons

The first question would be how a given set of neurons could be represented. We could of course have implemented a separate *Neuron* class. But since the only interesting attributes on a neuron in this context are its firing time (or *spike*) and its potential, this would create unnecessary overhead. Our solution was to represent the neuron's spikes as an array of integers, in which each integer corresponds to a firing time simulated in milliseconds from a relative start time. Then each neuron layer would be an extra dimension to the array, and finally the set of layers would constitute the third dimension. Thus we arrive at representing the total set of the spikes of the neurons as a three-dimensional integer array.

The neuron's potential, simulating the electric potential of the biological neuron, is represented by a decimal number. The neurons of a layer would

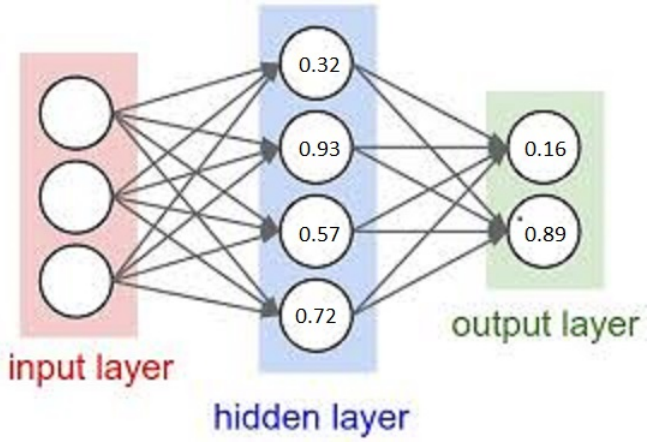

Figure 13: A possible configuration of neuron potentials. Note that the input layer neurons do not have potential as an attribute.

then be an array of these numbers, and the total set of neurons gives us a two-dimensional array representing potentials. We should note that the input layer neurons have spike trains, but no potential. Incoming spikes is a logic requirement to create change in potential, and these neurons do not have any incoming spikes. Rather, the source of their spike trains is an input parameter. A possible configuration of neuron potentials is shown in Figure 13.

4.2.2 Synaptic Weights

As already stated in section 1.1.4, the *synaptic weight* is a measure describing to what extent a spike will affect the postsynaptic neuron. As explained in section 2.4 and illustrated in Figure 10, we use in our implementation multiple synapses and thus multiple synaptic weights between any two connected neurons. Each synaptic weight is represented by a decimal number indicating the above described measure. Then the connection between two neurons will be an array of decimal numbers, all connections departing from a neuron will be a two-dimensional array, the connections from the set of neurons in a layer to the subsequent layer will be a three-dimensional array, and the total set of synaptic weights will constitute a four-dimensional array of decimal numbers.

We also noted that the synaptic weights are *dynamic* of nature. Their

value can (and most often will) change at every time step. To keep track of these changes, we use another array (called "DeltaWeights"), so that every synaptic weight has a corresponding *delta-weight* value that will be added to it (or subtracted from it, if negative) at the end of each time step cycle. Thus the *delta-weight* array will be reset after every time step, while the weights array will be updated.

4.2.3 Time Window

In biological neural networks, all activity goes along the time line. We also want to model this behaviour in our implementation, although we have not been experimenting with data that are strictly temporal in nature. This is done by choosing a certain *time window*, which is then divided into discrete steps. The size of the window is in general set experimentally. It must be big enough to allow for learning to happen. Still, if the time window is too extended the results may be levelled out, and it may give very long running times of the program.

4.3 Classes and Methods

As can be seen from Figure 14 our software structure is very simple viewed from the class level. Much of the challenge lies in programming compound mathematical formulas, and therefore the complexity is found in the methods rather than in the higher level structure.

It should also be noted that two different programs have been used for experimentation on the hyphenation problem, as explained in chapter 6. One is the software for the spiking neural network that is referred to in this chapter. The other program is an implementation of the conventional *backpropagation* algorithm. The software structure of this program will not be elaborated here, since part of it is taken (with permission) from other sources.

4.3.1 MultilayerSNN

In the following, we will have a brief look at the most central methods in the class *MultilayerSNN*.

neuronPotential This method computes the electric potential for a single neuron. The neuron may be a part of a hidden layer or the output layer. The computation is done by implementing formula 2.5.

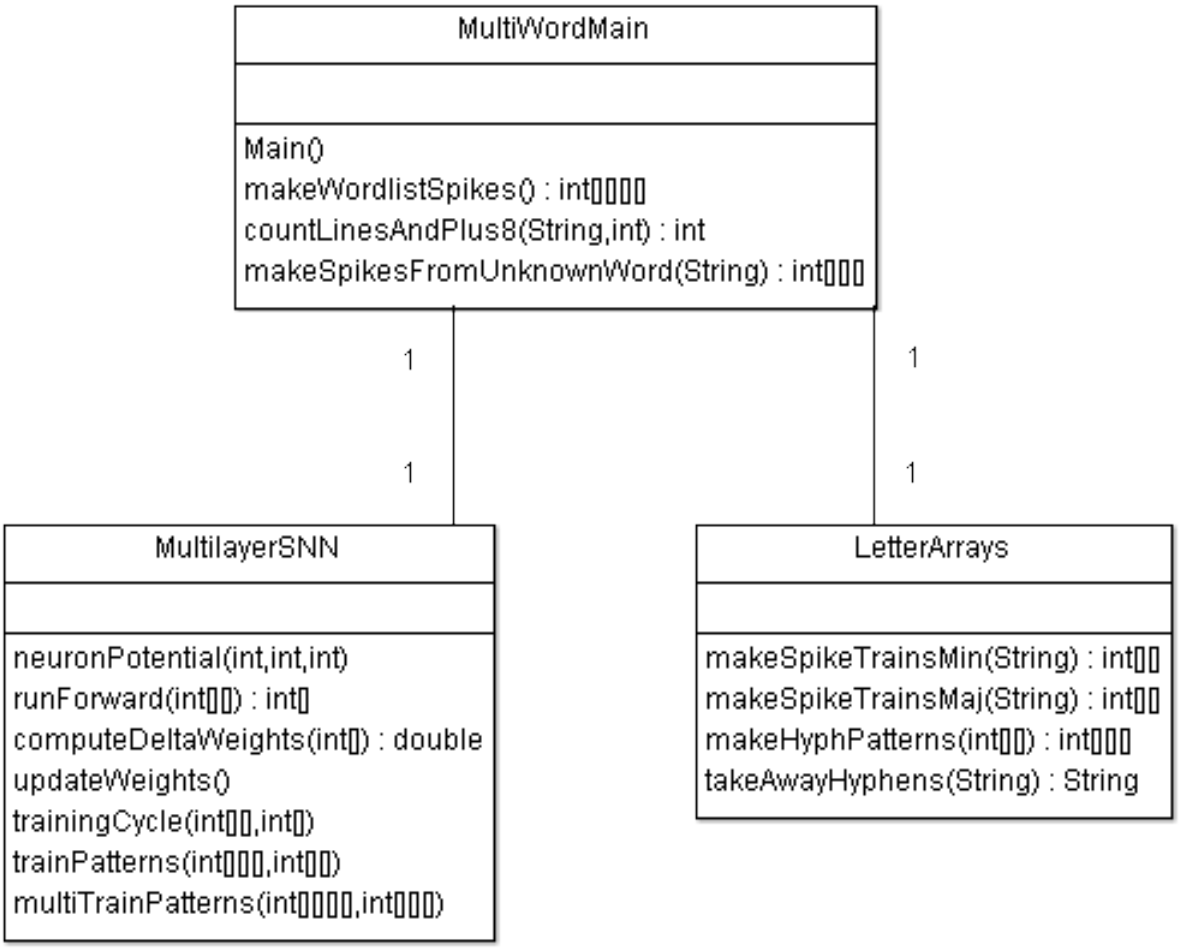

Figure 14: Software structure of Hyphenation problem. Utility methods have been excluded.

runForward After resetting the potential of all neurons, this method runs through the entire network using the above mentioned *neuronPotential* method. It is starting from the input layer and ending up with the output neurons.

computeDeltaWeights The difference between the results from *runForward* and the desired outputs gives us the network errors. These errors are here propagated backwards from the output layer through the network. The derivations in chapter 3 is the base of the computations in this method. For every synaptic weight, its *delta value* (i.e. value to be added or subtracted to the weight) is computed combining the formulas 3.23 and 3.28 to get a value for eq. 3.18.

updateWeights Using the values computed in the *computeDeltaWeights* method, the weights array is updated. This in turn should be a step towards minimizing the total network error.

trainingCycle This method essentially combines the methods *runForward* and *computeDeltaWeights* into a cycle that is repeated throughout the training process. In the hyphenation problem, the input parameter is corresponding to a word with a given hyphenation.

trainPatterns In this method, the input parameter represents all possible hyphenations of a word. The above described training cycle is repeated for the chosen number of iterations. In each iteration, a pattern from the input array is chosen randomly. In the hyphenation problem (chapter 6), one pass of this method corresponds to training a single word.

multiTrainPatterns Here, an entire word list serve as input parameter. The method first gives all synaptic weights a random value within a set interval. Then the *trainPatterns* method is called once for every word in the word list we want to train. At the end, the weights should be adjusted in order to be able to recognize good hyphenations.

The class contains around 800 lines of code.

4.3.2 LetterArrays

We will also give a brief description of the methods of the class *LetterArrays*:

Constructor Here a two-dimensional array is set up, in which every letter of the Norwegian alphabet is represented as an array of spikes.

makeSpikeTrainsMin This is the method that encodes a word, represented as a *String* of minuscules (lower-case letters), as a two-dimensional array of spikes.

makeSpikeTrainsMaj This is the same as the above method, but is used whenever the input word list is written in majuscules (i.e. upper-case letters).

makeHyphPatterns This method takes a word represented as spikes as input parameter. Then it returns an array of the same word, but now with a hyphen in one of its possible locations. (For example the word "BORD" will return "B-ORD", "BO-RD" and "BOR-D".)

makeHyphenArray From a training word list of hyphenized words, this method makes an integer array of the acceptable hyphen locations. (For example the word "OPP-TRINN-ET" will return the array [3,8]) This array is then used in training the network.

Some of these methods will be further elaborated in chapter 6.

4.3.3 MultiWordMain

Finally, we will give an outline of the *MultiWordMain* class:

makeWordlistSpikes This method uses methods from the *LetterArrays* class, turning an entire word list into spike arrays. These arrays will then be the input for training the system.

main This is of course the starting point of the system. Here the word to be tested is set, along with a range of different parameters, such as interval for synaptic weights, time window, spike threshold and learn rate. You can also choose if you want a network with or without hidden layers. Then the training takes place, calling the *multiTrainPatterns* method from the *MultilayerSNN* class. Finally, the test word is given a run, and the results are displayed.

In the description of the above three classes, utility methods have been excluded. The complete code for these classes can be found at [15] (GitHub repository).

5 Validation

As already mentioned, there are numerous areas in which artificial neural networks have been successfully applied. This is true, whether we talk about neural networks in general, or of the special case of spiking neural networks. I will now describe some of the results that were found using the spiking neural network that has been developed. We will look at simple boolean functions as a starting point, and then proceed to a problem involving higher numbers of input and output neurons, as well as spike trains with more than one spike for each input neuron. Finally we will make a simple application for word hyphenation based on pattern recognition. In this case, we will also compare the performance of the spiking neural network to the results of a more conventional approach based on the backpropagation algorithm.

5.1 Boolean Functions

We assume that boolean functions are the simplest functions we can think of. They therefore require very simple network topologies, and we have started out with a one-layered network with two input neurons and one output neuron (see.fig 15)

We started out with an AND-function (see Table 1). As it was discussed in section 1.1.5, it is not always obvious how data should be encoded into spikes. Here we have chosen (inspired by [12]) to simply encode the input value "TRUE" as a spike at $t = 0$, and the input "FALSE" as a spike at $t = 6$. For the output, we have a spike at $t = 10$ as "TRUE", and $t = 16$ as "FALSE". We then get the table as depicted in Table 2.

It may of course be asked what is the justification for these values to be chosen. The answer is that they are more or less arbitrary, we just have to keep in mind that in order for an input spike to influence the output at time t, the input spike of course need to come at an earlier point in time.

As will be explained in the following, the network was able to learn the function. This became clear as there was found a relation between the number of iterations and the results of the error function (described in Equation 3.1). This error function is measured first as an average over the first twenty runs, and then over the last twenty runs, which should give us a good picture of the error reduction. It seemed that the network needed around 200 iterations for the error function to drop below a measure of 1.0 (see Figure 16), which we consider a satisfying result. But it turns out that the error

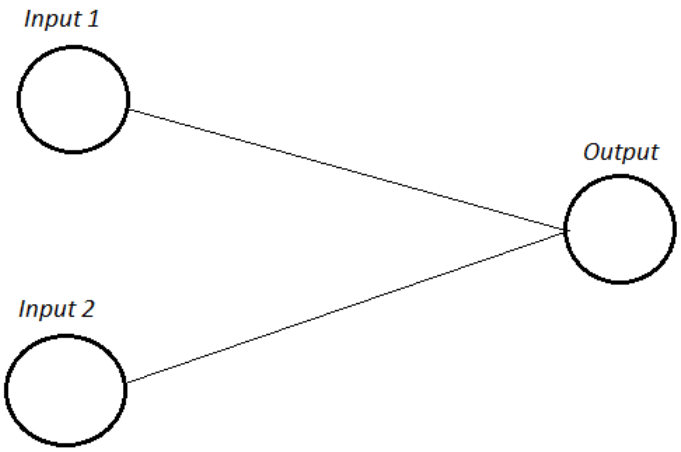

Figure 15: A simple SNN for learning boolean functions

Input 1	Input 2		Output
True	True	→	True
True	False	→	False
False	True	→	False
False	False	→	False

Table 1: Boolean AND function

Input 1	Input 2		Output
0	0	→	10
0	6	→	16
6	0	→	16
6	6	→	16

Table 2: Boolean AND encoded as spikes

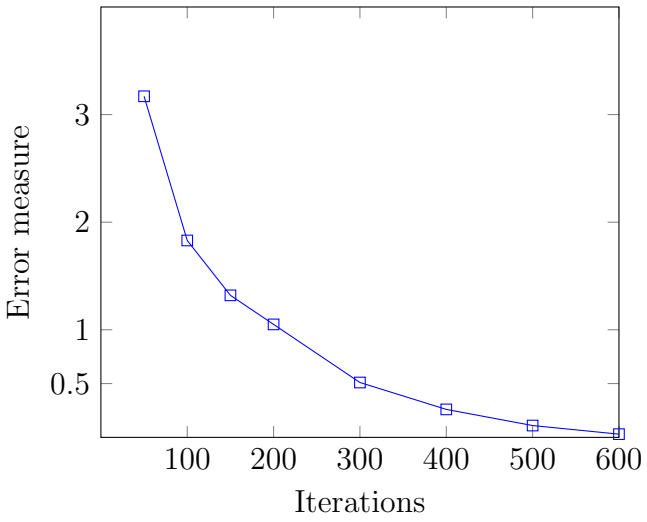

Figure 16: Relation between errors and number of iterations

can be arbitrarily minimized by increasing the number of iterations, which is also indicated by the figure. As could also be expected, similar results were found for the boolean OR-function.

We have already seen (Figure 5) that the boolean AND- and OR-functions are linearly separable, whereas the XOR ("exclusive OR", see Figure 17) is not. It was shown in 1969 by Minsky and Papert [21] that a one-layered artificial neural network (or *perceptrons*, see section 2.1.1) would not be able to learn functions that are not linearly separable. Although Booji [9] claims that he has done this with a one-layered spiking neural network, it seems quite obvious (as he also admits) that this can only be done in a so-called *hair-trigger* situation. This is to say that the network can in fact give the correct results, but only if the parameters are very precisely defined, so that any disturbance (or "noise") will cause the network to fail. Since we know that noise generally will be present in the real world, this means that this solution has no robustness or value for real-world applications.

5.2 Validation on Generated Spike trains

We still consider a one-layered network, but now we will use input of higher complexity, and we will also use several neurons for output. For the input

x	y	x XOR y
0	0	0
0	1	1
1	0	1
1	1	0

Figure 17: The XOR function

neurons, we will generate random spike trains within a time window of 20 units. That is to say, the spike train can in theory contain from 0 to 20 spikes, but it will typically have in the range from 1 to 6 spikes. The average is 3 spikes, as we from experimenting have chosen to give each time step a 15% chance of having a spike. Furthermore, we used 10 input neurons and 4 output neurons. Each output neuron represents one pattern, and presented with the pattern it is trained to recognize, it should fire a spike earlier than the three others. In other words, the first output neuron to fire a spike determines which class the input pattern belongs to. This is illustrated in Figure 18 (although for simplicity, only 6 input neurons are drawn, whereas we tested on 10 input neurons). Then, for a test to be successful, we demand that the right output neuron fires first, and also that none of the others fire at the same time. Testing 100 times, we found that output neuron 1 and 4 had a success rate of 98%, while neurons 2 and 3 both had a rate of 100%. The result of the error function was reduced from an initial value of 192.01 down to 2.89.

We are aware that in many real-world applications, a certain measure of noise in the data is inevitable. Therefore, we also performed the above experiment on "noisy" patterns. More precisely, for one of the ten input neurons we generated a new randomized spike train, totally different from what was used in the pattern training. This should give us about 10% noise in the input data. A comparison between the original and noisy version of the experiment is shown in Table 3. We find that the performance went somewhat down when noise was introduced, as could be expected. But still it is far from a total failure. This can serve as an illustration of a general feature of neural networks,in that they *degrade gracefully*, meaning that their performance are reduced slowly as the data get more and more insufficient. This can stand as a contrast to traditional computational approaches, which

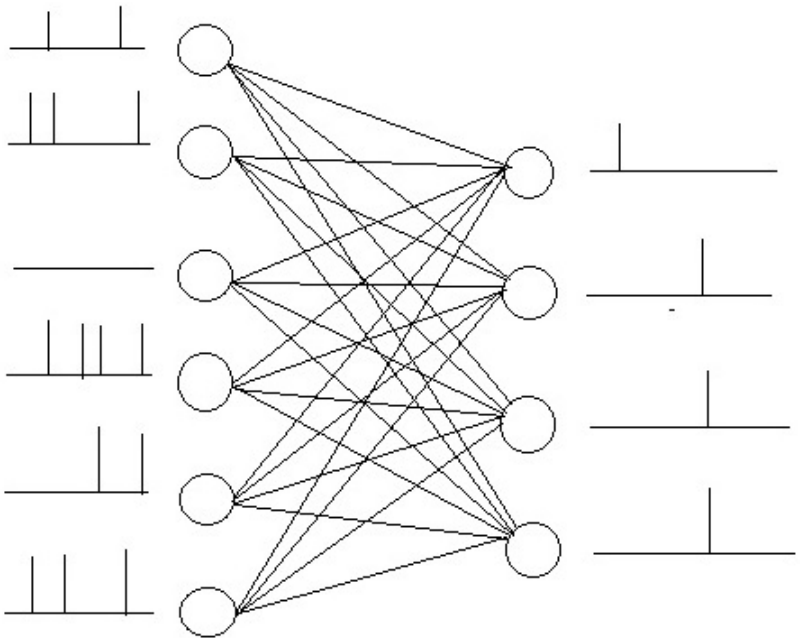

Figure 18: Network for testing generated spike trains

Output	Original	Noisy
Neuron 1	98%	84%
Neuron 2	100%	79%
Neuron 3	100%	86%
Neuron 4	98%	85%

Table 3: Success rate for generated spike trains

Input 1	Input 2	Target output	Our output
0	0	16	13
0	6	10	13
6	0	10	13
6	6	16	13

Table 4: XOR results

usually give you either the exact answer, or no (sensible) answer at all.

5.3 Multilayer Networks

So far, we have been looking only at one-layered spiking neural networks. It was our aim to use a multi-layered SNN for some of the tasks that the one-layered network could not accomplish. This include the already mentioned boolean XOR-function. However, we did not get the results we were hoping for, as the table 4 displays. (Remember that for input, TRUE is represented by $t = 0$ and FALSE by $t = 6$, while for output TRUE is represented by $t = 10$ and FALSE by $t = 16$.) We find that, instead of learning the desired output values, the network is levelling out the results. The measure of the error function is reduced from an initial value of 201.55 to 9.45, indicating that the network is actually learning. Still, there are many witnesses that it should be possible to learn the XOR with a multi-layered SNN ([9], [12]). But we have been unable to find out why the network is not giving the desired results in this case. Since our theoretical model has been thoroughly verified, we suspect that there may be some kind of error in our implementation. The code has of course also been the object of tedious scrutiny, but its complexity is hard to reduce into more manageable units, and the problem has remained unsolved.

6 The Hyphenation Problem

We have been looking at boolean functions and artificially generated spike trains, but still we have not been using our network on real-world data. We will now turn to the problem of automatic word hyphenation. In books, newspapers and other publications, words sometimes need to be split up and a hyphen inserted. But as we know, there are good positions for a hyphen, less good (but still considered possible), and finally there are hyphen positions that should be avoided completely. Now, this is by no means an unsolved problem. The most straight-forward solution is to simply apply an extensive word-list. Since every language has a finite number of words (de facto true, if not in theory), it is not beyond the capacity of modern computers to search through a hyphenated word list whenever the problem arises. This will of course demand a separate list for every language, but still the challenge is manageable.

Another approach is to find rules that govern hyphenation and patterns that are repeated in certain ways. In the publisher program LaTeX, an algorithm is used that matches candidate words against a set of hyphenation patterns [22]. Still, the task of finding these rules and patterns is quite a complex one. Further complicating matters, it is known that even within the rather homogeneous English language, there are quite some differences in the rules governing the British variant and the American one.

This leads us to the neural network approach. Here the idea is not to sort out the rules and patterns explicitly, but rather to let the network itself find the relevant patterns. This is done by training the network on a hyphenated word list. There can be different opinions as to how extensive this training list should be. Some are advocating that practically all available words of the given language should be in the list. In this thesis we take a different approach: If we use a smaller fraction, e.g. 10-20% of the total number of hyphenated words, the same pattern should be extractable. Otherwise put, the findings using 10 000 words in your base should be only marginally different from the results you get if you use 100 000 words.

We will actually try two different approaches to the hyphenation problem: One using the conventional backpropagation algorithm, the other one using our spiking neural network implementation. Then we will compare their performances and discuss our findings.

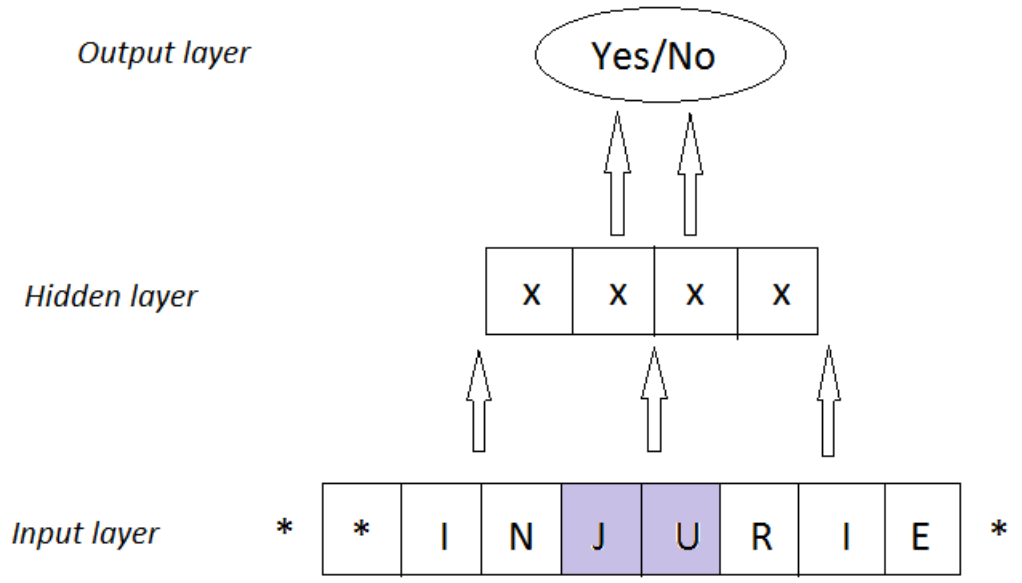

Figure 19: Illustration of our strategy for deciding acceptable hyphen positions

6.1 Hyphenation using backpropagation

The backpropagation algorithm has already been mentioned (sections 2.1.2 and 4.1), and it was considered quite a breakthrough when it appeared. The base class of the version used in this thesis has been developed by Terje Kristensen [23].

6.1.1 Data Representation

The first question to answer is how the data should be represented. Here we have used the approach of [23]. Each alphabetic letter will be encoded in a unary way: They will simply be represented by a string of the length of the alphabet (29 letters in Norwegian, plus one for the empty space), consisting of 29 zero's and one 1. The letter 'A' will then be "10000(...)0", the letter 'B' becomes "01000(...)0" and so on. Finally the last Norwegian vowel will be "000(...)0010" and the empty space, replaced by an asterisk ('*') will be "000(...)0001". The use of the asterisk will be explained shortly.

Our strategy will be to look at every possible hyphen location in a given

*	*	*	I	N	J	U	R
*	*	I	N	J	U	R	I
*	I	N	J	U	R	I	E
I	N	J	U	R	I	E	*
N	J	U	R	I	E	*	*
J	U	R	I	E	*	*	*

Table 5: Illustration of word rolling through a window of size 8

word. A possible hyphen position is always between the letters of a word, so then a word of length n gives us $n - 1$ candidate positions. Then there will be a yes/no-question for the network to answer between any to consecutive letters in a word. But we know that two letters is by no means sufficient to decide such a question, we need a *context*. The two letters in focus along with the context will form a scope or working window. This is illustrated in Figure 19. Then how large should this window be? This will always be a trade-off between efficiency and accuracy - a larger window will give better accuracy, a smaller one will give better efficiency. We have chosen a symmetric window with 8 letters as input for our network. If a word has fewer letters than this, the window will be filled with asterisk (space) symbols.

6.1.2 Network Topology

The number of input neurons will actually be the window size multiplied with the length of the current alphabet. As we count also the asterisk symbol, this will amount to 8 * 30 = 240 in our case. The output consists of only a single neuron. This neuron will have a real number as its output. Whenever this number exceeds 0.5, it will be interpreted to mean that this is a good position of a hyphen, and the value of the output neuron will be "rounded upwards" to 1.0. If it remains lower than this value, it is not considered a suitable position, and the output will be rounded down to 0.0. This rounding is done because, as illustrated in Figure 19, each single run should give a *yes* or *no* answer.

Hyphen position 1	0.0
Hyphen position 2	1.0
Hyphen position 3	0.0
Hyphen position 4	0.0
Hyphen position 5	1.0
Hyphen position 6	0.0

Table 6: Position values

We give the word "INJURIE" as an example. It has 6 possible hyphen positions, but there are only two good positions, so it should be hyphenated as "IN-JUR-IE". The desired output of the network should then be as depicted in table 6. Of course a more advanced system would have more alternatives than these yes/no answers. Any given position could for example be labelled on a scale from 0 to 5, in which '0' would mean "Not at all acceptable", '3' could be interpreted as "possible but less desirable" and '5' could be "Very suitable" hyphen position. But for our purposes, we find it easier to measure the results of a simpler version. The questions that remain concerning the topology of the network is how many neurons should a hidden layer have? And furthermore, how many hidden layers should there be? It seems hard to find theoretical answers to these questions, so we will have to decide this on the basis of experiments.

6.1.3 Experiments

We now want to do some simple experiments using our backpropagation network. For this purpose, we have a training word list of 2000 Norwegian words, randomly chosen from a much larger list of about 67000 hyphenated words. Any word in this training list will have from 0 to 10 hyphens. An example with many hyphenations is "PA-PIR-IN-DUS-TRI-AR-BEID-ER-FOR-BUND-ET". But more typically, the words will have 1 or 2 hyphens. To keep things manageable, we concentrate on the last mentioned cases (i.e. 1- or 2-hyphen words). We will test the network both on words from the training list, and on words that are not found in this list. We also try to avoid words that are clearly compound (i.e small distinct words having been put together) in order to not simplify the task. For a word to be tested, it will be given 100 runs of training and testing, and all parameters of the

Network topology	No hidden layer	One hidden layer (10)	One hidden layer (30)	Two hidden layers (30, 6)
Hyphen position 1	0.0	0.0	0.02	0.0
Hyphen position 2	0.04	0.0	0.0	0.0
Hyphen position 3	0.45	0.78	0.78	0.66
Hyphen position 4	0.0	0.0	0.0	0.0
Hyphen position 5	0.0	0.0	0.0	0.0
Hyphen position 6	0.0	0.0	0.0	0.0
Hyphen position 7	0.0	0.0	0.0	0.0

Table 7: One-hyphen word from training list: KON-KRETE

network is reset for every new run. In other words, this should be equal to 100 single runs of the network, in which the training for every run consists of 1800 iterations. Finally, the average for every possible hyphen position is taken. The position with the highest score is then considered the best one, the second highest score may indicate a second hyphen, and so on. We also tried three different network topologies. The input and output layer were held as described above, but the following hidden layer configurations were investigated:

- No hidden layer
- One hidden layer of 10 neurons
- One hidden layer of 30 neurons
- Two hidden layers, the first of 30 and the second of 6 neurons.

As a first example we consider the word "KON-KRETE", a word from the training list with one hyphen. The results for the different network topologies are found in table 7. We see that the network correctly gave hyphen position 3 the highest value. Two other scores were just over zero, but still so low that they will not be assigned any significance. Moreover, we find that the topologies using only a single hidden layer gave slightly better performance than the topology with two hidden layers. This may be considered somewhat surprising, since it is often supposed that more layers should yield better accuracy, at the cost of efficiency.

Network topology	No hidden layer	One hidden layer (10)	One hidden layer (30)	Two hidden layers (30, 6)
Hyphen position 1	0.0	0.0	0.0	0.0
Hyphen position 2	0.55	0.87	0.75	0.3
Hyphen position 3	0.0	0.01	0.03	0.0
Hyphen position 4	0.07	0.05	0.41	0.0
Hyphen position 5	0.0	0.02	0.0	0.0
Hyphen position 6	0.0	0.0	0.0	0.0
Hyphen position 7	0.03	0.02	0.02	0.0
Hyphen position 8	0.0	0.0	0.0	0.0

Table 8: 2-hyphen word from training list: TABL-ETT-ER

Network topology	No hidden layer	One hidden layer (10)	One hidden layer (30)	Two hidden layers (30, 6)
Hyphen position 1	0.0	0.0	0.01	0.0
Hyphen position 2	0.26	0.08	0.04	0.13
Hyphen position 3	0.57	0.11	0.76	0.21
Hyphen position 4	0.05	0.0	0.0	0.0
Hyphen position 5	0.0	0.0	0.0	0.0

Table 9: 1-hyphen word not from training list: MAL-ERI

Now we turn to a word that is registered with two good hyphen positions in our training list: "TABL-ETT-ER". We then get the following results: Here we find that for all four topologies, the network favours the wrong hyphenation, putting the first hyphen at the second possible position: "TA-BLETTER". Still, the topology with a single 30-neurons hidden layer gives a significant second placement, at the fourth position, which is a desired result. The output values for the other good position (pos. 7) are throughout so low that they are not considered relevant. As above (in table 7), we note that the three-layer network (with two hidden layers) is not performing particularly well.

We also want to find out how the network will perform on words that are not in the training list. Starting with a one-hyphen word we will use "MAL-ERI" (Table 9). It is interesting to note that even if the network has

Network topology	No hidden layer	One hidden layer (10)	One hidden layer (30)	Two hidden layers (30, 6)
Hyphen position 1	0.0	0.0	0.0	0.0
Hyphen position 2	0.02	0.0	0.0	0.0
Hyphen position 3	0.99	0.99	0.97	0.94
Hyphen position 4	0.0	0.0	0.0	0.0
Hyphen position 5	0.02	0.0	0.0	0.0
Hyphen position 6	0.0	0.0	0.0	0.0
Hyphen position 7	0.0	0.0	0.0	0.0
Hyphen position 8	0.04	0.0	0.01	0.0

Table 10: 2-hyphen word not from training list: FOR-SYN-ING

not at all been trained on this word, two of the topologies performs pretty well. The choice of position 3 as hyphen location is very clear in the 30-neurons topology with one hidden layer. Not far behind follows the network without hidden layers. The situation is a bit different for the topology with one hidden layer of 10 neurons and the one with two hidden layers. Here we see that even though they also have their highest scores in positions 3, their values are much lower and they are also less distinct from the values of the other positions.

Finally in this section, we will use a two-hyphens word that is also not from our training list: "FOR-SYN-ING". We found the following results: We see that when it comes to comparing the different topologies, these are the most homogeneous results so far. They have all a very strong preference for hyphen position 3, which we recognize as a "good" location. But none of them had found the other acceptable hyphen position (pos. 6), a fact for which we have no plausible explanation. But our experiments obviously indicate that reliably finding more than one acceptable hyphen position in a word is challenging for this kind of network.

We also find interesting results when we are comparing on the one hand words that are from the training list, and on the other hand words that the network has not encountered before. It seems that the performance of the network for these two groups is not very different. The differences we found are probably more related to the fact that certain patterns are more prominent than others. From this we can further conclude that the network has actually discovered patterns in the training set of hyphenations, and that

these patterns are applied whenever the network is presented with a word, whether it be "known" or "unknown".

It was also of interest to see how the different network topologies performed. From the four different topologies that we have examined, the best overall seems to be network with one hidden layer of 30 neurons. But still the performance differences as seen against the other topologies were in general not big. It may be noted as somewhat surprising that the network without hidden layers performed almost as well as the best one. This fact will be further treated in the next chapter ("Discussion") We also did not expect that the three-layer network would (in average) come out last in the test.

6.2 Hyphenation Using Spiking Neural Networks

We have seen how a large number of input neurons for our backpropagation network resulted in a single output neuron representing a yes/no decision for any candidate hyphen position. Our approach will be somewhat different for the spiking neural network, as will be explained in the following.

6.2.1 Data Representation

The spike train is the natural data unit in a spiking neural network. Since our current data is alphabetic symbols, we need to decide whether one such symbol should be represented by a single neuron with its spike train, or by more than one neuron. In our input data, we should ensure a certain amount of spacing between any two spikes in a train, since it will be hard for the system to distinguish two spikes coming very closely together. Appropriate spacing will increase the robustness of our system, and we obtain this by only allowing spikes at predefined spots within a certain time interval. So if we have a time interval from 0 to 10 and a spike train of maximal length 3, the only possible spike "positions" may be defined to be at $t = 1$, $t = 5$ and $t = 9$. Then what will be the relation between the length of a spike train and the maximal set of symbols to be represented? To start out with the simplest possible spike train, we consider a train of maximal length 1. This can represent a set of 2 different symbols (or values): Either there is a spike, or we have the empty train. Moving on to a spike train of maximal length 2, we have the following possibilities: We can have a spike in the first position, a spike in the second position, spikes in both positions or again the empty train. This is to say that a spike train of length 2 can represent a set of 4

different symbols. Generalizing, we find that a spike train of maximal length n can represent 2^n different symbols. Or the other way around, a set of m symbols will need a spike train with maximal length at least $\lceil \log_2 m \rceil$. We will be using a set of 30 symbols: 29 letters (of the Norwegian alphabet) and the hyphen symbol. Then we find that a spike train of maximal length 5 (since $5 = \lceil \log_2 30 \rceil$) will be sufficient to represent a single symbol, and every symbol in our set will then be represented with a spike train of minimal length 1 and maximal length 5. The (time) interval for the input spike trains will be from $t = 0$ to $t = 17$. The candidate positions within this interval will be $t \in \{1, 5, 9, 13, 17\}$, which should ensure appropriate spacing between spikes. The actual coding of the different letters can be found in the constructor of the *LetterArrays* class, in the Github repository [15].

6.2.2 Network Topology

When we have decided using one input neuron for each alphabetic symbol, another question arises: How many input neurons should we use? Ideally, we should of course use as many input neurons as there are letters in a word. But, in order to train the network on a list of words, we need a fixed number of input neurons. This is somewhat parallel to what we in section 6.1.1 called a *window*. When it comes to the length of this window, we have some of the same considerations that were mentioned there. We could have used as many input neurons as the length of our longest word, but then most of the words would need to be filled in with a large number of blank spaces (or another appropriate symbol). This would probably inhibit accuracy, and certainly it would worsen efficiency. Neither should the window be too small, since it is proportional to the size of the input data that is the basis for the training of the network. The smaller the size of the input data, the more patterns will be similar, and the harder it will be to distinguish them. We will try to find a window size that balances between these considerations.

When a window size has been chosen, the words in our list will be divided into three cases:

- If its number of letters is the same as the size of the window, then no special treatment is needed.
- If its number of letters is less than the window size, the difference between the two will be filled in with blank spaces.

- If it is longer than the window size, it will need to be "rolled through", which is to say that for a word of length (window size $+n$), this word will be made into a list of $n + 1$ new strings.

We should still keep in mind that it is the hyphenated word list we are using for training the network. So when we are "rolling through" a word that is longer than the window size, we will get a hyphen either at the beginning or the end of some of the new strings we are constructing. We will use the word "RE-STRIK-TIV" as an example. With a window size of 8, this will be made into the following new strings: "RE-STRIK", "E-STRIK-", "-STRIK-T", "STRIK-TI" and "TRIK-TIV". Remember that the hyphen is here (as opposed to in our backpropagation network) counted as a distinct symbol. We find that the second and the third out of these five instances are ending and beginning (respectively) with a hyphen. These strings will then be eliminated from the training set, since a hyphen is always in-between: No real-life word begins or ends with a hyphen. As was the case in our backpropagation network, we have a single output neuron. The first spike of this neuron should give us indications about a given candidate location: An early spike should signify an acceptable hyphen position, a later spike would have the opposite meaning.

So how then does the output signal hyphen positions? The network is trained so that a (relatively) early spike from an output neuron should signal an acceptable hyphen position, while a later spike signals no hyphen. These early and late spike times are given as parameters to the network. Naturally, even the "early" output spike comes after the input interval, that is to say after the latest possible input spike. As we have seen, the input spikes are scattered in the interval $\{1 - 17\}$. Then we have chosen $t = 22$ as our target early output, and $t = 30$ as our late output. Table 11 shows us the target results for the word "ALTANEN", hyphenated "AL-TAN-EN". Here the table gives us the value 22 for hyphen positions 2 and 5, indicating acceptable hyphen locations.

6.2.3 Experiments

For a start, we wanted to train the network on a simple list of 100 words, chosen using the pseudo-random generator. Now a natural question would be why only 100 words, as opposed to the backpropagation experiment, using a list of 2000 words? But testing the network we found that running times

Table 11: Target values for "ALTANEN"

Hyphen position 1	30
Hyphen position 2	22
Hyphen position 3	30
Hyphen position 4	30
Hyphen position 5	22
Hyphen position 6	30

were so much longer for this type of network, that using a larger list would be impractical. Also, when it comes to experimenting, it does not hurt to start out with a smaller amount, and then enlarge it in the case of success. This problem of running times and efficiency will be further discussed in the next chapter.

We could of course fear that just running through the word list from top to bottom, the latter part of the list would in the end influence the weights of the network more than the earlier part. To avoid this we made a training algorithm that first ran through half of the given number of iterations, then half of the remaining half, and so on until we reached a lower limit (which we chose to set at 10 iterations). So instead of running through all iterations at once, we divided them into portions: $1/2 + 1/4 + 1/8...$ Naturally, it could be objected that this way we would never reach the full number of iterations. But since this number is more or less arbitrary, changing it by a small margin will not have any significance. To give an example: If we start out with 1200 iterations, using this method will leave us with 1180 iterations.

We proceeded to test the network with the word "FORESATT", hyphenated "FORE-SATT". The results are shown in table 12.

We see that position 4 has a slightly earlier spike than the others. In principle it is the right hyphen location, but the difference is so small that it is hard to say whether it is significant or not. Further experiments are needed to enable to answer this question. We went on to test the network with the word "STREIKEN", hyphenated "STREIK-EN", and the word "AVGRENSA", hyphenated "AV-GRENSA". These were chosen in order to try a word with late (pos. 6) and early (pos.2) hyphenation respectively. But to shorten the story a little, the results were exactly the same as in table 12. This was of course not at all encouraging. We have also tried adjusting the different free

Hyphen position 1	28
Hyphen position 2	28
Hyphen position 3	28
Hyphen position 4	27
Hyphen position 5	28
Hyphen position 6	28
Hyphen position 7	28

Table 12: Results for "FORESATT"

parameters, such as learning rate, bias factor and time constants, but with basically the same results. It seems safe to conclude that the given one-layer spiking neural network is not able to learn hyphenation in any measurable way.

So how then would a two-layer network perform on these matters? Remembering our results from the boolean functions in section 5.3, we could not be too optimistic. Nevertheless, for the sake of completeness we also performed an experiment on the word "BEFRIR", which should be hyphenated "BE-FRIR", and got the results that are shown in table 13. Not surprisingly, we find that the spike times are levelled out (as was also the case of the experiments in section 5.3). Experimenting with other words has given similar results.

As already mentioned, the running time of the experiment was also an issue. Even with a training list of only 25 words, along with the modest number of 300 iterations, training the network took more than an hour. The implications of such long running times will be discussed in the next section.

We have seen that we did not get the results we were hoping for, and we also cannot give any ready explanation for this fact. Still it seems reasonable to suspect that at least part of the problem is that every time a new word is trained, much of the previous learning will get "overwritten", and former information will be lost. In [28] we have found an interesting notion called *data reinforcement* that actually addresses this problem. They define this term as "re-presenting previous information to the network together with

Hyphen position 1	28
Hyphen position 2	28
Hyphen position 3	28
Hyphen position 4	28
Hyphen position 5	28

Table 13: Results for "BEFRIR"

the new information so that the old data is sufficiently retained and stays balanced with the new information". We did not have time to investigate this solution, but we note that it sounds promising, and it is something that we would like to pursue in our further works.

7 Discussion, Conclusion and Future Directions

We have looked at different motivations for constructing spiking neural networks, and we have considered the relationship between this kind of artificial neural network and the biological neural network as found in the brain of mammals. For comparison, we have also described different types of conventional artificial neural networks. Furthermore, we have investigated a certain mathematical theory within this field of research, and we have used it as a basis for developing from scratch our own implementation of a spiking neural network. Then we have performed different experiments with this neural network, and we have also compared it to a conventional neural network based on the backpropagation algorithm. In the following we will highlight and discuss some of our findings.

7.1 Hyphenation and Linearity

In our experiments on hyphenation using a conventional neural network, we found somewhat surprisingly that a one-layer network had almost as good performance as the multilayer networks. Still, it has been shown [21] that one-layer networks can not learn to recognize functions or patterns that are not linearly separable. Now, in the simple boolean functions it is not hard to define what "linearly separable" means, but when it comes to hyphenation it is not quite so straight-forward. We will now sketch a way to think about this problem.

For this purpose, we will use a shortened input alphabet: {A, B, C, D}, and from that a couple of newly made words: "ABC" and "BCD" (which we now define to be the only words originating from this alphabet). The letters from the alphabet could be coded by means of the same unary encoding as was used in section 6.1.1. Then 'A' would be encoded "1000", 'B' would be "0100", 'C' is "0010" and 'D' is "0001". We see that the words "ABC" and "BCD" share a common substring, namely "BC". This substring has of course only one candidate hyphen location, between the two letters. This location may or may not be an acceptable hyphen position. Now, the following hypothesis [3] seems reasonable to us: This hyphenation problem is linearly

[3] I am aware that a hypothesis normally should be formally proven, but that will be outside the scope of this thesis. The given justification should then hopefully be enough

separable if the candidate hyphen location of the substring has the same status (acceptable or not acceptable) in all the words in which the substring is found. Since we now pretend that this very alphabet has only the above two words originating from it, it is easy to exemplify: The stated hypothesis claims then that if "AB-C" is an acceptable hyphenation, then "B-CD" must also be. Likewise, if the hyphenation is not acceptable for "AB-C", then neither can it be for "B-CD".

If we then turn to natural language, it is not hard to find examples of groups of words showing that the hyphenation problem is not linearly separable according to the above given hypothesis. As examples, we can use the words "TIMELØNN", hyphenated "TIME-LØNN" and "TIMEN", hyphenated TIM-EN". In their common substring "TIME", the latter has an acceptable hyphen position between 'M' and 'E', while the former has not. On the other hand, we know that letters in natural language words are not placed totally at random. Certain combinations are frequent, while others are rare or non-existing in a given language. (This is of course the reason why the problem can be considered suitable for pattern classification in the first place.) Going through a word list, we see that even if we can find examples as the above "TIMEN" and "TIMELØNN" (another one can be "TURN-ER" and "TUR-NIPS"), they are relatively rare. More than 90% of all words starting with "TIME" will be hyphenated "TIME-(...)", and likewise for words starting with "TURN(...)". This is to say that even if the hyphenation problem in principle is not linearly separable, if we look at it statistically, it can still be considered to be quite close to this notion. This may in turn explain why our one-layer network (i.e. without hidden layers) performed almost as well as the multilayer networks for this problem.

7.2 Spiking Neural Networks and Efficiency

As already stated, we wanted to develop a spiking neural network from scratch in order to reach a deeper understanding of its theoretical basis. Going through the underlying mathematical equations (see chapter 3), their complexity was somewhat remarkable. Although they are not very mathematically advanced, the total quantity of computations is large: The number of sums to be added, equations to be computed (some of which are recursive) and loops to be run through to compute a single weight change is worth not-

in this case.

ing. When in turn this is done for every weight and every time step (not even mentioning the multiple synapses between any pair of connected neurons), it is natural to ask what the consequences will be regarding efficiency.

Some answers to these questions were found during the experiments. As mentioned in section 6.2.3, training a network with one hidden layer on a very limited input set with a modest number of iterations took more than an hour. In contrast, we trained a conventional neural network based on the backpropagation algorithm. Training this network on a list of 2000 words through 5000 iterations (as opposed to the 25 words and 300 iterations in the SNN experiment) took then only 0.86 seconds. Other experiments have confirmed this huge difference of efficiency between the two approaches. A lot could be said about the elegant solutions of the backpropagation algorithm, but again, that is not our current focus. Still, we note that our presuppositions about rather heavy computations from going through the theory were justified.

What then are the implications of this apparent lack of efficiency, and what can be done about it? Although not being our main focus, we have tried to look for ways to increase efficiency. Here we can point to the simplification we made in the formula for the Spike Response Model, explained in section 2.3 and implied in equation 2.5.

A more radical change could be to choose another neuron model. The Leaky Integrate-and-fire model is formally described as follows:

$$\tau_m \frac{\mathrm{d}u}{\mathrm{d}t} = -u(t) + RI(t) \tag{7.1}$$

In a model with discrete time steps, the neuron potential can then be described (as shown in [24]) by the following formula:

$$V_j(t) = V_j(t-1) + \sum_{i=0}^{N-1} x_i(t) s_i \tag{7.2}$$

We see that here, when calculating the potential for a given time step, we are adding the value for the previous time step. This is in contrast to the Spike Response Model we have been using, in which the potential is calculated independently of previous steps. In future works, it would be interesting to see if this model could simplify the computations and enhance the efficiency of the network.

7.3 The TrueNorth Architecture, Cognitive Computing and New Paradigms

We have to keep in mind that what we have done is a *simulation* of a spiking neural network. As we have seen, the biological neuron is the model and inspiration, but the reason that the brain has such a high degree of efficiency is that the neurons are processing physically independently of each other. A traditional computer is built on what is known as a *von Neumann architecture*. In these kinds of machines, there may be some parallel processing (depending upon the number of cores), but still the main way of processing is sequential in nature. This in turn leads to what has been called the *von Neumann bottleneck*: Even though processor speed and memory capacity both have increased substantially in later years, data still has to be passed from the processor to memory and back. Often they share the same bus for this data transportation, but even if they do not, a certain measure of latency is unavoidable. We therefore suggest that the real solution to the efficiency problem of spiking neural networks lies in the hardware. In the following, we will take a closer look at some of the later development in this area.

The field of cognitive computing has been rapidly evolving in the last few years. Trying to extensively define the term *cognitive computing* would be too ambitious within this thesis, but we can state that it involves self-learning systems that use data mining, pattern recognition and natural language processing, and to a great extent these systems mimic the way the human brain works. (The term *Brain-inspired computing* has also been used.) Now, we will not try to give a broad picture of what is happening in this field, but we will give a short presentation of a prominent cognitive computing system from IBM Research, namely the TrueNorth [24] system.

Our interest in this system is mainly based on the fact that at its heart, it is actually a spiking neural network. But then it is not only (as we have done so far) simulating it, the researchers have developed what they call a *neuromorphic* chip that constitutes the main building block of the system. This chip is built from a number of cores, in which each core has 256 input channels (parallel to the biological axons), a 256 x 256 synapse crossbar, and 256 digital neurons. An illustration of this chip is found in Figure 20. Information flows from axons to neurons gated by binary synapses, and all this information is in the form of spikes. As we remember from section 1.1.3, the biological spikes are electrical action potentials that are uniform in nature, and the information they transmit is actually contained in their time

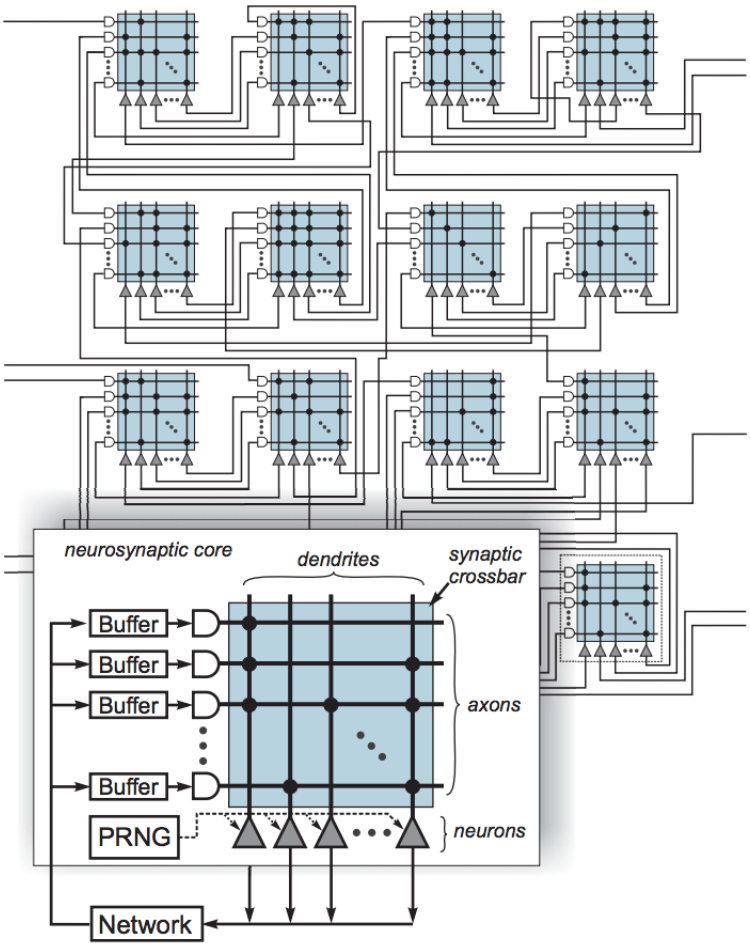

Figure 20: Illustration of the neuromorphic chip. Figure from [24]

of emission. Since we are now talking about hardware systems, it is possible to mimic this behaviour quite closely, using real electric spikes as the means of communication. Not only is the communication within the core consisting of spikes, but also the inter-core information passing, and even the input and output of the chip as a whole is in the form of spikes. The complete chip has (as of June 2015) 4096 cores, one million neurons, up to 256 million synapses and is made up of 5.4 billion transistors.

Along with the TrueNorth chip, the researchers also developed a simulator [25] (called "Compass") that is totally equivalent to the chip, running on the Sequoia Blue Gene supercomputer. This was undertaken in order to perform testing and algorithm development while the work on the hardware was progressing. It also allowed for comparing the speed and power consumptions of the two different approaches. The tests showed that the TrueNorth chip was about 1000 times faster, while consuming on the order of 400.000 times less energy than the simulated counterpart. Actually, its power consumption is as low as 73 milliwatts. Some of the reason why this is possible is that the neurons are implemented in an event-driven way, so that the neuron's active power usage is proportional to the number of spikes it is processing. Also, circuits that are currently not active can be "turned off", further reducing the need for power.

The IBM also states that the TrueNorth system can operate in real-time. Although the definition of the notion *real-time* varies, it has been described as one which "controls an environment by receiving data, processing them, and returning the results sufficiently quickly to affect the environment at that time". [26] Usually we talk about the millisecond level. It is not hard to see that this combination of speed and low power consumption opens a range of new application possibilities. Health devices is one field that is frequently mentioned in this respect. Here processing should ideally happen at the same speed as in the brain, and especially if it is an implanted device, it should not need to be charged too often.

7.3.1 A New Paradigm?

It has been claimed that the development of the TrueNorth and similar systems is not only a natural extension of the continuous evolution that has taken place since the dawn of the computer age: It is actually more a *paradigm shift*. Again using the brain as an analogy: Until now, computers have mainly been von Neumann-machines, and the programming languages

have been appropriate for the logic and arithmetic operations constituting the basis for the development within programming. This can be thought of as equivalent to the left-hand side of the brain, which is known to be the center of logic and rationality. Cognitive computing, on the other hand, will then be equivalent to the right-hand side of the brain, governing creativity, emotions, patterns and visual symbols.

So in what sense, and to what degree, can this really be called a paradigm shift? The phrase *paradigm shift* was coined by the American philosopher of science and physicist Thomas Kuhn, and is used to describe fundamental changes in the basic concepts, as well as the experimental practices, of a scientific discipline [27]. But this is generally also taken to mean that the previous concepts and practices to a great extent are discarded as obsolete. Then, can this be the case when it comes to cognitive computing? It seems clear that the answer to this question is negative. Sequential computing has of course not been proven wrong or invalid in any way, but more importantly, there is also no reason to believe that cognitive computing will make it superfluous. Although TrueNorth and its peers are Turing-complete and as such in principle can do everything a conventional computer can do, the latter ones do still have a (relative) simplicity, accuracy and logic power that will not be outdated. But then again, cognitive computing does to a great extent represent a fundamental change in basic concepts and practices, and in that sense it is obviously not far from being a paradigm shift. Still, we find it more natural to emphasize it as a complementary system. Going back to the brain analogy: Even if you learn to exploit the potential of the right-hand side of your brain, it still does not mean that you will discard the left-hand side that you've always been using.

7.4 Conclusion

We have been investigating the latest generation of artificial spiking networks, which most commonly is known as Spiking Neural Networks. We started out by examining the biological inspiration of these systems: Neurons, synapses, neuronal dynamics and the theories regarding spike coding. We have also looked at the different aims and motivations for the current development, and we found that it is probable that both computer science and the field of neurobiology will benefit from the research being carried out. We have briefly been going through the history and evolutions of artificial neural networks, stating that there are three main stages to be found. Then we have in some

detail described the Spike Response Model, since this serves as the basis of our further work. The problem of spike coding in SNN's has also been treated.

We went on to present a detailed derivation of a learning algorithm, first for a spiking neural network without hidden layers, then for a network with one or more hidden layers. In this we used an existing theoretical model, but we made some modifications of our own.

We chose to develop a SNN from the very ground, using this learning algorithm. With this we carried out experiments, from the simple boolean functions to more complex generated spike trains, with and without noise. We found that a one-layer network performed reasonably well on classifying these patterns, with gradually decreasing performance in the presence of noise. But proceeding to at two-layer network, we found that although the network error function was minimized to a large degree, the output results were levelled out so that they were not useful for classification purposes. This became evident when we tried to classify the XOR function, known as the most basic function that is not linearly separable. We have spent quite some time searching for the reason for this behaviour, but we have not really succeeded in explaining it. We also had to realize that this fact would have a serious impact on our further experiments with multi-layered networks.

We wanted to make a comparison between the performances of a conventional artificial neural network and a SNN. For this purpose we chose the problem of hyphenation of words of the Norwegian language. Here we found that both for one-layered and multi-layered networks the conventional networks performed better than their spiking counterparts. This conclusion was of course not unexpected after the problems we had already encountered, as described above. During the experiments, we also noted that our SNN had an issue with efficiency. Other matters that were discussed were the notion of linearity with regard to hyphenation, and a concrete example of the later development when it comes to hardware implementation of a spiking neural network. We also gave a short evaluation of the claim that cognitive computing represents a paradigm shift in computer science.

7.5 Future Directions

Looking at the future, it seems natural to start with the problems that remain unsolved in this thesis, notably the lack of classification results for a multi-layered SNN. We suspect that the main problem lies in the implementation

rather than in the mathematical derivations of the algorithm.

Then we would also like to investigate different mathematical approaches to derive learning algorithms, as we find that there seems to be no general consensus, but rather a large degree of diversity and different opinions in the body of literature in this field. A deeper understanding of the mathematical and theoretical background will also give us a better base for our development work.

As noted, we have not been able to use data that are strictly temporal of nature, like speech, music and motion recordings. In the future we would also like to experiment with this kind of data, using a system based on spiking neural networks.

References

[1] http://www.odec.ca/projects/2010/sambxj2/Nervous_System.html

[2] https://www.cs.mcgill.ca/~rwest/link-suggestion/wpcd_2008-09_augmented/wp/c/Chemical_synapse.htm

[3] http://philschatz.com/anatomy-book/contents/m46526.html

[4] Wulfram Gerstner and Werner Kistler: *Spiking Neuron Models: Single neurons, populations, plasticity.* Cambridge University Press, 2002.

[5] Wulfram Gerstner: *Time structure of the activity in neural network models.* Phys. Rev. E 51, 1995 738758.

[6] Edgar D. Adrian: *The Basis of Sensation.* Hafner, London, 1928.

[7] Donald O. Hebb: *The Organization of Behavior.* Wiley and Sons, New York, 1949.

[8] Thorpe, S., Fize, D., and Marlot, C.: *Speed of processing in the human visual system.* Nature, 381:520 522.

[9] Olaf Booji: *Temporal Pattern Classification using Spiking Neural Networks.* Master's thesis, Universiteit van Amsterdam, 2004

[10] Wolfgang Maass: *Networks of Spiking Neurons: The Third Generation of Neural network Models.* Technische Universitt Graz, 1997

[11] https://en.wikipedia.org/wiki/Activation_function

[12] Sander M. Bohte et al.: *Error-backpropagation in temporally encoded networks of spiking neurons.* Neurocomputing 48, 2002

[13] https://www.researchgate.net/publication259972016_Image_Processing_with_Spiking_Neuron_Networks

[14] Xin-She Yang *Artificial Intelligence, Evolutionary Computing and Metaheuristics.* P. 532, Springer Berlin Heidelberg, 2013

[15] https://github.com/ketilyv/SNN_Project

[16] Anwani, N. and Rajendran, B.: *NormAD Normalized Approximate Descent based Supervised Learning Rule for Spiking Neurons.* Proceedings of International Joint Conference on Neural Networks (IJCNN), 2015

[17] Ponulak, F.: *Analysis of the ReSuMe Learning Process For Spiking Neural Networks* International Journal of Applied Mathematics and Computer Science 18(2):117-127, 2008

[18] J. J. Hopfield: *Neural networks and physical systems with emergent collective computational abilities.* Proceedings of the National Academy of Sciences of the USA, vol. 79 no. 8 pp. 25542558, April 1982

[19] Rumelhart, David E., Hinton, Geoffrey E. and Williams, Ronald J.: *Learning representations by back-propagating errors.* Nature 323: 533536, 1986

[20] `https://en.wikipedia.org/wiki/Human_Brain_Project`

[21] Minsky, Marvin and Papert, Seymour: *An Introduction to Computational Geometry.* MIT Press, 1969.

[22] Liang, F.G.: *Word Hyphenation by Computer.* PhD Thesis, Stanford University, August 1983.

[23] Kristensen, Terje: *A neural network approach to hyphenating Norwegian.* Proceedings of the International Joint Conference on Neural Networks (IJCNN), volume 2. IEEE, 2000

[24] A. S. Cassidy et al.: *Cognitive computing building block: A versatile and efficient digital neuron model for neurosynaptic cores.* International Joint Conference on Neural Networks (IJCNN). IEEE, 2013.

[25] T. M. Wong et al.: 10^{14}. IBM Research Division, Research Report RJ10502, 2012

[26] Martin, James: *Programming Real-time Computer Systems.* Englewood Cliffs, NJ: Prentice-Hall Inc, 1965

[27] Kuhn, Thomas: *The Structure of Scientific Revolutions.* University of Chicago Press, 1962.

[28] Jason M. Allred and Kaushik Roy *Unsupervised Incremental STDP Learning Using Forced Firing of Dormant or Idle Neurons.* (Not yet published.)

[29] McCulloch, W. and Pitts, W.: *A logical calculus of the ideas immanent in nervous activity.* Bulletin of Mathematical Biophysics, 5:115133, 1943

[30] Kasinski, A. and Ponulak, F.: *Comparison of suoervised learning methods for spike time coding in spiking neural networks.* Int. Journal of Applied Mathematics and Compututer Science, Vol. 16, No. 1, 101113, 2006

List of Figures

List of Tables