Temporal Pattern Classification using Spiking Neural Networks

Olaf Booij

UNIVERSITEIT VAN AMSTERDAM

Temporal Pattern Classification using Spiking Neural Networks

Master's thesis Artificial Intelligence,
specialization Intelligent Autonomous Systems.

Under supervision of
Hieu Tat Nguyen
and
Marcel Worring

August 2004

by
Olaf Booij

Intelligent Sensory Information Systems
Informatics Institute
Faculty of Science
Universiteit van Amsterdam

A novel supervised learning-rule is derived for Spiking Neural Networks (SNNs) using the gradient descent method, which can be applied on networks with a multi-layered architecture. All existing learning-rules for SNNs limit the spiking neurons to fire only once. Our algorithm however is specially designed to cope with neurons that fire multiple spikes, taking full advantage of the capabilities of spiking neurons. SNNs are well-suited for the processing of temporal data, because of their dynamic nature, and with our learning rule they can now be used for classification tasks on temporal patterns. We show this by successfully applying the algorithm on a task of lipreading, which involves the classification of video-fragments of spoken words. We also show that the computational power of a one-layered SNN is even greater than was assumed, by showing that it can compute the Exclusive-OR function, as opposed to conventional neural networks.

keywords: *spiking neural networks, temporal pattern recognition, classification, gradient descent.*

Contents

1 Introduction

...
- Hello and welcome to the NS information-center. On which day do you want to travel ?
- Tomorrow.
- So you want to travel tomorrow ?
- Yes, that's right.
- At what time do you want to depart ?
- That doesn't matter.
- So you want to depart at 5 o'clock in the afternoon ?
- Eh... well, it doesn't matter.
- I didn't understand you. Please answer with yes or no. Again: so you want to depart at 5 o'clock in the afternoon ?
- Eh, yes.
- I didn't understand you. Please answer with yes or no. Again: so you want to depart at 5 o'clock in the afternoon ?
- Nevermind.
  <click>...

Although this telephone conversation seems pretty awkward, it is being held hundreds of times a day. We quickly realize that one of the persons is not a real human being but a computer, programmed to ask a predefined set of questions[1]. Once again it becomes clear that artificial intelligence has not accomplished what was envisioned 40 years ago.

It was expected that by now robots could interact with humans in a dynamic environment, without the requirement of a keyboard. The fact is that computers still have a very hard time dealing with continuous streams of data in an intelligent manner. Although a lot of artificial intelligence techniques have been developed for the processing of numeric and symbolic data [13], there are very few intelligent techniques to cope with multimedia and the performance is poor. We will focus on multimedia that are time-varying, such as speech and video signals, while there is a growing demand for all sorts of automated systems that can process this type of data In the example above we have already seen that a computer had to perform a simple speech-recognition task, transforming an audio signal into one of a limited set of words. Besides systems for audio-recognition there is

[1] A conversation with the speaking computer of the Dutch public transport information center (Reisinformatiegroep); translated from dutch.

also a growing need for tools that can process video-data, such as detecting criminal behavior with surveillance cameras or automated lipreading systems. Furthermore we can think of applications such as robot-control, weather-prediction and scene-segmentation in films. In all these tasks the notion of time is important and the processing system has to cope with this characteristic.

It is common practice to accomplish these tasks using the same technology that is used for the processing of static data. In numerous speech-recognition systems and other signal-processing systems the dynamic data is represented spatially; that is, every time-step is represented by a different element of the pattern-vector, which is fed to a static recognizer, as if it was a 1-dimensional picture [53]. There are a few undesirable side effects when using this technique [14]. First, the duration of the input-signal is fixed by the size of the pattern-vector, while most signals that need to be compared differ in length. Another problem is that the signal should be buffered before it can be processed by the recognition system. In case of processing of audio-signals this only results in a delay of the output. When this method is used for video-recognition, the extra problem of memory-storage becomes significant, while the input then becomes a 3-dimensional pattern, two for the pixel-location plus one for the location in time. A lot of systems performing object recognition in video therefore discard the time aspect altogether. For example, face-recognition in video is often done by trying to recognize the face in each single video-frame using the same routines as for image-recognition [25].

Besides the practical disadvantages of applying methods designed for static data to time-varying data, there are also some more theoretical drawbacks. The problem is that the time domain is clearly different than the spatial domain. The most particular feature of temporal data is causality: an event can only cause an effect in the future, not in the past. This phenomenon is omnipresent in all types of temporal data. For example, sounds, and especially musical tones, can usually be characterized by a sharp onset followed by a relatively long gradual decay. This clearly reflects the causal effect of the production of a tone at a certain moment of time, which results in a sound in the time-period following it until it fades out. There is no equivalent of this phenomenon in the spatial domain. For a better comprehension of dynamic data it is thus important to use a method that does not discard these characteristics of temporal data.

There is actually only one technique that really tries to capture the temporal structure of dynamic data called the Hidden Markov Model (HMM), which can learn to model a set of signals from example patterns [46]. HMMs are very popular due to their success in the field of speech-recognition and are thereby the dominating method in all studies in which temporal data has to be processed. But the application of HMMs is not without criticism. A big problem is that they were originally designed for discrete time data streams and discrete input-variables, while a lot of real-world problems take place in continuous time and require the processing of continuous variables such as sound-volume and light-intensity. It would be advantageous to have a computational tool that has less trouble in dealing with continuous values and could cope with tasks that occur in continuous time.

A well established computational tool for dealing with real-valued data is the method of Neural Networks (NNs), also called Parallel Distributed Processing systems (PDPs), which are based on the biological neural network of vertebrates [7]. This method has some interesting characteristics, that distinguishes it from other statistical methods. When an NN learns a certain task it distributes the desired process over a large number of very simple processing units. By doing this a network is created that is very fault-tolerant, even when some of the processing units are removed; it is said that NNs degrade gracefully. A negative aspect of this procedure is that it is hard to say how NNs solve a certain task. A bigger problem with regard to temporal patterns is that practically all types of NN-models are designed to process static data, so it is troublesome to apply NNs on dynamic domains.

In the last ten years the aim of neural network research has shifted to models that are more biologically plausible, that is, they mimic biological networks more closely. One of these models is called the Spiking Neuron Network (SNN) and we will use this model throughout our study [32, 18, 4]. An important feature of these SNNs is that they model the propagation of the biological network through time and thus they are by themselves continuous dynamic systems. This makes them especially suited for the application on temporal patterns and could serve as an alternative to the HMM method. However, SNN theory lacks an important component: there is no practical way to learn the model to process temporal patterns. The presently available learning algorithms for SNNs are all designed for static data and thus limit the representational power of the dynamic neuron-models [5, 50]. What is needed is a learning-rule that changes the free parameters in the SNN-model so to learn the appropriate output to a set of dynamic inputs.

1.1 Problem description

In this study we will try to develop a learning algorithm for spiking neural networks, that makes it possible for a network to learn to perform classification tasks on temporal data. The trained network should produce an output that represents the category the input-signal belongs to. So the architecture of the neural network must be constructed in such a way that it can receive input and produce an output. A network that has this property is the well-known layered feedforward construction. We will use this architecture, although the resulting learning algorithm could be applied to a much more general set of spiking networks.

The learning algorithm will be required to train the network in order to learn a specific task using example pairs of input and output. After it has been trained the network should be able to reproduce the correct output to a given training input, but also generalize over the training-data to produce a correct answer to input it has not seen before, such as noisy variants of the training-data. This method of learning through experience is a very common procedure in neural networks and other computational techniques and is referred to as supervised learning [2].

To evaluate the capabilities and the performance of the learning algorithm, it must be applied to both artificially generated tasks and real-world problems. The data that is involved in theses tasks must be transformed in such a way that it can be fed to the learning algorithm. For artificial data this will not be a big problem, because we can generate the required data ourselves. For a real-world task however this will be an important issue, not only because it is harder to deal with real-world data, but also because the performance of the algorithm will partially depend on the way the data is fed to it.

Although the goal is to develop a system that can learn to classify temporal data and thus provide an alternative to the Hidden Markov Models, we will not make an elaborate comparison between the two techniques. Because both techniques need specific preprocessing of the data, such a comparison would be too involved for this study. Another problem in comparing computational techniques is that the existing computational tools are fine-tuned to handle a certain task. For example, the systems that are used to perform speech-recognition have HMMs at the heart, but in addition all kinds of techniques specially designed for HMMs are used to boost the performance. It is of course impossible to design an algorithm from scratch that can beat such an expert-system. It will take a lot of research before a speech-recognition system using SNNs can be build that can be compared with the present systems. In this study the focus will be on developing a generic tool that can be used for all kinds of domains without the requirement of incorporating prior knowledge into the system. Thus the user does not have to bother with the technical details of the recognition system, which is one of the virtues of neural network tools.

Because SNNs model biological neural networks more closely, a great deal of research is being invested to also model the learning process of humans [17, 33]. We will however take no effort in making our learning-rule biological plausible. Our research will only aim at developing a working learning-rule that is computational efficient. As there is not much known about the way humans learn, the algorithm could also be of interest for the neuro-science community.

1.2 Organization of this thesis

The thesis is organized as follows.

First in chapter 2 spiking neural networks will be introduced including a mathematical description of the specific model we will be using. A lot of terminology, partly originating from the field of neuro-biology, introduced in this chapter will be used throughout the thesis. Also the architecture of the network we will use will be described.

In chapter 3 the actual learning algorithm will be derived. We will first restrict ourselves to an SNN architecture consisting of only an input- and an output-layer. Later in the thesis we will extend this to a more general network with more layers. The algorithm will be analyzed and tested on a simple artificial benchmark.

In Chapter 4 we will extend the developed learning-rule so it can also be applied to network architectures of more layers. It will be shown that although the learning algorithm without the extension is very powerful it can not compute all possible classification tasks. The extended algorithm will be tested on data-sets that were impossible to classify.

Then, in chapter 5, the learning algorithm will be used to build a lipreading system, which will be applied to a lipreading task. By doing this it will be shown that spiking neural networks can be used for very practical purposes. It will also be a good test-case for our algorithm.

Finally, in chapter 6, we will draw conclusions of our research given the results of the experiments. Furthermore we will indicate possible improvements to the algorithm and generally to the research of SNNs.

2 Introduction to Spiking Neural Networks

All organisms live in an dynamic environment and to be able to interact with it intelligently, some have evolved a special organ: the brain. This complex organ can deal with a large quantity of streams of input-information in a highly efficient manner. In the field of Artificial Intelligence (AI) we want to imitate such a machine so it can be used for all kinds of processing tasks. Looking more closely at the brain, we see that it consists of a large number of nerve-cells, more specifically called neurons. These neurons form connections with each other to compose a network, hence the name neural network. The study of artificial neural networks, a sub-field of AI, tries to model these biological neurons and by forming networks with these model-neurons wants to achieve the same processing-qualities as the brain. The model we will use is called the Spiking Neuron model, which is recently gaining much interest in in the field of artificial neural networks.

In this chapter we introduce these neural networks and describe the way we model them. We will first explain how biological neurons work in section 2.1. Then, section 2.2 describes the different types of models of these neurons. Section 2.3 concentrates on the Spiking Neuron Model, that we will use, and gives a formal definition. In 2.4 we show the general architecture of the network that is used. The possible coding-schemes for transforming an input-signal so it can be fed to a neural network is described in 2.5. Finally in 2.6 we will give an overview of the application of spiking networks.

2.1 Biological neurons

In order to imitate biological neural networks as the brain it is important to understand biological neurons, the building blocks of neural networks. Also, most of the terminology used in artificial neural networks originated from their biological counterparts. In this section we will introduce biological neurons and explain how they work.

Neurons are the actual processing units of the brain. The computation they do is very simple and compared to silicon-chips very slow. A network of a large quantity of these simple units however proves to be very powerful. A silicon-based computer usually has only one processing unit, while in a neural network all the neurons work in parallel. To form this network every neuron is connected with on average thousands of other neurons.

Although there are all kinds of different neurons the basic structure is the same. The cell-body, or the soma, of the neuron has many fine branched fibers, called dendrites, and one or more axons that extends away from the cell to other neurons and branches at the end, see figure 2.1. The basic operation of neurons is as follows.

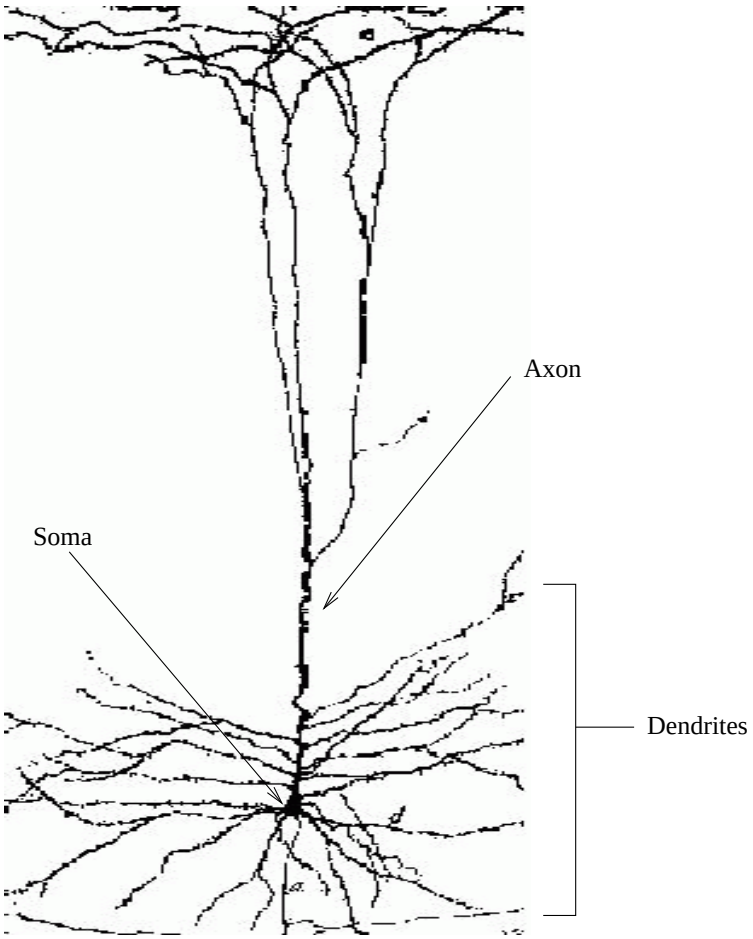


Figure 2.1: Drawing of a human neuron. The triangle-shaped soma is located at the bottom with the dendritic tree around it. The axon extends upwards and branches in on the top. Drawing of Ramon y Cajal.

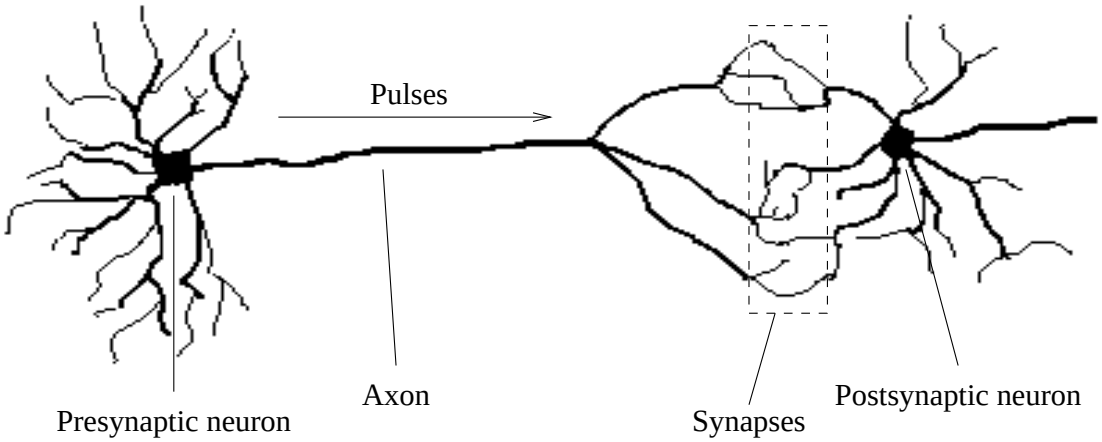


Figure 2.2: Drawing of two connected biological neurons. When the presynaptic neuron fires it sends a pulse along its axon to the synapses. At the synapses the pulse causes a change in the potential of the postsynaptic neuron.

Neurons have a small negative electrical charge of -70 mV, their resting potential. Certain stimuli from other neurons can cause the potential in the cell to rise. When the potential reaches a threshold, typically around -55 mV, the neuron fires an electrical pulse along its axon, also called a spike. At the end of the axon the axon-branches are connected to dendrites of other neurons. This connection is called a synapse, see figure 2.2. When a spike reaches such a synapse it causes a change of potential in the dendrites of the receiving neuron. This process is relatively slow, so the effect is delayed with a certain time typical for that synapse. The neuron that sends the spike is called the presynaptic neuron and the one that receives it the postsynaptic neuron. Depending on the type of synapse this change can be positive, raising the postsynaptic neurons potential, or negative, lowering it. When it raises the potential and thereby can cause the neuron to fire, the synapse is called excitory. When it lowers the postsynaptic neurons potential, making it harder for that neuron to fire, the synapse is called inhibitory. The effect of the potential-change is only temporal; after a while it fades away because the neuron will always try to stay at its resting potential. After a neuron fired the spike it needs some time to recover, before it is able to spike again. This time interval is called the refractory period.

The type of the synapse, inhibitory or excitory, can never change, but the intensity of the potential-change it causes can. This effect called synaptic plasticity, enables the network to learn from past experience. Bioneurological research has already produced lots of information on when and how these synapses change [38, 48]. This knowledge however only concerns isolated neurons, not a bigger network. We still do not know how biological neural networks like our brain learn.

As said, this is the basic operation, but there are a lot of different types of neurons and synapses. Some types of neurons have very large axons, so they can influence other

brain-regions. Other neurons only compute locally, having both a short axon and short dendrites. Some neuron-types only develop inhibitory synapses, others merely excitory. Axons do not always form synapses with dendrites. Some form synapses with the cell-body of an other neuron, so it has a bigger influence on it. Some even form inhibitory synapses with other axons so to prevent that axon from propagating its spike.

A biological neural network is always built out of a mixture of these neuron-types. There is no part in the human brain consisting of a homogeneous pool of one type of neurons. These networks are highly recurrent; that is, there are a lot of loops present in the network that facilitate positive and negative feedback.

It is clear that a single biological neuron is a very complex dynamic system. It would be very hard to imitate such a neuron in all its detail, let alone imitating a highly recurrent network of a heterogeneous pool of different neuron-types. Maybe we do not have to be so accurate. A crude model could already exhibit some of the qualities of the brain.

2.2 Neuron models

In this section we will give a brief overview of models which simulate biological neurons. In particular the sigmoidal neuron and the spiking neuron will be described and the distinction between them.

One model is not necessarily better then another, but it is more a difference in the level of abstraction. Some models try to simulate the neuron very accurately, taking all the different biochemicals into account [37, 22]. Usually the aim of such models is not to build a neural network, but to see what computations are possible with one single neuron. Other models are much more abstract and do not describe the state of a neuron in terms of molecules, but just by a real number, called its activation [49, 51]. With these kind of models it is much easier to make a network and to figure out how to make them learn something.

By far the most popular neuron-model is the sigmoidal unit [51], see figure 2.3. In this model the output or activation of a neuron is modeled by a single variable, usually between 0 and 1. The synapse between two neurons is modeled by a weight-variable that describes the strength of the impact on the postsynaptic neuron. These weights do not have to be positive, but also can be negative modeling an inhibitory synapse. The sigmoidal neuron sums up all the weighted firing-rates of its presynaptic neurons to get its potential. From this potential the activation is computed using an activation function. This activation function is a sigmoid function, hence the name sigmoidal neuron [7].

The activation-variable in this model can be seen as the rate with which the neuron fires its spikes; that is, the number of spikes in a certain time window. It was generally believed that this was the only information that was passed between two biological neurons [47]. The so called neural-code of the neural-network is the firing rate. In recent years it is argued that the firing rate can not be the only neural-code. Psychological experiments have pointed out that some neural processing is too fast for this kind of computation

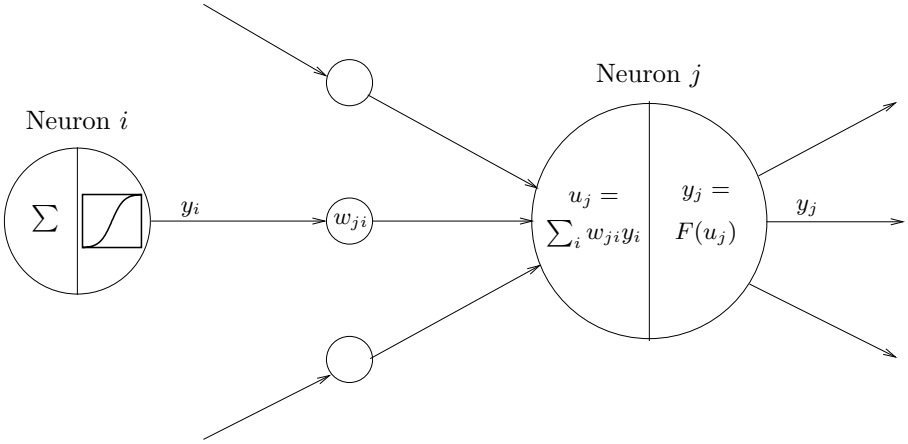


Figure 2.3: The basic operation-steps of a sigmoidal unit are summation of its input and calculating its activation. The output of neuron i is weighted by the synapse connecting the two neurons giving the weighted input $w_{ji}y_i$. This is summed for all inputs-neurons of neuron j to get the potential u_j. Using the sigmoidal function $F(\cdot)$ the activation y_j is calculated. The form of this function is shown in neuron i.

[58]. The neurons could not have calculated the average number of spikes in such a short time. There also have been numerous neurobiological studies that point towards another kind of neural-code [3, 47], that of precisely timed spikes: the information that is sent from neuron to neuron is not coded in the firing rate of the spikes , but in the precise timing of the spikes. This explains the high speed of neural processing.

Spiking neural networks (SNN) (also pulse-coupled or integrate-and-fire networks) are more detailed models and use this neural-code of precisely timed spikes [32, 18, 4, 29]. The input and output of a spiking neuron is described by a series of firing-times, called a spike-train, see figure 2.4. One firing-time thus describes the time a neuron has sent out a pulse. Further details of the pulse like the form are neglected, because all pulses of one neuron-type look alike.

The potential of a spiking neuron is modeled by a dynamic variable and works as a leaky integrator of the incoming spikes: newer spikes contribute more to the potential then older spikes. If this sum is higher than a predefined threshold the neuron fires a spike. Also the refractory period and synaptic delay is modeled.

This makes an SNN a dynamic system, in contrast with the sigmoidal neuron networks which are static, and enables it to perform computation on temporal patterns in a very natural way.

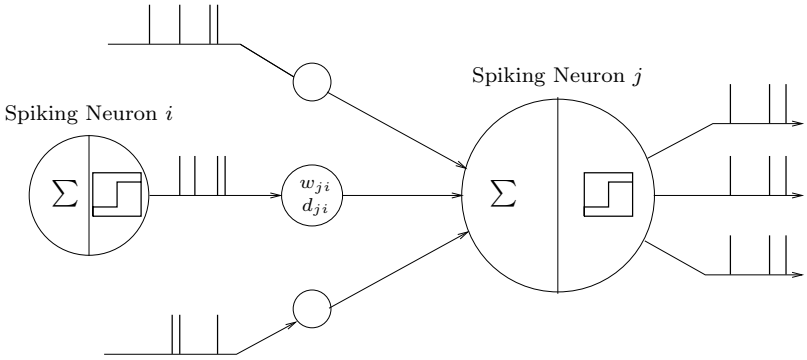


Figure 2.4: The input and output of a spiking neuron are series of firing-times called spike-trains. The firing-times are represented by vertical bars.

2.3 Spike Response Model

In this section we will present the spiking neuron model we will use, called the Spike Response Model (SRM) [16]. The SRM formally describes how the incoming spike-trains are processed to produce a new spike-train leaving the neuron.

The state of spiking neuron j in the SRM is described by its potential $u_j(t)$. When this potential crosses a certain constant threshold-value ϑ the neuron fires a spike, that is describes by its spike-time $t^{(f)}$. We will use a threshold-value ϑ of 1 for all our simulations.

The output of neuron j is thus fully characterized by the array of spike-times:

$$\mathcal{F}_j = \{t_j^{(f)}; 1 \leq f \leq n\} = \{t | u_j(t) = \vartheta\}, \tag{2.1}$$

where n denotes the number of spikes. The spike-train $\mathcal{F}_j$ is chronologically ordered; so, if $1 \leq f < g \leq n$, then $t_j^{(f)} < t_j^{(g)}$.

The potential of a neuron can change due to spikes of its presynaptic neurons $i \in \Gamma_j$, where

$$\Gamma_j = \{i | i \text{ is presynaptic to } j\}. \tag{2.2}$$

If presynaptic neuron i has fired a spike at time $t_i^{(g)} \in \mathcal{F}_i$ the potential of postsynaptic neuron j at time t is raised by $w_{ji}\varepsilon(t - t_i^{(g)} - d_{ji})$. The variable w_{ji} denotes the weight of the connection and d_{ji} denotes the delay of the connection. The spike response function ε describes the effect the presynaptic spike has on the potential of the postsynaptic neuron. Different mathematical formulations are possible but the shape of the function is always a short rising part followed by a long decaying part and $\varepsilon(s) = 0$ for $s \leq 0$ to insure causality, see figure 2.5.

The formula we will use is the difference of two exponential decays [16]:

$$\varepsilon(s) = \left[\exp\left(-\frac{s}{\tau_m}\right) - \exp\left(-\frac{s}{\tau_s}\right)\right] \mathcal{H}(s), \tag{2.3}$$

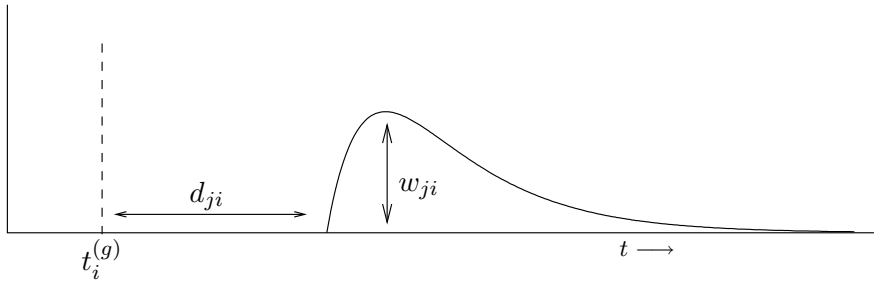


Figure 2.5: The change in the potential of postsynaptic neuron j, given a spike of presynaptic neuron i is given by $w_{ji}\varepsilon(t - t_i^{(g)} - d_{ji})$. At time $t_i^{(g)}$ the presynaptic neuron i fires a spike, shown in the figure with the dashed vertical line. After the delay of the connection, d_{ji}, the spike has an effect on neuron j scaled by the weight w_{ji}. First the potential of postsynaptic neuron j increases fast, followed by a long decay until the spike has no influence anymore.

where $\mathcal{H}(s)$ denotes the Heavy-side step function: $\mathcal{H}(s) = 0$ for $s \leq 0$ and $\mathcal{H}(s) = 1$ for $s > 0$. The two time-constants τ_m and τ_s (with $0 < \tau_s < \tau_m$) control the steepness of the rise and the decay of the function and more importantly they control the position of the top of the function. In all simulations we used $\tau_m = 4.0$ and $\tau_s = 2.0$.

Another possible formulation of the spike response function, for example used in [43] and [5], is given by:

$$\varepsilon(s) = \frac{s}{\tau} \exp\left(1 - \frac{s}{\tau}\right) \mathcal{H}(s), \tag{2.4}$$

where τ is a time constant controlling the rise- and decay-time. Although this function looks very different, it behaves almost completely the same. If τ is set to 2.7, the top of the function will be at approximately the same position as the ε-function we use.

Another process that changes the potential of a neuron is the refractoriness. This is modeled by the refractory function η. If neuron j emitted a spike at $t_j^{(f)}$ its potential at t is lowered with $\eta(t - t_j^{(f)})$. Again the Simple Response Model does not fix this function, but to insure causality it is required that $\eta(s) = 0$ for $s \leq 0$ and it is usually non-positive, see figure 2.6. We use a simple exponential decay:

$$\eta(s) = -\vartheta \exp\left(-\frac{s}{\tau_r}\right) \mathcal{H}(s), \tag{2.5}$$

where ϑ is the threshold of the neuron, $\mathcal{H}(s)$ is the Heavy-side step function and τ_r is another time-constant. We used $\tau_r = 20.0$ in all our simulations.

The formula of the potential of a neuron, $u_j(t)$, is the sum of the influence of the presynaptic spikes and of its own spikes:

$$u_j(t) = \sum_{t_j^{(f)} \in \mathcal{F}_j} \eta(t - t_j^{(f)}) + \sum_{i \in \Gamma_j} \sum_{t_i^{(g)} \in \mathcal{F}_i} w_{ji}\varepsilon(t - t_i^{(g)} - d_{ji}) \tag{2.6}$$

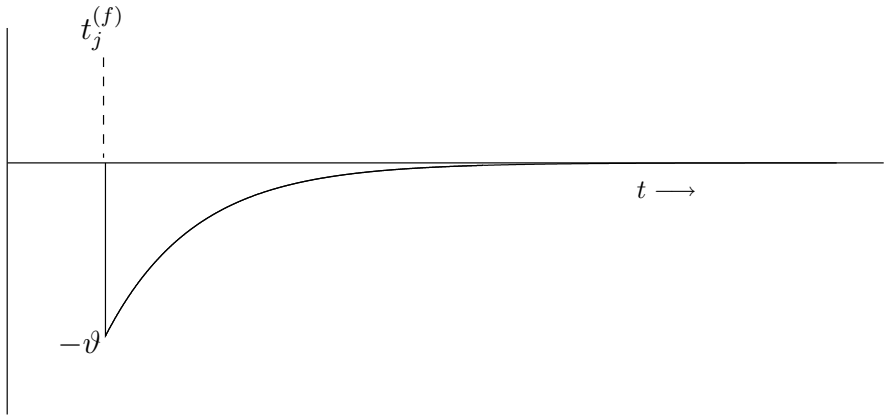


Figure 2.6: The change in the potential of neuron j given a spike is given by $\eta(t-t_j^{(f)})$. At time $t_j^{(f)}$ neuron j fires a spike, shown in the figure with the dashed vertical line. Immediately after the spike the potential of postsynaptic neuron j drops to $-\vartheta$ and then slowly recovers, followed by a long decay until the spike has no influence anymore.

Equations 2.1 and 2.6 define the Spike Response Model and together with equations 2.3 and 2.5 they fully describe the behavior of one single neuron. A simple example illustrating the process is shown in figure 2.7, where the spikes of two neurons cause a third neuron two fire.

2.4 Network architecture

Like with conventional neurons various networks can be built with spiking neurons, depending on its purpose [35, 43, 12]. Our goal is to build a classifier and for this a layered feedforward architecture proved to be successful for conventional neurons [2].

A layered feedforward network consists of layers of neurons which only form connections with neurons in subsequent layers, see figure 2.8. The first layer, called the input-layer, acts as the input of the network. Actually this layer does not consist of neurons, because there is no real processing involved: the so called input-neurons are forced to have a certain output. In our case of spiking neurons the input-neurons fire a predefined spike-train. The last layer is the output-layer. The spike-trains of these neurons form the output of the network. In between the input- and output-layer there could be any number of hidden layers. When learning the neural network these hidden layers form their own representation of the input that is sometimes necessary for learning a classification task.

The set of connections between two layers of neurons can also be seen as a layer. Because these layers of connections are a very important concept in neural network theory, it is the convention to denote layered network by the number of these layers and not by the

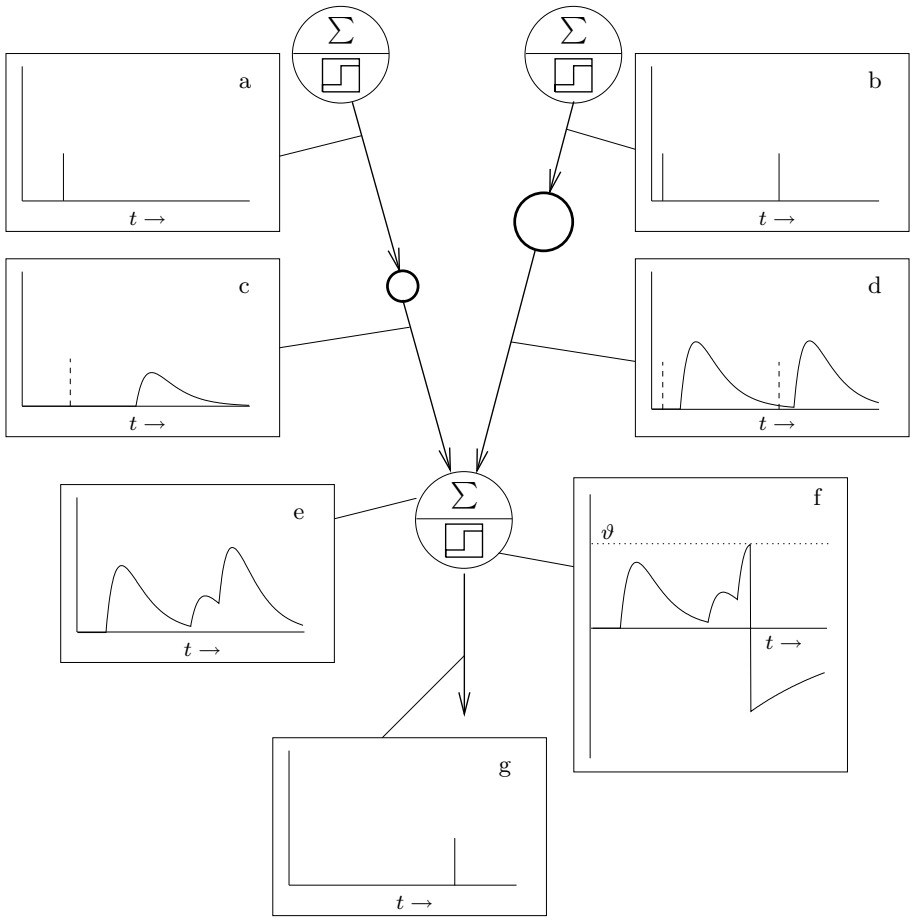


Figure 2.7: A graphical example explaining the workings of a spiking neuron. a, b: the two top neurons fire spikes, the left fires once, the right fires two times. c, d: the synapses delay and weigh the spikes and then transform them into spike responses using the ε-function; the left synapse has a larger delay then the right synapse, but a smaller weight. e: the incoming spike responses are summed to give the neurons potential. f: the potential is thresholded and because the potential crosses the threshold, the potential drops due to the refractoriness. g: the output of the neuron is a spike at the time the potential reached the threshold.

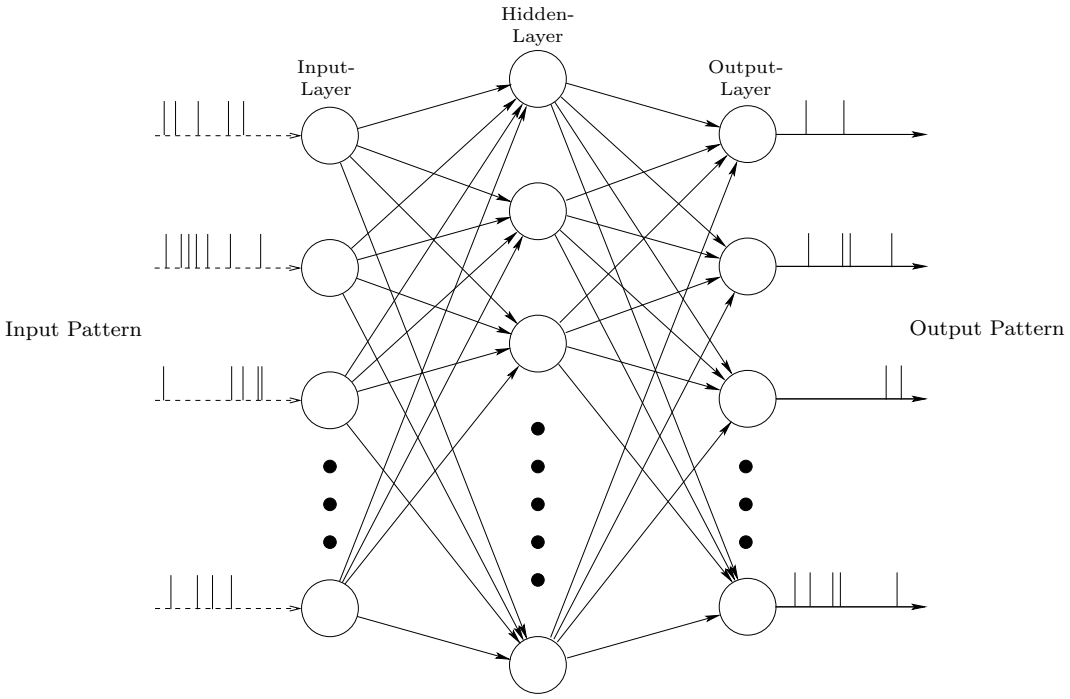


Figure 2.8: A feedforward network with one hidden layer. The neurons of the input-layer are forced to fire a certain spike-pattern shown with the dashed arrows. The spike-trains of the output-neurons form the output of the network.

number of layers of neurons [2]. For example, a network with one hidden-layers is called a 2-layered network.

We use multiple synapses per connection, as introduced in [17] and used in [43] and [5]. Every synapse has an adjustable weight and a different delay, see figure 2.9. These different delays provide a way for the presynaptic neuron to influence the postsynaptic neuron on a longer time-scale than the time-interval of the spike-response and on a more detailed level.

All connections consist of a fixed set of l delays: $D = d^k; 1 \leq k \leq l$, so we can drop the indication of the presynaptic and postsynaptic neurons. The corresponding weight is denoted with w_{ji}^k. Equation 2.6 can thus be rewritten as:

$$u_j(t) = \sum_{t_j^{(f)} \in \mathcal{F}_j} \eta(t - t_j^{(f)}) + \sum_{i \in \Gamma_j} \sum_{t_i^{(g)} \in \mathcal{F}_i} \sum_{k=1}^{l} w_{ji}^k \varepsilon(t - t_i^{(g)} - d^k) \tag{2.7}$$

With the right learning algorithm this network should be able to learn pairs of input- and output-patterns; it would be trained to produce the output belonging to a given input. Using spiking neurons the input- and output-patterns consist of a set of spike-trains, see

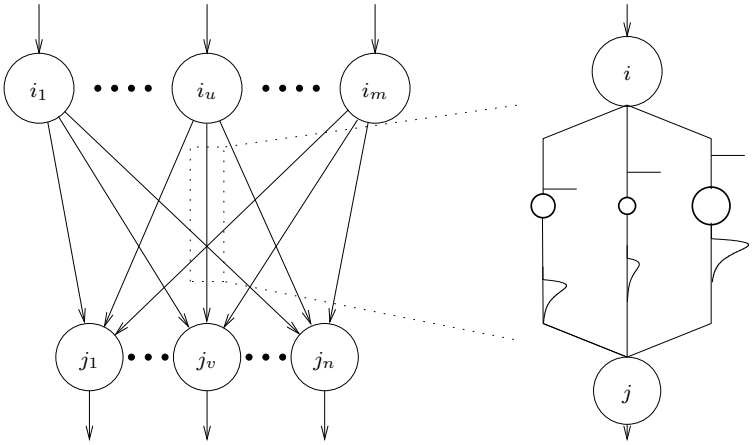


Figure 2.9: One connection between two neurons consists of multiple delayed synapses which all have an adjustable weight.

also figure 2.8. The input-pattern is fed to the input-layer and the network processes this to give a certain output-pattern. The learning algorithm then has to change the weights of the synapses in such a manner to minimize the difference between the given output and the desired output-pattern. The network could also be learned by changing the delay-times of the synapses [36]. As this is still a bit unusual in neural network theory, we chose to keep them fixed.

2.5 Spike coding

In this section we will describe the methods for encoding an analog signal into spike-trains.

Spiking neural networks use sets of spike-trains as their input and output, but usually we want to do computations on analog data or streams of data. In order to use SNNs we somehow have to encode this analog data in spike-trains in order to feed them to the network. A lot of research has been done to figure out what kind of coding biological neurons use to represent analog data [47, 1, 16, 52]. Very few of the proposed representations can be used in the network architecture as the one we presented. We will describe these, as well as some other representations that originate from the field of computer science.

Maybe the simplest way to convert an analog value to spike code is to use only one spike with its firing-time proportional to the analog value. This is called time-to-first-spike coding [16] and is used in a lot of studies [31, 43, 4, 59]. The drawback of this coding is that every spiking neuron fires only once and thus limits its capabilities.

A stream of analog values can be encoded in this way by sequentially encoding one analog value in one spike-time during a certain time-period and combining these time-periods

one after each other. This kind of coding is called phase coding [20, 16]. It suffers from the same shortcomings as the time-to-first-spike coding: the expressive power of the spike-train is limited because there is only one spike in a certain period.

Another simple way to encode a stream of analog values is by thresholding it, reducing it to a stream of bits, that can be seen as a spike-train [47]. The thresholding causes a large loss of data, reducing every value to a bit, and thereby a lot of useful information could be thrown away. However this dimension-reduction could also be an advantage, because it results in compact spike-trains. The thresholding can also be done with spiking neurons, producing more "natural" spike-trains.

Data from one variable can also be scattered over more than one neuron. This is generally called population coding [32], but there are various methods to do this. In [4] for example an analog value is transformed into a couple of spike-times each belonging to a different neuron using receptive fields. Every neuron overlaps the analog variable with a Gaussian-kernel, which has a specific mean and variance. The hight of the Gaussian-kernel at the value of the analog variable determines the firing-time of the neuron. [44, 4]

When coding a sound-signal it is customary to code the different frequency-responses with different neurons [23]. In [55] the fast Fourier transform was used to convert sound samples into 128 different frequency-signals. Every frequency-signal is then thresholded using a spiking neuron to produce a spike-train.

It is difficult to say which method is good and which is not, but some seem more suited for certain applications then others. If only a couple of values should be coded and these values have a high information density, then the receptive field method is probably a good choose, because it distributes the information over a pool of neurons. If however the input consists of thousands of streams of data like a video sample, it is better to threshold the values in order to reduce the number of neurons and spikes.

2.6 Applications using Spiking Neural Networks

In this section we will give an overview of the aims of studies in SNN research and will briefly describe the achievements of some of these studies.

The research field of spiking neural networks is comparable with that of traditional sigmoidal neural networks and can be divided in the same sub-goals [51]:

- Auto-associator. The network is fed with a training-set of patterns, which it is supposed to store by tuning its free parameters. When presented with a new pattern it should reproduce the most similar training-pattern.
- Pattern-association. The network is fed with a training-set of pairs of patterns and the network should learn the generalized mapping between the input- and output-patterns. When presented with a new input-pattern it should produce an output-pattern that is consistent with this mapping.

- Classification. This can be seen as sub-task of the pattern-associator; the required output-pattern consists of the predefined category the input-pattern belongs to.
- Clustering. For this task the classification of the training-patterns is not known a priori, but the network should discover certain salient features with which it is able to divide the data in different classes.

Because of the dynamic character of SNNs, much of the focus in research points towards computations with temporal patterns, such as speech-recognition and time-series-prediction. Also this dynamic feature can be used to solve the notorious binding-problem: the encoding and detection of conjunctions of features [4].

In [5] a multi-layer network architecture, as described in section 2.4, with an accompanying learning-rule called SpikeProp is used, which can classify sets of static data. Experiments on real-world data-sets produce good results, comparable with those of the conventional backpropagation algorithms. A classification method for dynamic data is presented in [54]. The specific task in this study is predicting the value of a certain dynamic set of variables given the history of this variable. However the method could also be used for general classification-tasks on temporal patterns.

There are also a lot of studies trying to develop unsupervised learning rules that can be used to cluster data using SNNs. In [43] an algorithm is introduced that can cluster static and dynamic data. An example of clustering static data is described in [6], where an image is sub-divided in areas with the same color. An example of clustering temporal patterns is given in [55], where a network learns to discriminate two different audio-samples.

In addition to feedforward architectures there are studies concerning totally connected SNNs in analogue with the conventional Hopfield network [35, 41]. Like their conventional counterpart these networks can serve as auto-associators or content-addressable memories.

Of course it is good to see that SNNs can perform the same tasks as conventional networks. It would be even better if they could be used for applications where other algorithms experience difficulty. We therefore think SNNs can make a real difference in processing temporal patterns, because of its dynamic nature.

3 Learning algorithm for one-layered SNNs

In the previous chapter we have introduced the architecture of a basic spiking neural network that is capable of mapping sets of input spike-trains to sets of output spike-trains. What we want to do is to make a classifier using this network, that is able to classify temporal patterns. In order to do this we need a learning algorithm that can change the weights of the synapses in such a way that the network can classify a certain pattern of input spike-trains. Although there have been various studies introducing supervised learning-rules designed for spiking neural networks [17, 50, 5], most of them only consider spiking neurons that spike once. Some studies have proposed methods to learn SNNs that use multiple spikes per neuron, but none of them is practical; that is, they use awkward methods such as expanding the network with each input-spike it receives, leading to a problem with memory-storage [54].

In this chapter we will develop such a learning-rule. In section 3.1 we will derive a learning-rule that can tune the weights given an input-pattern and the desired output. The parameters of the learning-rule and the network architecture that have to be set before we can learn a specific tasks are discussed in section 3.2. The SpikeProp algorithm of Bohte [5] has close resemblance with our learning-rule. In section 3.3 we will compare the learning algorithms with each other. Finally section 3.4 will describe a simple benchmark test to show what the learning-rule is capable of.

3.1 Derivation of the learning-rule

In this section we will formally derive a learning-rule that can determine the weights of a spiking neural network without hidden layers so that it can successfully classify spike-train patterns.

In order to find these weights, they are sequentially tuned to minimize a certain error-measure. This minimization is accomplished by the gradient descent method; that is, descending the error landscape proportionally to the derivative of the error. This kind of parameter-optimization is an important tool in the field of learning algorithms [51, 2].

The error-measure is determined by the mismatch between the desired output of the output layer and the actual output. So it is important to decide what kind of neural coding the output-neurons should have.

For classification-tasks we do not need a large representational power, because the output should only provide the class the input-pattern belongs to and the number of classes is typically small. We choose to code this in the time of the first spike $t_j^{(1)}$, so we ignore

all the spikes that come later. The error of the network can then be determined by the sum of the squared differences of the desired spike-time and actual spike-time:

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

where $\hat{t}_j^{(1)}$ denotes the desired first spike-time of neuron j and J denotes the output layer.

It would be more difficult to find an adequate error-function if we would demand more then one spike per output-neuron, because it would be difficult to determine which actual output spike corresponds with which desired spike.

In order to minimize the network-error we should change each weight proportionally to the derivative of the error with respect to this weight. The weight-change for a synapse from neuron i to neuron j is thus denoted by:

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

where η is a small constant called the learning rate and w_{ji}^k is the weight of the synapse from input neuron i to output neuron j with a delay of d^k. Because the weight w_{ji}^k only influences the spike-times of output-neuron j the chain rule can be used to expand the second factor of (3.2) to:

$$\frac{\partial E_{net}}{\partial w_{ji}^k} = \frac{\partial E_{net}}{\partial t_j^{(1)}} \frac{\partial t_j^{(1)}}{\partial w_{ji}^k}. \tag{3.3}$$

The first factor, that expresses how the error of the network changes given the first spike of output-neuron j is easy to compute:

$$\frac{\partial E_{net}}{\partial t_j^{(1)}} = t_j^{(1)} - \hat{t}_j^{(1)}. \tag{3.4}$$

Calculating the second factor of (3.3), that expresses the change in spike-time given a change of the weight, is more difficult, because there is no formula expressing the firing-time in the weight. In order to calculate it we have to realize that neuron j spiked at time $t_j^{(1)}$ because the potential u_j reached the threshold at that time (see equation (2.1)):

$$u_j(t_j^{(1)}) = \vartheta. \tag{3.5}$$

Since the potential is constant, that is ϑ, for every $t_j^{(1)}$ it holds that

$$du_j(t_j^{(1)}) = 0. \tag{3.6}$$

By expressing the firing-time $t_j^{(1)}$ in w_{ji}^k,

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

we can expand (3.6) to:

$$\frac{\partial u_j(t_j^{(1)})}{\partial w_{ji}^k} dw_{ji}^k + \frac{\partial u_j(t_j^{(1)})}{\partial t_j^{(1)}} \frac{\partial t_j^{(1)}}{\partial w_{ji}^k} dw_{ji}^k = 0. \tag{3.8}$$

Dividing by dw_{ji}^k we get:

$$\frac{\partial u_j(t_j^{(1)})}{\partial w_{ji}^k} + \frac{\partial u_j(t_j^{(1)})}{\partial t_j^{(1)}} \frac{\partial t_j^{(1)}}{\partial w_{ji}^k} = 0. \tag{3.9}$$

The last factor of the second term is the expression that we want to compute, so we only have to compute the other two terms. Remembering the formula for the potential of a neuron,

$$u_j(t) = \sum_{t_j^{(f)} \in \mathcal{F}_j} \eta(t - t_j^{(f)}) + \sum_{i \in \Gamma_j} \sum_{t_i^{(g)} \in \mathcal{F}_i} \sum_{k=1}^{l} w_{ji}^k \varepsilon(t - t_i^{(g)} - d^k), \tag{3.10}$$

we can easily calculate these derivatives. The first partial derivative of the potential with respect to the weight is given by:

$$\frac{\partial u_j(t_j^{(1)})}{\partial w_{ji}^k} = \sum_{t_i^{(g)} \in \mathcal{F}_i} \varepsilon(t_j^{(1)} - t_i^{(g)} - d^k). \tag{3.11}$$

For the partial derivative with respect to the first spike-time we do not have to worry about the refractoriness-term, because $\eta(s) = 0$ for $s \leq 0$ reflecting the fact that later spikes do not influence earlier spikes.

$$\frac{\partial u_j(t_j^{(1)})}{\partial t_j^{(1)}} = \sum_{i,k} \sum_{t_i^{(g)} \in \mathcal{F}_i} w_{ji}^k \varepsilon'(t_j^{(1)} - t_i^{(g)} - d^k). \tag{3.12}$$

Filling in equations (3.11) and (3.12) in (3.9) we can calculate $\frac{\partial t_j^{(1)}}{\partial w_{ji}^k}$:

$$\frac{\partial t_j^{(1)}}{\partial w_{ji}^k} = \frac{-\sum_{t_i^{(g)} \in \mathcal{F}_i} \varepsilon(t_j^{(1)} - t_i^{(g)} - d^k)}{\sum_{i,k} \sum_{t_i^{(g)} \in \mathcal{F}_i} w_{ji}^k \varepsilon'(t_j^{(1)} - t_i^{(g)} - d^k)} \tag{3.13}$$

Combining the results we can express the formula for the weight-change (3.2) in a concrete way:

$$\Delta w_{ji}^k = -\eta \frac{-\sum_{t_i^{(g)} \in \mathcal{F}_i} \varepsilon(t_j^{(1)} - t_i^{(g)} - d^k)}{\sum_{i,k} \sum_{t_i^{(g)} \in \mathcal{F}_i} w_{ji}^k \varepsilon'(t_j^{(1)} - t_i^{(g)} - d^k)} (t_j^{(1)} - \hat{t}_j^{(1)}). \tag{3.14}$$

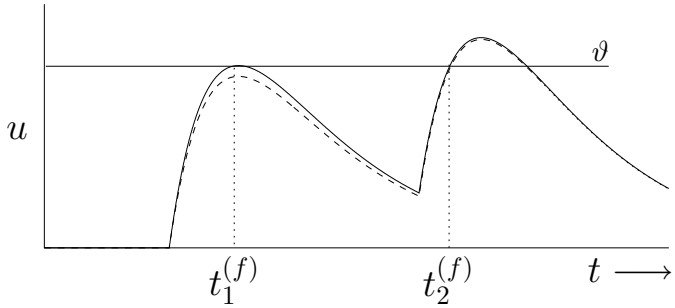


Figure 3.1: A small weight change can cause a jump in the firing-time of the first spike. The two potentials in this graph are almost the same. Except that the dashed potential is somewhat lower in the first part caused by a lower weight and does not reach the threshold until $t_2^{(f)}$. The solid potential however is a bit higher in the first part due to a small positive weight-change and reaches the threshold a lot earlier at $t_1^{(f)}$.

With this learning-rule we should find a minimum in the error-landscape. We must however be aware that this landscape is not continuous. A small change in a weight can cause a neurons potential to reach the threshold, where it did not reach it before the weight-change; see figure 3.1. If the resulting spike is the first spike, the error jumps up or down depending on the desired spike-time. So the partial derivative of the error with respect to this weight could be very large.

This can also be seen by looking closely at equation 3.14. The denominator of the fraction consists of the gradient of the potential of the neuron during the spike. If this gradient is very small, meaning that the threshold was barely reached by the potential, then the weight-change becomes very large. Because of the discretization during simulation this could even lead to a negative potential gradient, when the potential is immediately decreasing after it reached the threshold. This would lead to a very large weight-change in the wrong direction.

To circumvent this problem we put a lower bound on the gradient of the potential while calculating the weight-change. If the gradient turns out to be less then a certain value, the weight-change is calculated using the bound-value instead of the real gradient. Such a bound could slow down the gradient descent method, but the difference would be minimal if the value of the bound is low. In all our simulations we used a bound of 0.1.

The algorithm for tuning the weights for one pattern is summarized in Algorithm 1. To learn a set of patterns this procedure must be repeated several times for each pattern.

Iteratively tuning the weights using this learning algorithm would bring us to a minimum in the error-landscape. The minimum would most likely not be a global minimum, because the gradient descent method does not converge to the optimal parameter settings. This should however not be considered as a drawback, because finding parameters that would classify all the training-patterns correct, would in general mean that we have

Algorithm 1 Learning algorithm for a 1-layered SNN

```
for all output-neurons j in J do
    compute partial derivative of network with respect to j according to (3.4)
    compute gradient of potential (the denominator of 3.13)
    if gradient < 0.1 then
        gradient ← 0.1
    end if
    for all input-neurons i in I do
        for all weights w_ji^k of the connections form i to j do
            compute partial derivative of network with respect to w_ji^k using the numerator
            of (3.13) and the gradient
            compute weight-change Δw_ji^k using (3.2) and (3.3)
        end for
    end for
end for
for all weights w_ji^k do
    w_ji^k ← w_ji^k + Δw_ji^k
end for
```

overfitted the training data [2].

3.2 Parameter settings

Before training a network we have to set the parameters used by our learning procedure with sensible values. For some of these we can just choose a value that seems appropriate, without tuning them to improve the networks performance. For other parameters it is difficult to come up with a good value, so we have to make a rough estimation using some preliminary tests. We will discuss the parameters one by one and give empirical formulas where possible.

- The number of delays per connection. There are a number of synapses with different delays d^k with $k \in 1, 2, \ldots, l$ between every input- and output-neuron (as described in section 2.4). The delay-interval $(d^l - d^1)$ should strongly depend on the duration of the input-pattern and the desired output-spikes. The smallest delay should be the difference between the time of the last input-spike and the earliest output-spike and the largest delay should be the difference between the time of the earliest input-spike and the last output-spike. By doing this, every input-spike can be delayed in such a degree that it can influence the desired early and late output-spikes. In all the experiments we chose to distribute the delays 1 ms apart from each other, as in [5, 43], so that $d^k = k$ ms with $k \in \{1, 2, \ldots, l\}$.
- Weight-initialization. Before training begins the weights should be initialized to some random value. If the weights are too low the output-neurons will not fire

and there is no way to calculate the error with respect to the weights. So it is important to make a good estimate. We chose to pick these initial weight-values randomly from a uniform distribution. The average of this distribution depends on a number of things including the Fan-In of the output-neurons, which is the number of synapses leading to one output-neuron. In our architecture this is the product of the number of input-neurons $\mathcal{I}$ and the number of delay lines per connection k: Fan-In $= \mathcal{I}k$. The formula for a good estimation of the average weight-value is given by:

$$E(w) = \frac{\vartheta}{\text{Fan-In } \bar{\mathcal{N}}_i \int \varepsilon(s)\, ds}, \tag{3.15}$$

where $\bar{\mathcal{N}}_i$ denotes the average number of input-spikes per input-neuron and $\int \varepsilon(s)\, ds$ compounds to $\tau_m - \tau_s$.

We will take the width of the distribution to be somewhat bigger then the average, so that some weights will be initialized with negative values.

- The learning-rate η. It is difficult to pick a suitable value for the learning-rate η, introduced in equation (3.2). If η is too high the algorithm will most likely not converge, but will change the weights with such big steps that it will overshoot the minimum and start oscillating around it. If it is too low the algorithm will change the weights too slow and thus it will take a long time before the network-error will reach a minimum. Like the initial weight-values, η also depends on the Fan-In of the output-neurons. A rough estimation is given by:

$$\eta = \frac{\vartheta}{\text{Fan-In } \bar{\mathcal{N}}_i \int \varepsilon(s)\, ds} 0.01 \tag{3.16}$$

- The stopping-criteria. When dealing with real-world data there is always presence of noise. So it would be very unlikely that the algorithm finds weight-values that reduces the network-error to zero. This would also be infeasible, because it would indicate that we have overfitted the data. It is better to stop training when the error drops beneath a certain threshold. The order of the error, and thus the height of the threshold, depends on a number of things: the number of training-patterns, the number of output-neurons and the coding-scheme of the output:

$$\text{ErrorThreshold } = \frac{\mathcal{P}\mathcal{J}\text{Var}(t_j^{(1)})}{3}, \tag{3.17}$$

where $\mathcal{P}$ denotes the number of training-patterns, $\mathcal{J}$ the number of output-neurons and $\text{Var}(t_j^{(1)})$ the variance of the desired output-spikes.

The given formulas should be used as a guide to find good parameter values, but are by no means optimal.

3.3 Related work

In this section we compare our learning-rule with SpikeProp, a learning-rule derived by Bohte in [5]. It must be noted that SpikeProp is meant for a network architecture with hidden layers. We shall generalize our learning-rule for these networks in chapter 4, but we can already compare the basic features.

The SpikeProp algorithm was designed for classifying spike-times using the same network-architecture. As in our approach, a gradient descent method was used for finding a learning-rule. Thus the learning-rules are basically the same, although the mathematical derivation is different.

The main difference is that SpikeProp is only capable of learning one spike per input-neuron, while our rule is intended for input-neurons that fire more than once. The patterns that SpikeProp could classify are sets of real values, coded in the spike-time of each input-neuron. The domain of classification tasks is thus the same as that for conventional neural networks using conventional backpropagation.

Our aim is at developing a learning algorithm that can classify temporal patterns. Of course temporal patterns can be coded as a set of real values; for example in [34] sound-fragments of spoken words coded in sets of 40 individual spike-times are classified using a spiking neural network. The conversion of dynamic data to static values does not seem to be appropriate. The strength of spiking neural networks is that they can process dynamic data without encoding it in static values.

The experiments conducted in [5] indicated that the time-constants of the spike-response function $\varepsilon(s)$, see equation (2.3) and (2.4), should be large, otherwise the SpikeProp learning algorithm will not converge. To be more specific, the rising part of the spike-response function should take longer then the time interval in which the input-patterns are encoded, see figure 3.2. In our algorithm we use a $\varepsilon(s)$-function with its top at $s = 4 * \ln(2) \approx 2.8$ ms, which is much smaller than the input time-interval. Nevertheless we did not experience any problem. We suspect it has something to do with the fact that if the gradient of the potential is close to zero while crossing the threshold, the weight-change becomes very large, as explained in section 3.1. In the case of very large time-constants this effect does not occur because the spike-response function only rises. Another solution is discarding very small gradient values when computing the weight-change, as we did.

The SpikeProp algorithm was tested on a number of real-world benchmark tests that are frequently used in the field of static pattern recognition. The results are comparable with those of conventional backpropagation networks, although the computational costs do differ. Despite the fact that SpikeProp uses less training-cycles, it needs more CPU-time, because each connection consists of a number of synapses that need to be tuned.

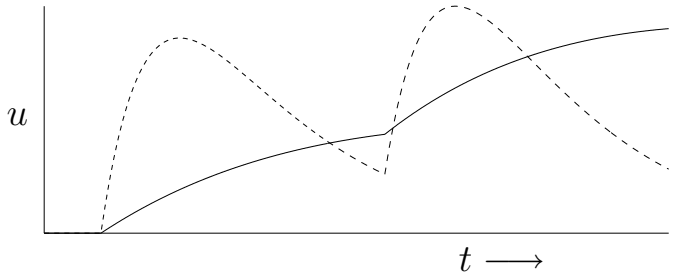


Figure 3.2: The effect of varying the time-constants of the spike-response-function $\varepsilon(s)$ on the potential-change due to two incoming spikes. The potential of a neuron with small time-constants, represented with a dashed line, clearly reflects that the incoming spikes only have a temporal effect and this effect is highly non-linear. However, the solid line that reflects the potential with large time-constants has a long during effect on the potential and the potential increases almost linearly with time.

3.4 Simple benchmark problem

Before doing experiments on real world problems, we will first test the algorithm on artificially generated data. The network has to learn to classify different patterns and will be evaluated on generalization and robustness. To produce random input-patterns we use homogeneous Poisson processes. Because Poisson processes simulate the occurrences of rare random events in time, like spikes in biological neurons, they are very suitable for simulating spike-trains [47, 19] and are used in a lot of SNN research [21, 43, 42].

3.4.1 Task description

The task consists of learning to classify a set of input-patterns in one of 4 categories. We first generate one random input-pattern for each of the 4 categories. Because we want to test the robustness of the learning-algorithm, we make 10 noisy variants of these patterns to present to the network. The resulting 40 input-patterns are then split to produce a training-set and a test-set in order to evaluate the generalization capabilities of the algorithm.

The spiking neural network has an input-layer of 10 neurons, so the input-patterns are composed of 10 independent random spike-trains, one for each input-neuron (see figure 3.3).

For the spike-train a Poisson process of 16 milliseconds is used with a constant rate of 0.2 spikes per millisecond:

$$S = \{t | x[t] \leq 0.2\}, \tag{3.18}$$

where $x[t]$ is a sequence of random numbers uniformly distributed between 0 and 1 (see [19] for a detailed description of simulating spike-trains with Poisson processes). The

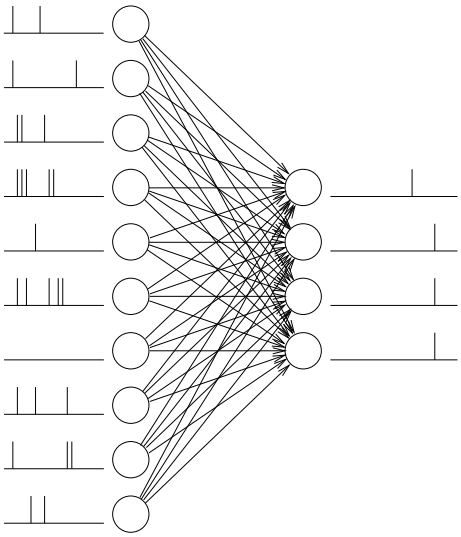


Figure 3.3: The Poisson spike-train benchmark. An input-pattern consists of a set of 10 Poisson spike-trains that are fed to the 10 input-neurons. The desired output is an early spike for that output-neuron representing the right classification of the input-pattern and a late spike for all other output-neurons. In this case the input-pattern belongs to the first class.

resulting spike-train S thus consists of on average 3.2 spikes with independent firing times.

Noisy versions of these spike-trains are made by shifting each spike by a random amount taken from a normal distribution with a standard deviation of 4.0 milliseconds and zero mean.

The output of the network is represented by 4 output-neurons, one for each class. In the training procedure we desire that the output-neuron that represents the class of the input-pattern fires an early spike, $\hat{t} = 17$, while all other output-neurons should fire a late spike, $\hat{t} = 22$ (also see figure 3.3). During the testing-procedure we do not evaluate the specific firing-times of the output-neurons. We will only look at the order in which the neurons fired. The first output-neuron that fires a spike wins, so the networks classification is the class to which that neuron belongs.

3.4.2 Results

The Poisson-benchmark was tested 10 times using the network architecture depicted in figure 3.3. We used $l = 20$ different delays per connection with delay-times of 1 ms, 2 ms, ..., 20 ms. The weights were randomly initialized between $[-0.01, 0.1]$ and the learning rate η was set to 10^{-4}. Training was stopped when the sum squared error became smaller then 100 (ms^2), an average of 1.25 squared error with respect to every

output-spike.

The network had no problems with the task. During the 10 tests it needed on average 14.4 cycles with a variance of 0.68 to learn the training-set before the stopping-criterion was met. During all 10 experiments the network successfully classified the test-set, without making a single misclassification. So although there are a lot of free parameters that need to be tuned (10 input-neurons x 4 output-neurons x 20 synapses = 800 weights) and thus there is a high change of overfitting [2], the algorithm did generalize well.

The success on this artificially generated benchmark is of course very hope-giving. Nevertheless we must keep in mind that the Poisson spike-trains are ideal spike-trains for our algorithm. The spikes in one spike-train are completely independent of each other, so the information in one spike-train is nicely distributed over each individual spike-time. The algorithm uses this fact by weighing the effect of every input-spike on the error with the same amount. In real world problems it is very difficult to encode the input information in such a way that the firing-times are independent of each other, so the spikes of the input-neurons will most likely be highly correlated [47].

4 Multi-layered networks

In the previous chapters we developed a learning algorithm and showed that it can learn to classify temporal data. The algorithm can only learn spiking neural networks with one layer of adaptable weights. It is arguable that one-layered spiking neural networks can learn all possible mappings from input to output. In this chapter we extend the learning algorithm to learn networks with more layers.

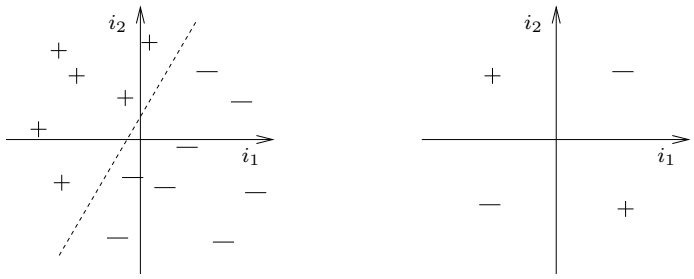
Section 4.1 explains why we need more layers of adaptable weights to classify certain pattern-sets. In section 4.2 we will extend our learning-rule for network architectures with more layers. In section 4.3 the new learning algorithm is tested on classification tasks that could not be learned with our previous learning algorithm.

4.1 Necessity of more layers

This section explains why the learning algorithm using a one-layered SNN is insufficient for learning all possible input-output mappings. The problem is that a one-layered SNN may not be able to represent all possible mappings. We will first review the little information that is already known about the representability of a one-layered network. We will expand this knowledge by investigating a specific mapping called the Exclusive-OR function. Then it will be shown empirically that the learning algorithm has difficulty learning this specific function and other more complicated functions.

4.1.1 Background

The computational power of a networks of spiking neurons is, as yet, poorly understood. It is known however that a spiking neuron using spike-time coding can simulate a conventional sigmoidal neuron. So SNNs have at least the same power as sigmoidal neural networks [30]. It is also known that conventional neural networks with one layer of adaptable weights can only classify input-patterns that are linearly separable [2]. Linearly separable means that hyperplanes can be put in the input-space that separates patterns of different classes, see figure 4.1(a). So it follows that we can build a linear classifier using a one-layered network of spiking neurons. It is easy to see that if the number of input-patterns is smaller than the number of dimensions of the input-space, then the input-patterns are linearly separable [2]. When classifying video-data using pixel-values, for example in the lipreading-task in chapter 5, this will certainly be the case. However, we must be careful: although a pattern-set with a high dimensional



(a) (b)

Figure 4.1: Input-patterns of two categories (+ and -) in an input-space of two dimensions R^2. (a) A set of patterns that is linear separable. The dashed line shows a possible separation. (b) A very simple pattern-set, that is not linearly separable, called the Exclusive-OR problem. It is not possible to draw a line that separates the two categories.

input-space usually is linearly separable, the best generalization of the separation could be nonlinear. Besides that, there are other linear classifiers that work very efficient, so it is important to construct a system that can do more.

It turns out that spiking neurons can do more. In [30] it is proven that there is a function called *element distinctness* that can be computed with just one spiking neuron, but needs a hidden layer when computed with sigmoidal neurons. The *element distinctness* function ED_n, that checks if some of its n inputs is the same, is defined by:

$$ED_n(s_1, \ldots, s_n) = \begin{cases} 1, & \text{if } s_i = s_j \quad \text{for some } i \neq j \\ 0, & \text{if } |s_i - s_j| \geq 1 \quad \text{for all } i, j \text{ with } i \neq j \\ \text{arbitrary}, & \text{else.} \end{cases} \tag{4.1}$$

The input-variables $s_1, \ldots, s_n$ are encoded in the firing-times of the incoming spikes of the neuron and the binary output is encoded by the firing/non-firing of the neuron. For a complete analysis of the spiking and sigmoidal neurons computing this function see [31] or [29].

It is clear now that for certain non-linear classification tasks an SNN of only one layer is needed, whereas conventional sigmoidal neural networks need an additional hidden layer of neurons.

4.1.2 The Exclusive-OR problem

The most famous non-linear function in the field of neural networks is the Exclusive-OR problem (XOR) which maps two boolean input-variables two one boolean output-variable, see table 4.1 and figure 4.1(b) [2, 7]. Although this problem is purely theoretical,

Input 1	Input 2		Output
false	false	→	false
false	true	→	true
true	false	→	true
true	true	→	false

Table 4.1: The XOR-function, which maps two boolean values to one boolean value.

it is desirable for a general pattern-recognition technique to have the ability to solve it, because most real-world tasks imply this problem. Maybe the most comprehensible example of such is real-world task is the detection of a wink; a person is said to wink if it closes one and only one eye. So a "wink-detector" should react to a face that has only the left or only the right eye closed, but not if both are closed or both are open.

The seemingly simple XOR-function can not be computed by a one-layer network of sigmoidal neurons because of its non-linearity and thereby caused a great deal of disillusion in neural network research in the 70s. It is therefore interesting to question if a one-layered SNN can solve this problem. In the following we will show that this can indeed be done although the solution is not very neat.

First of all it is easy to see that the XOR-function is actually a special case of the *element distinctness*-function. For example, the boolean input-variables true and false can be coded by an input-spike at time $t = 0$ and $t = 6$ respectively. The output false is then coded by a spike, while the output true is coded by the absence of a spike. Given that a spiking neuron can solve the *element distinctness*-function, leads to the conclusion that it can also solve the XOR-function.

This solution however is not very elegant. The problem is that the coding of the output is not the same as the coding of the input. The input-variable is coded in the spike-time while the output is coded as firing/non-firing. This causes trouble if we want to use this output as the input for another spiking neuron. We could force the neuron to produce an output-spike at a much later time-step by introducing another input-spike that causes the potential of the neuron to rise above the threshold a while after the input-range, for example at time $t = 20$. By doing this the output is also coded in the spike-time of the neuron, namely: a spike before $t = 20$ is interpreted as false and a spike at $t = 20$ as true. This only partially solves the problem because the time-scale of the input and the output are different. Another method, described in [28], is to feed the output of the neuron to a so-called synchronizing module, which is composed of spiking neurons, that translates the firing/non-firing code into the spike-time coding on the right time-scale. Unfortunately this would change our one spiking neuron into a multi-layer network, so we can not use it.

A better encoding of the XOR-function would be to code both the input and the output in time-to-first-spike coding with equal time-scale for input and output. Such a coding is given by Bohte in [5], see table 4.2. Notice that a third input-neuron, a so-called

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

Table 4.2: Exclusive-OR encoded with one spike per input neuron as in [5]. Each row shows one input-output mapping. The input consists of 3 spike-times, one for each of the 3 input-neurons (Input1, Input2 and Bias) and the output consists of one early or late spike-time representing the class of the input.

Input		Output
0 5 10	→	20
0 5	→	15
0 10	→	15
0	→	20

Table 4.3: Exclusive-OR encoded in a single spike-train. The output is encoded in the same way as in table 4.2, but the input is encoded using 1, 2 or 3 input-spikes for one single input-neuron.

bias-neuron, should be used that always fires at $t = 0$, to specify the starting time of the problem. Of course there are more possible ways of coding the problem in precisely timed spike. In table 4.3, for example, an encoding is shown that represents the problem in only one spiking-train, using the fact that spiking neurons can fire more than once. We focus on the first encoding, because it is considered to be unsolvable for a one-layered SNN [5].

We will show that a one-layered SNN in fact can solve the XOR-problem as given in table 4.2 with an arbitrarily small error. What follows is a description of a specific network-architecture and an indication of the values for the delays and weights. Specific numerical values for the delays and weights are shown in table 4.4. We must point out that there are other values which also lead to a solution for the XOR-problem.

The architecture of the network is partially fixed by the problem: there are 3 input-neurons and 1 output-neuron. The number of synapses that run from each input-neuron to the output-neurons can be chosen. It turns out that 7 synapses in total are enough: three from the bias-neuron leading to the output-neuron and two from both other input-neurons, see figure 4.2 for the naming-convention. First we will set the delay and weight of a synapse from the bias-neuron, in such a manner that the output-neuron will always fire at $t = 16$, no matter what the spike-time of the other input-neurons are. The other two synapses from the bias must cause the potential of the output neuron to peak just under the threshold at $t = 10$, so only little extra potential rise is needed to produce an early output-spike. Now all we have to do is ensure that the output-neuron's

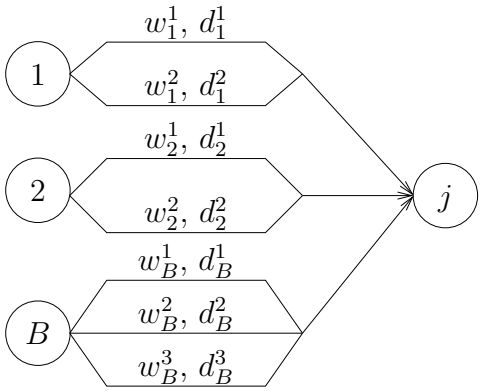


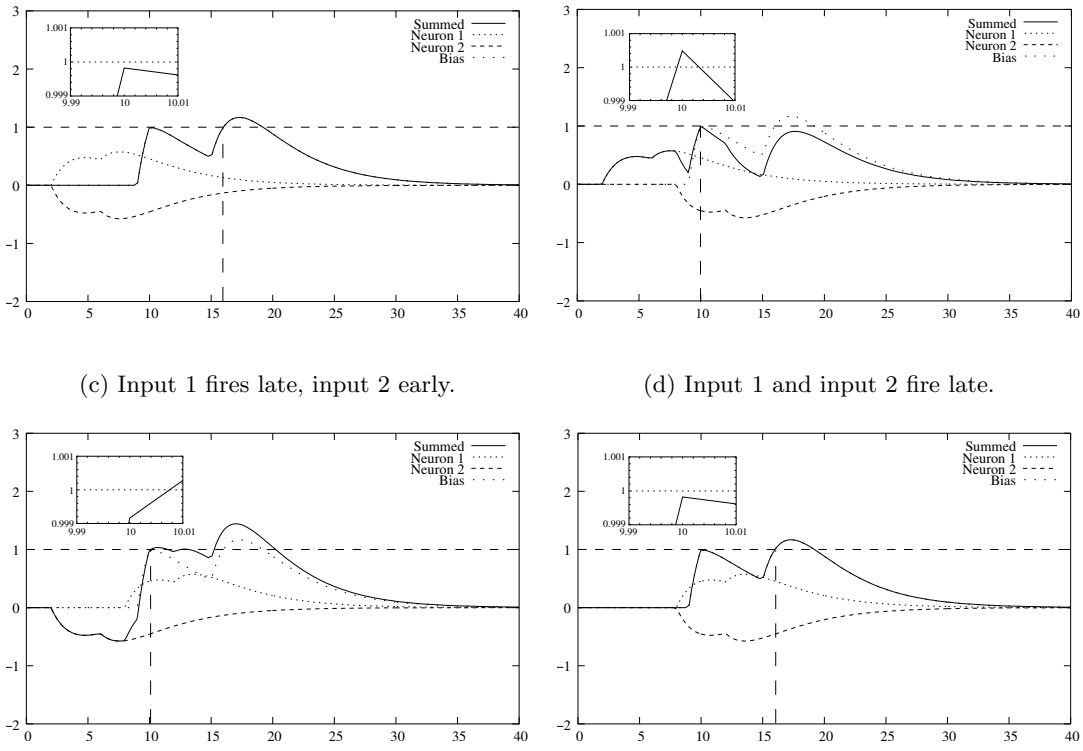
Figure 4.2: An SNN architecture that can solve the XOR-problem. The output-neuron has two synapses from input-neuron 1 and 2 and three from the bias input-neuron B. The weights w and the delays d are specified using the input-neuron as subscript and the delay-number as superscript.

potential is not or negatively changed around $t = 10$ when the first two input-neurons spike simultaneously and positively otherwise. By choosing the same delays for both neurons ($d_1^1 = d_2^1$, $d_1^2 = d_2^2$) but mirroring the weight-values ($w_1^1 = -w_2^1$, $w_1^2 = -w_2^2$), the net influence on the potential will be zero if the input-neurons spike simultaneously. It is impossible to cause an output-spike at exactly $t = 10$, when the input-neurons fire dissimilar. So it be approximated, by firing slightly before $t = 10$ for the first dissimilar pattern and slightly after for the other dissimilar pattern. This is done by choosing the weights and delays of the first neuron in such a manner that the influence on the potential after a delay of $t = 4$ is equal to that of $t = 10$, but at $t = 4$ the potential is rising while the potential is descending at $t = 10$. Because the weights from the other neuron are the same but negative, the influence of this neuron is the other way around: descending at a delay of $t = 4$ and rising at $t = 10$. If the first neuron fires at $t = 6$ and the second at $t = 0$ then both neurons produce a rising influence at $t = 10$ of the potential of the output-neuron, so that the potential is positively influenced just after $t = 10$. If on the other hand the first input-neuron fires an early spike while the second fires a late spike, then the output-neuron is positively influenced before $t = 10$. In both cases the potential of the output-neuron reaches the threshold around $t = 10$ and fires a spike.

Figure 4.3 shows the potential change of the output-neuron caused by the input-patterns. As can be seen the potential just stays under the threshold at $t = 10$ for the input-patterns that are supposed to let the neuron fire at $t = 16$ and it barely crosses the threshold at $t = 10$ for the other patterns. This is called a hair-trigger situation and should be avoided because the network will not be robust against noise [30]. If, for example, the two input-neurons do not fire at precisely the same time then the net influence on the potential of the output-neuron will not be zero and could result in an early output-spike.

(a) Input 1 and input 2 fire early

(b) Input 1 fires early, input 2 late.



(c) Input 1 fires late, input 2 early.

(d) Input 1 and input 2 fire late.

Figure 4.3: The influence of the 3 input-neurons on the potential-change of the output-neuron for all four XOR-patterns using the weight-values given in table 4.4. The solid line stands for the sum of the three contributions making up the potential of the output-neuron. The influence of the bias-neuron is the same for every pattern, because it always fires at $t = 0$. However, the influence of the other two neuron is shifted to the right if they fire a late spike. The dashed vertical line shows when the potential crosses the threshold. From the inset, which zooms in on the threshold at $t = 10$, it is clear that there is a hair-trigger situation.

Delay	Value	Weight	Value
d_1^1	1	w_1^1	1.906419
d_1^2	5	w_1^2	1
d_2^1	1	w_2^1	-1.906419
d_2^2	5	w_2^2	-1
d_B^1	8	w_B^1	5.8038
d_B^2	9	w_B^2	-2.6
d_B^3	14	w_B^3	3.59

Table 4.4: A set of delay and accompanying weight-values for the network-architecture depicted in figure 4.2, with which the XOR-problem can be solved.

In table 4.4 a set of delays with their weights is shown that computes the XOR-function with a sum squared error of 10^{-4}. This error could be made arbitrarily small by further fine-tuning of the weights and using smaller time-steps for the simulation. This also worsens the effect of the hair-triggering; that is, the difference between the potential and the threshold at $t = 10$ becomes smaller.

A spiking neural network can thus solve the classical XOR-problem with just one layer of adaptable weights with an arbitrarily small error. It can however only be done using a hair-trigger situation and this is not very feasible. It means that we can not use it on real-world problems that involve computing the XOR-function, because then there will be noise in the data.

4.1.3 Learning the XOR-function with one layer

It is important to know what kind of mappings SNNs can represent, but it is at least as important to understand what mappings the SNN is able to learn. Investigating the learnability of one spiking neuron analytically is very difficult [36], so hardly anything is known about the learnability of a network of spiking neurons. Therefore, we investigate the learnability experimentally, by trying to learn the XOR-problem using our learning-rule.

A network of 3 input-neurons each connected by 16 delay-lines with 1 output-neuron was trained on the data-set as given in table 4.2. The weights were initialized between -1 and 2 and a learning-rate of 10^{-4} was used. Over 10 trails the network reliably learned the XOR-function within a sum squared error of 1.0 (ms^2). For clarity, it must be stated that the found weight-configurations were different than that given in the 4.1.2. To reach a solution it needed on average more than 10^5 training-cycles, which is a long time, for example as compared with the 14.4 cycles used to learn the Poisson spike-trains in section 3.4. These long learning times suggest that the algorithm has trouble to converge because of the hair-trigger situation. To investigate if this was the case we tested the found classifiers on test-patterns that were noisy variations of the original training-patterns. Shifting the input-spikes by a random amount taken from a

normal distribution with a standard deviation of only 0.1 ms, caused the sum squared error of the output to increase 52.1 (ms^2). Clearly, the solutions that are found by the learning-algorithm are not noise-tolerant and thus indicate a hair-trigger situation.

4.1.4 Other non-linear functions

By showing that a one-layered SNN can solve the XOR-problem, it is not proven that it can solve all possible non-linear mappings. Although the XOR-function is a problem with a lot of historical significance in the field of neural networks, it is by no means a complex function.

An example of a somewhat more difficult non-linear problem is the Parity function, which is a generalization of the XOR-function with n input-variables:

$$P_n(b_1, \ldots, b_n) = \begin{cases} 1, & \text{if the number of } \{b_i | b_i = 1\} \text{ is odd} \\ 0, & \text{otherwise.} \end{cases} \tag{4.2}$$

We tried to find a solution for the case of $n = 3$ input-variables using a one-layered SNN and our learning algorithm with the same parameters as used for the XOR benchmark. However, the algorithm did not converge to a solution, but got stuck in a local minima with a sum squared error 20.

It is clear that our algorithm with a one-layered SNN has trouble learning all possible input-output mappings, although it is not proven that this is impossible. However, we can be sure that an SNN with more layers of adjustable weights can compute all classifications on single spikes, since two-layered sigmoidal networks are able to model any continuous function [2] and a sigmoidal neuron can be simulated by a spiking neuron using a single precisely timed spike [30], This brings us to the conclusion that it is important to generalize our learning algorithm in order to make it applicable to multi-layered networks.

4.2 Extending the learning-rule

In this section we derive a generalized version of our learning algorithm, so it is able to learn the weights of a spiking neural networks with more than one layer of adaptable weights. The derivation is comparable with that by Rumelhart, who generalized a learning-rule for one-layered sigmoidal networks to the well-known backpropagation-rule for multi-layered networks [38].

We focus on the case of a network with one hidden layer, but it is easy to see how this can be extended to networks with arbitrarily many layers. Figure 4.4 shows a very basic network to indicate the different variables that are used in a two-layered network. Both the weights of the connections from the input-layer H leading to the hidden-layer I and those from the hidden-layer I to the output-layer J should be tuned to minimize the

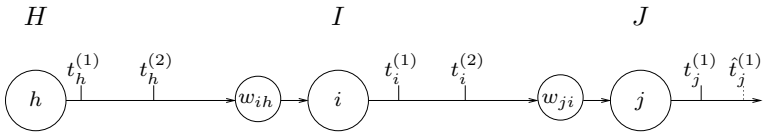


Figure 4.4: A very simple SNN with one hidden-layer to make clear what variables are involved.

network-error. The necessary weight-change for the weights w_{ji}^k leading to the output-neurons is equal to the weight-change of the one-layered network, equation (3.14). The rule for tuning the weights w_{ih}^k leading to hidden-neurons is derived in the same way, using the gradient descent method:

$$\Delta w_{ih}^k = -\eta \frac{\partial E_{net}}{\partial w_{ih}^k}, \tag{4.3}$$

Again the partial derivative of the network error with respect to the weight should be calculated, in the same way as was done with the weights leading to a output-neuron, see equation (3.3). In addition we must take into account that a hidden-neuron can fire more than once, in contrast with the output-spikes, where only the first spike influences the error. So the error depends on all spikes $t_i^{(g)}$ of the hidden-neuron, which are a function of the weight w_{ih}^k:

$$\frac{\partial E_{net}}{\partial w_{ih}^k} = \sum_{t_i^{(g)} \in \mathcal{F}_i} \frac{\partial t_i^{(g)}}{\partial w_{ih}^k} \frac{\partial E_{net}}{\partial t_i^{(g)}} \tag{4.4}$$

We will first compute the derivative of the error with respect to the spikes of the hidden-neuron $\frac{\partial E_{net}}{\partial t_i^{(g)}}$. It depends on the derivatives of the errors with respect to all the spikes of the neural successors it is connected with, denoted by Γ^i:

$$\Gamma^i = \{j | j \text{ is postsynaptic to } i\}. \tag{4.5}$$

So the derivative of the error with respect to the spike $t_i^{(g)}$ of a hidden-neuron can be expanded to

$$\frac{\partial E_{net}}{\partial t_i^{(g)}} = \sum_{j \in \Gamma^i} \frac{\partial t_j^{(1)}}{\partial t_i^{(g)}} \frac{\partial E_{net}}{\partial t_j^{(1)}}. \tag{4.6}$$

The partial derivative $\frac{\partial E_{net}}{\partial t_j^{(1)}}$ is already computed in equation (3.4), so we only need to calculate the dependency of the first spike of an output-neuron on a spike of a hidden neuron $\frac{\partial t_j^{(1)}}{\partial t_i^{(g)}}$. This can be done using the same method as the calculation of $\frac{\partial t_j^{(1)}}{\partial w_{ji}^k}$, see equation (3.9):

$$\frac{\partial u_j(t_j^{(1)})}{\partial t_i^{(g)}} + \frac{\partial u_j(t_j^{(1)})}{\partial t_j^{(1)}} \frac{dt_j^{(1)}}{dt_i^{(g)}} = 0. \tag{4.7}$$

The factor $\frac{\partial u_j(t_j^{(1)})}{\partial t_j^{(1)}}$ is already computed in equation (3.12) so we only need to compute $\frac{\partial u_j(t_j^{(1)})}{\partial t_i^{(g)}}$. Again remembering the formula for the potential,

$$u_j(t) = \sum_{t_j^{(f)} \in \mathcal{F}_j} \eta(t - t_j^{(f)}) + \sum_{i \in \Gamma_j} \sum_{t_i^{(g)} \in \mathcal{F}_i} \sum_{k=1}^{l} w_{ji}^k \varepsilon(t - t_i^{(g)} - d^k), \tag{4.8}$$

it is easy to compute:

$$\frac{\partial u_j(t_j^{(1)})}{\partial t_i^{(g)}} = -\sum_{k=1}^{l} w_{ji}^k \varepsilon'(t_j^{(1)} - t_i^{(g)} - d^k). \tag{4.9}$$

Combining (3.12) and (4.9), $\frac{\partial t_j^{(1)}}{\partial t_i^{(g)}}$ can now be expressed in

$$\frac{\partial t_j^{(1)}}{\partial t_i^{(g)}} = \frac{\sum_k w_{ji}^k \varepsilon'(t_j^{(1)} - t_i^{(g)} - d^k)}{\sum_{i,k} \sum_{t_i^{(f)} \in \mathcal{F}_i} w_{ji}^k \varepsilon'(t_j^{(1)} - t_i^{(f)} - d^k)}. \tag{4.10}$$

Filling equations (4.10) and (3.4) into equation (4.6), we get the formula to calculate the error of the network with respect to a spike of a hidden-neuron:

$$\frac{\partial E_{net}}{\partial t_i^{(g)}} = \sum_{j \in \Gamma^i} \frac{\sum_k w_{ji}^k \varepsilon'(t_j^{(1)} - t_i^{(g)} - d^k)}{\sum_{i,k} \sum_{t_i^{(f)} \in \mathcal{F}_i} w_{ji}^k \varepsilon'(t_j^{(1)} - t_i^{(f)} - d^k)} (t_j^{(1)} - \hat{t}_j^{(1)}). \tag{4.11}$$

We also need to calculate the first term of equation (4.4), the partial derivative of the spike of the hidden-neuron with respect to the weight leading to that neuron, $\frac{\partial t_i^{(g)}}{\partial w_{ih}^k}$. The derivation is basically the same as for the weights to the output-neurons, except that the spike $t_i^{(g)}$ does not have to be the first spike of the hidden-neuron i, so we can not neglect the refractoriness-term. Equation (3.9) thus becomes:

$$\frac{\partial u_i(t_i^{(g)})}{\partial w_{ih}^k} + \frac{\partial u_i(t_i^{(g)})}{\partial t_i^{(g)}} \frac{dt_i^{(g)}}{dw_{ih}^k} + \sum_{\substack{t_i^{(f)} \in \mathcal{F}_i \\ f<g}} \frac{\partial u_i(t_i^{(g)})}{\partial t_i^{(f)}} \frac{dt_i^{(f)}}{dw_{ih}^k} = 0 \tag{4.12}$$

The summation reflects the dependence on the spikes that the hidden-neuron fired in the past. The first term is calculated like in equation (3.11):

$$\frac{\partial u_i(t_i^{(g)})}{\partial w_{ih}^k} = \sum_{t_h^{(p)} \in \mathcal{F}_h} \varepsilon(t_i^{(g)} - t_h^{(p)} - d^k) \tag{4.13}$$

The first factor of the second term is a bit more difficult than was the case with the first spike in equation (3.12) because of the refractoriness:

$$\frac{\partial u_i(t_i^{(g)})}{\partial t_i^{(g)}} = \sum_{t_i^{(f)} \in \mathcal{F}_i} \eta'(t_i^{(g)} - t_i^{(f)}) + \sum_{h,k} \sum_{t_h^{(p)} \in \mathcal{F}_h} w_{ih}^k \varepsilon'(t_i^{(g)} - t_h^{(p)} - d^k). \tag{4.14}$$

Because $\varepsilon'(s) = \eta'(s) = 0$ for $s \leq 0$ we have dropped the condition $f < g$. We also need to compute the derivative of the potential around spike-time $t_i^{(g)}$ with respect to an earlier spike $t_i^{(f)}$:

$$\frac{\partial u_i(t_i^{(g)})}{\partial t_i^{(f)}} = -\eta'(t_i^{(g)} - t_i^{(f)}). \tag{4.15}$$

By filling in (4.13), (4.14) and (4.15) in (4.12), we can calculate $\frac{dt_i^{(g)}}{dw_{ih}}$:

$$\frac{dt_i^{(g)}}{dw_{ih}^k} = \frac{-\sum_{t_h^{(p)} \in \mathcal{F}_h} \varepsilon(t_i^{(g)} - t_h^{(p)} - d^k) + \sum_{t_i^{(f)} \in \mathcal{F}_i} \eta'(t_i^{(g)} - t_i^{(f)}) \frac{dt_i^{(f)}}{dw_{ih}^k}}{\sum_{t_i^{(f)} \in \mathcal{F}_i} \eta'(t_i^{(g)} - t_i^{(f)}) + \sum_{h,k} \sum_{t_h^{(p)} \in \mathcal{F}_h} w_{ih}^k \varepsilon'(t_i^{(g)} - t_h^{(p)} - d^k)} \tag{4.16}$$

As can be seen this equation is recursive: the partial derivative $\frac{dt_i^{(g)}}{dw_{ih}^k}$ is expressed in all $\frac{dt_i^{(f)}}{dw_{ih}^k}$ with $f < g$. So first the derivative of the weight with respect to the the first spike should be calculated, then to the second spike, and so on.

By filling in equations (4.16) and (4.11) in equation (4.4) we can calculate the network error with respect to the hidden weight and thus we can calculate the weight-change for the weights leading to the hidden-neurons that is needed to lower the network error.

We now have an algorithm that can tune both the weights leading to the output neurons and those leading to the hidden neurons and is thus able to learn a two-layered SNN. It is easy to extend the algorithm to learn networks of even more layers, using the calculated network errors with respect to the spikes of a hidden-neuron to calculate the error with respect to a spike of a neural predecessor.

4.3 Benchmark tests

This section describes the application of the extended learning-algorithm with a 2-layered network on problems that are difficult or impossible to learn by networks with a single layer. First, the previously defined Exclusive-OR problem is tested to see if a multi-layered network has less trouble in learning it. Then we investigate if the extended algorithm can learn the Parity-n function, which could not be learned by a one-layered network

The network was tested on the encoding using 3 input-neurons as described in table 4.2. One hidden-layer with 5 hidden-neurons was used and $l = 16$ different delays for all

connections in the network. The weights were randomly initialized between -1.0 and 2.0 and the learning-rate η was set to 0.01. The network stopped learning when the sum squared error became smaller then 1.0.

While testing we found the same problem as Bohte did when testing his multi-layered network [5]. The network is not able to converge, unless we use strictly inhibitory and excitory hidden neurons; that is, the weights connecting one hidden-neuron with the output-neuron are all positive or all negative and should not change sign during learning. In our tests we used 4 excitory neurons and 1 inhibitory neuron.

After on average 95 training-cycles over 10 trials the algorithm successfully learned the XOR-problem. This number is significantly less than the 10^5 cycles that were needed to learn a one-layered network. The resulting networks were then tested on the noisy data, as with the one-layered networks, to test the stability of the solutions. The error increased only a little to 4.2, which is much less then the 52.1 of the one-layered network. It is save to draw the conclusion that the resulting networks do not use a hair-trigger situation.

We also tested the more general Parity-n function with $n = 3$ using the same architecture and parameter values as used for the XOR-problem. As opposed to the one-layer algorithm, the two-layer algorithm did correctly learn the function. After on average 564 training-cycles the network classified the 8 patterns within the 1.0 error-threshold.

4.4 Conclusion

A one-layered SNN is very powerful, even more powerful than conventional networks with one layer. Specifically, we have shown that a one-layered SNN can solve the classical XOR-problem, while this was assumed to be impossible [5]. Besides the XOR-problem however, it is not able to solve all possible non-linear functions. An SNN with multiple layers however can represent all possible functions. Therefore we have extended the learning-rule so that it is applicable to networks with more layers. We have shown that the extended learning-algorithm can learn SNNs to perform non-linear classification tasks on sets of spike-trains. In addition it can solve the XOR-problem with less computations than using a one-layered network.

5 Lipreading benchmark

In this chapter we apply our learning architecture to the problem of automated lipreading. The field of lipreading, or speechreading, is concerned with the difficult task of converting a video signal of a speaking person to written text. In order to successfully accomplish this task it is clear that a recognition system is needed that can cope with spatio-temporal patterns, such as a spiking neural network.

In section 5.1 we will briefly review the field of automated lipreading. Section 5.2 describes the general architecture of the classification system and gives a description of the specific lipreading task it will be tested on. In section 5.3 we report the results of the conducted experiments. And finally, in section 5.4 we make some concluding remarks.

5.1 Introduction lipreading

Human speech does not only use an acoustical signal but also a visual signal in the form of lip movement. The information carried by the visual signal complements the information carried by the acoustical signal [10, 11, 39]. So speech recognition can be greatly improved by not only processing the audio signal but also taking the visual signal into account. Research shows that combined audio-visual speech recognition systems perform better then audio-only recognizers, especially in a noisy environment [11, 10, 39, 60].

The same recognition techniques that are used to cope with the temporal aspect of audio recognition are also used for lipreading systems. So it is not surprising that most lipreading systems use Hidden Markov Models (HMM) to recognize the temporal correlations of the lipreading video [9, 10, 56]. Because HMMs can only use a limited set of input variables [46], quite a lot of preprocessing is needed to code each frame of the video in a few input values. Usually prior knowledge of the speechreading problem is incorporated in this preprocessing stage. For example, various studies use parametrized models that are optimized to describe the form of the lips in only a few variables [27, 56]. And some even build feature-extraction methods to find out whether the tong is visible or not [62, 45].

There are also lipreading systems using neural networks. These systems need considerable less preprocessing because the number of input variables can be much larger. The most notable neural network architecture used is the Time Delayed Neural Network (TDNN) [39, 8, 56], which is a special case of a sigmoidal feedforward network using standard backpropagation. And some studies use recurrent neural networks like

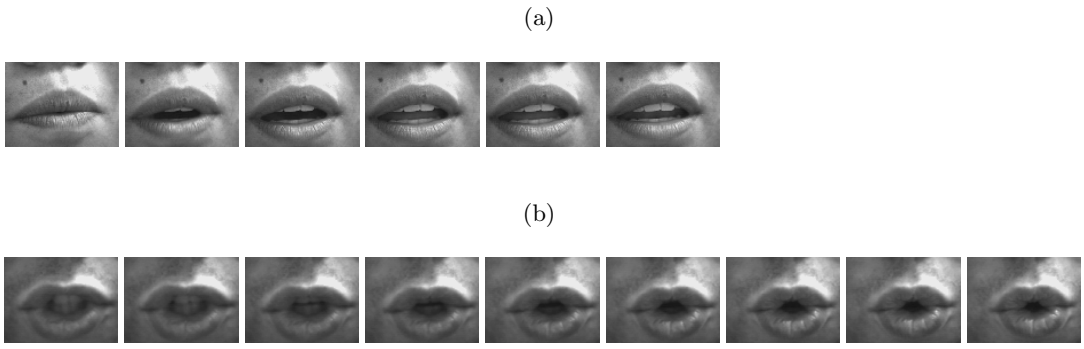


Figure 5.1: Two example image-sequences of the Tulips database. (a) A video of 6 frames showing someone pronouncing "one". (b) A video of 9 frames showing someone pronouncing "two".

the Elman Hierarchical Neural Network [60]. The neural networks are usually fed with pixel-based input, meaning that a feature represents the value of one or a few pixels in a video-frame, and avoid using explicit features as with HMMs. The idea behind this is that the algorithm should find its own internal features to learn a lipreading task [39].

We want to investigate the capability of spiking neural networks in lipreading. We will also try to limit the preprocessing as much as possible and use pixel-based features as input. This is done because we want to test the general learning capabilities of our algorithm on spatio-temporal patterns and much of the dynamic characteristics of these patterns are lost if we extract explicit features.

5.2 System overview

For the experiments we use the Tulips Audiovisual database [40]. It contains gray-value image-sequences of 12 persons pronouncing the first four digits 2 times. Thus in total there are 96 video fragments, each consisting of 6 to 16 frames. In figure 5.1 two image sequences are shown; one showing the pronunciation of the word "one" and the other of the word "two". The SNN should learn to distinguish these lip-stances and discover the temporal structure among the frames.

Spiking neural networks can only be fed with spike-times, so we should first encode every video into spike-trains. We do this by first converting every frame to a black and white version and then coding these monochrome videos into spikes, see figure 5.3. This conversion should be done in a sensible way to try to preserve salient information as much as possible. We chose to invert the gray-value of every frame and then threshold the result, so to keep only the dark mouth-opening, which gives a good description of the stance of the mouth. With a threshold value of 205 most facial features are discarded

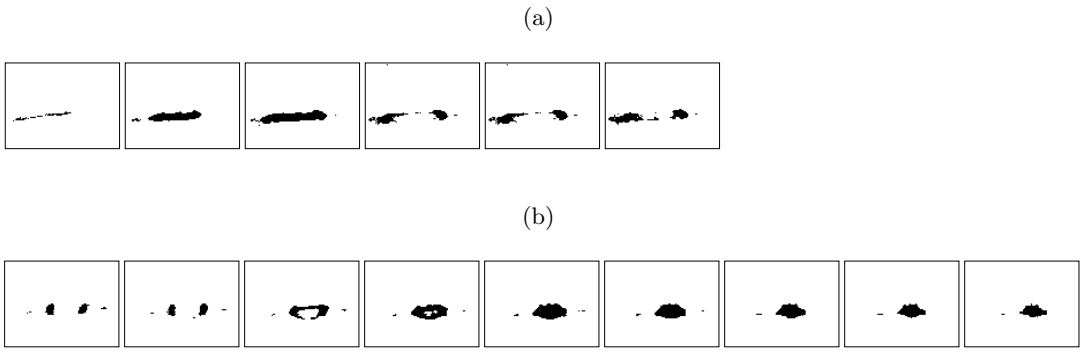


Figure 5.2: The thresholded versions of the two examples of figure 5.1 (the images are not inverted for visual clarity). The black pixels in this figure had a low gray-value in figure 5.1. As can be seen the information of the position of the lips is preserved, although a lot of data is discarded.

while the mouth-opening is preserved. In figures 5.2(a)-(b), which are the thresholded versions of figures 5.1(a)-(b) respectively, the stances of the lips are still discernible although a lot of information is lost. Coding these monochrome image-sequences into spike-trains is straightforward. For every pixel-location in the video we assign one spiking neuron, see figure 5.3. The spike-train of that neuron is given by the times that that neuron is white. The resolution of the videos in the data-set is 100 by 75, so each video is converted to a set of 7500 spike-trains. Because we chose the thresholding value to be rather high each video was coded in only 2500 spikes on average.

As can be seen the preprocessing is kept simple, so a lot of weight is put on the recognition system, in our case an SNN. There is a lot of variation in the videos apart from the different words that are uttered. The center of the mouth is generally not in the middle of the images and between speakers there is big difference of the illumination. Also, the length of the video varies a lot, from 6 frames to 16 frames. The SNN should learn to generalize over these properties and only learn those features that are invariant for the recognition of words.

5.3 Experiments

In the following subsections we will describe two experiments. The first is the classification of the two digits "one" and "two". The second task is somewhat more difficult: the classifier has to learn all four digits and is tested on a new person that it has not seen before.

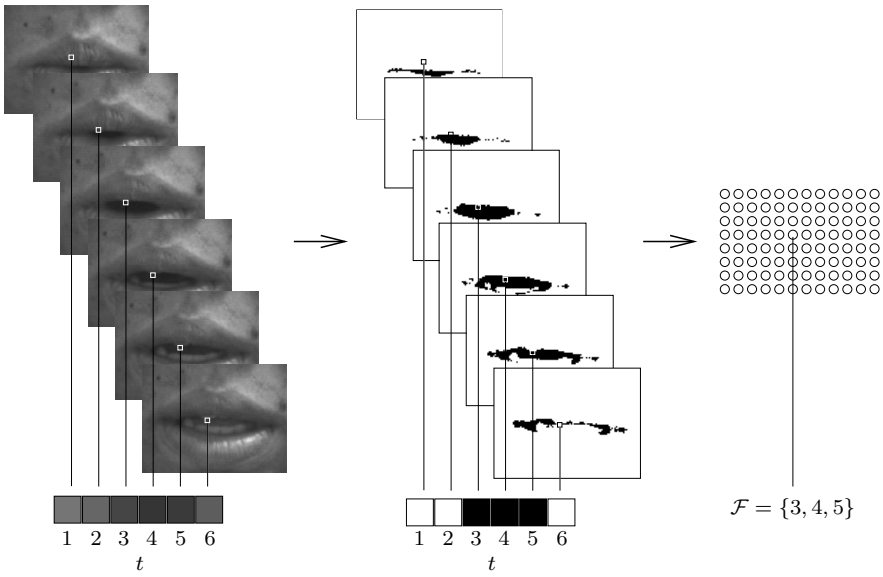


Figure 5.3: In this overview the preprocessing of an image-sequence is shown and for clarity one pixel-location is enlarged. First all frames of a gray-scale video are transformed into monochrome video using thresholding. (The images are not inverted for visual clarity.) Then all the pixel-locations are transformed to spike-trains. The enlarged pixel-location has a small value for time $t =$ 3, 4 and 5, so the spiking-neuron for that pixel-location will spike at those times.

number of delays	20
initial weights	$[-10^{-5}, 10^{-4}]$
learning-rate	10^{-5}
stopping error	250

Table 5.1: Parameter settings for the lipreading task.

5.3.1 Classify "one" and "two"

In this task a spiking neural network has to learn to discriminate between the utterance of the two digits "one" and "two". There are 24 image-sequences for each of these classes in the data-set, which are randomly split into a training-set and a test-set. Because the number of patterns is quite low, we only use 4 patterns for the test-set and the remaining 44 as training-patterns. By repeating the experiments a lot of times with different test-sets, we can ensure that there are enough tests for reliable results.

The network-output consists of only one spiking neuron that is trained to fire an early spike at $\hat{t} = 17$ if an utterance of "one" is fed to the network and a late spike at $\hat{t} = 24$ if the input is of class "two". Each of the 7500 input-neurons is connected to the output-neuron with 20 synapses with delays $\{1, 2, \ldots, 20\}$ ms. The weights were randomly initialized between -10^{-5} and 10^{-4} and the learning-rate η was set to 10^{-5}. The training phase was stopped when the sum squared error dropped below 250 (see table 5.1 for an overview of the parameter settings).

While doing the experiment we discovered that for some input-videos the output neuron would not spike at all, because there were too few input-spikes, due to a brighter image-sequence. The problem for the learning algorithm is that the error is undetermined if there is no output-spike. Using larger initial weights, so that all input-patterns caused an output-spike, did not counteract this problem, because the learning algorithm would set the weights lower in order to correctly classify the patterns with an average number of spikes. We fixed this by assigning an network error of 4.0 if there was no spike and raising the weights a little.

After training, the evaluation of the output of the train- and test-patterns were done as follows. If the output-spike was closer to the early spike-time ($t < 20.5$) then the classification of the network was "one" and if the output-spike was closer to the late spike-time ($t > 20.5$) or there was no output-spike the classification of the network was "two". If the time of the output-spike was exactly in the middle (20.5), what sometimes happened, a first order approximation of the potential around $t = 20.5$ was performed to get a real valued approximation of the spike-time (as in [5]).

After 5.8 training-cycles, averaged over 100 experiments, the stopping-criterion was met and 91% of the training-patterns was classified correctly. This number of training-cycles is very low in comparison with other neural network studies [57, 24]. The test-set was classified on average 75% correct.

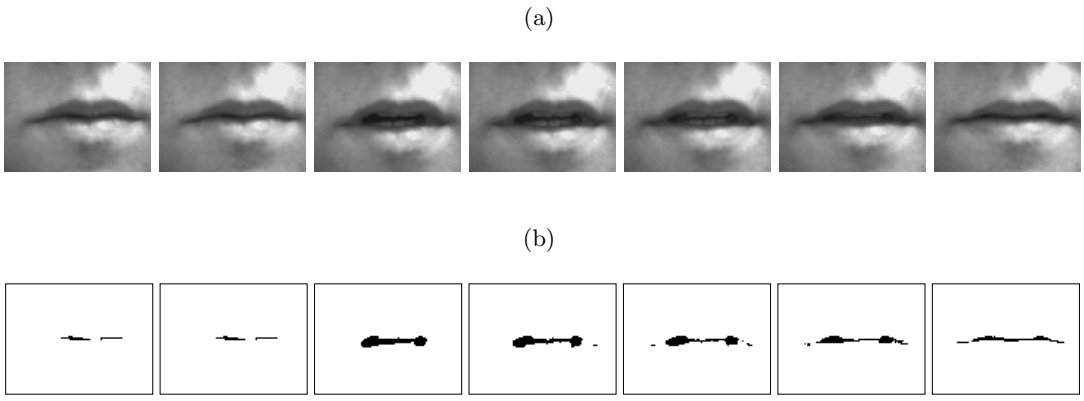
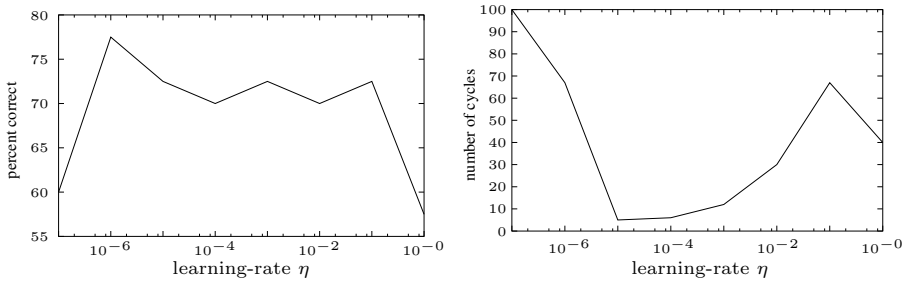


Figure 5.4: An image-sequence of a pronunciation of "one" that is very hard to classify. (a) In the original gray-scale image there is hardly any information, because the mouth opens only slightly. (b) Very few pixel are dark enough to be converted to a spike-time.

Looking more closely at the outcome of the test phase, it became clear that some patterns are very difficult and almost never correctly classified, while others are very easy. Partly this has to do with differences of the number of spikes due to the variation in brightness. Some of the image-sequences are very bright and thus have less dark pixel-values, producing around a 1000 spikes, while others are coded in 5000 spikes. An example of a pattern that is very difficult to classify, because it has so few spikes is depicted in figure 5.4.

We further investigated the influence of the learning-rate parameter η on the performance of the network. Like in conventional neural networks using backpropagation it is difficult to set this parameter to a sensible value, so it is important to see if this choice has a big influence on the resulting classifier. In figure 5.5(a) can been seen that for a wide range of values, from 10^{-6} to 10^{-1}, the accuracy is the same, around 72% correct. Only an extreme setting of the learning-rate will increase the error. As could be expected the learning-parameter also has an effect on the training time of the algorithm, see figure 5.5(b). Sometimes the algorithm could not reach the stopping-criterion in a reasonable number of cycles. This is caused by the slow convergence if a small learning-rate is used and the bouncing-effect for large values, each time-step overshooting a local minimum. To prevent very long training times, another stopping-criterion had to be used, stopping the training-phase after 100 cycles.

For this lipreading task a wide range of values for the learning-rate parameter can be used, without much influence on the performance. Thus the choice of the learning-rate is not that important. And this is very important, because people without much knowledge of SNNs and the given data-set that should be classified, require to set the learning-parameters.



(a) (b)

Figure 5.5: The influence of the learning-rate on the performance. (a) The dependence of the error on the learning-rate; in a range of 10^{-6} to 10^{-1} the performance is stable. (b) The dependence of the number of training-cycles on the learning-rate; the learning-time is small if the learning-rate is in the range of 10^{-5} to 10^{-3}.

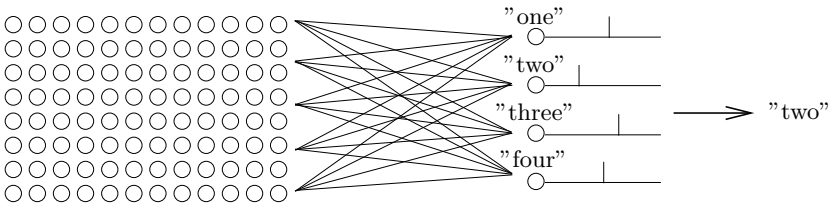


Figure 5.6: In the second task the output of the network is produced by four output-neurons, one for each digit. The neuron that fires first denotes the output of the network; so in this case the output is "two".

5.3.2 Classify new person

The second task we tested the algorithm on is somewhat more complicated. The network will be explicitly tested on its invariance of the speaker by training and testing on a different set of persons. Also, in this task the whole data-set is used including the digits "three" and "four". The date-set was split in a training- and test-set using the leave-one-out method: training was performed on the pronunciations off 11 speakers, comprising of 88 patterns, while the classifier was tested on the 8 image-sequences of the remaining speaker. Repeating this procedure 12 times, each time with a different person in the test-set, we get a reliable performance measure.

Because the output of the network consists of four classes the network architecture consists of 4 output-neurons, one for each digit, see figure 5.6. For the training- and testing-phase we will use the same scheme as for the Poisson spike-train benchmark (section 3.4). The network is trained to let the output-neuron, representing the correct

digit fire an early spike ($\hat{t} = 17$), while the other output-neurons fire a late spike ($\hat{t} = 24$). During the testing phase we only look at the neuron that fires the first spike and consider the corresponding digit as the output of the network. Sometimes two output-neurons fire at exactly the same time, so we use the same procedure as in the first experiment, approximating the real spike-time using the potential around the spike-time.

The parameters are set to the same values as the first experiment (see table 5.1), except for the stopping criteria. Because there are more output-neurons and training-patterns the sum squared error is generally higher (see equation 3.17). So we raised the error-threshold to 1500.

The number of cycles needed for the training-phase was only 6.25, averaged over the 12 runs. The network achieved a recognition-rate of 57% on the test-patterns that it had not seen before. This result is not as good as the accuracy reported in studies of specialized lipreading systems which range from 65.5% to 90.6% [26], but considering that the system is not optimized for lipreading or video-classification in general, the performance is quite good.

5.4 Conclusion

It is clear that the simple lipreading system using an SNN can perform a lipreading task. But it is difficult to compare the results with other lipreading systems, because these are highly optimized for this task, whereas our system is a very generic classifier. The system we used could also be optimized for lipreading, by doing more preprocessing on the video data and extract features that explicitly describe the stance of the lips and tongue. By doing this the accuracy will undoubtedly improve for the task we are considering. But that is not the goal of this study, which aims to provide a new way of learning general temporal data.

Another thing that can be improved is the network architecture. When processing spatial data, such as images and video, the system should take advantage of the local ordering of the information. In neural networks this can be implemented by using so-called receptive fields [12]; that is, there are one or more hidden-layers consisting of neurons that are only connected to a small region of the input and thus process local information of that region.

The important thing to conclude is that spiking neural networks can be used as video-classifiers. The classifier can be learned to perform a specific task using the learning-rule for spike-trains we developed. In this way we have developed a learnable classification technique, that is very generic because of its use of a neural network.

6 Discussion and Conclusion

In this study we developed and tested a novel learning-rule for feedforward spiking neural networks. It is now possible to apply an SNN on temporal data in the same way as conventional neural networks are applied to static data using the backpropagation-rule. This is an important step towards the integration of SNNs in intelligent systems that have to process streaming data, such as audio and video.

6.1 The algorithm

We developed a learning algorithm that can be applied to spiking neurons that fire multiple spikes and this makes it different to existing learning-rules, which limit the spiking neurons to fire only once [50, 5]. We accomplished this by taking the previous spikes into account when computing the error with respect to a new spike. In this way the learning-rule has become a recursive function and must first be applied to the earliest spike, then to the second and so on until the error is computed with respect to all spikes. Although this seems to be a computationally intensive task, a lot of the factors that have to be computed are shared among the spikes and can be computed in advance.

For the derivation of the learning-rule we used the gradient-descent method, which was also used for the backpropagation-rule for conventional neural networks. Bohte uses the same method for the development of his SpikeProp-rule and states that the discontinuous nature of the spiking neuron has to be overcome [5, 4]. The problem is that the spiking neuron uses a threshold function to compute its activation and the gradient descent method would require the calculation of the derivative of this function, which can only be done in approximation. This is the reason why conventional neural networks using the backpropagation-rule need a continuous activation-function, like the sigmoid-function, instead of a threshold function. We do not agree that this kind of linearization of the threshold function is necessary for calculating a gradient descent rule for SNNs and we did not use it in our derivation. Furthermore we do not think it is used in the SpikeProp-rule either. In our opinion the linearization that is mentioned in [5] is related to the linear direction of the gradient, which is common to all gradient descent methods. When using precisely timed spikes the threshold function does not have to be approximated, because the output of the neuron resides in the time domain, which is continuous. There is a continuous mapping from the input-spikes and the strength of the presynaptic weights to the time of the output-spike, so the gradient descent method can directly be applied. However, caution must be taken with spikes that were produced by a potential that was rising very slowly when passing the threshold. For this special case we incorporated

an ad hoc rule to counteract problems with the learning algorithm. We suspect that the SpikeProp rule can also learn more stable and more efficient if such a technique is applied.

The functionality of the learning algorithm is first of all shown by the ability to classify artificially generated Poisson spike-trains with a one-layered network. The ease with which this is done is encouraging, but also makes us realize that this benchmark is too simple for a good evaluation of the algorithm. Further testing should be conducted using temporal patterns that are more closely correlated among the classes. For example, in [42] patterns are created by combining segments of different Poison spike-trains. Because patterns can then contain the same segments but in a different recombination, the patterns are locally the same, but on the whole different. The algorithm could be better evaluated using these kind of experiments.

An SNN with two layers of weights was successfully applied on a few standard non-linear benchmarks: the Exclusive-OR (XOR) function with and without noise and the Parity-3 function. The aim of these experiments was to show that an SNN with more then one layer could indeed solve non-linear tasks, as opposed to one-layered networks that had problems solving them. Again, it must be noted that further testing is required to evaluate the performance of the algorithm on non-linear tasks. The most important problem domain on which we did not experiment is the classification of non-linear temporal patterns, while our learning algorithm using a multi-layered SNN should be very well suited for these kinds of tasks.

6.2 Lipreading

To test the algorithm on artificially generated tasks is fairly easy, because the amount of noise can be controlled and the number of test-patterns is boundless. It is important to perform such experiments as a first evaluation, but to really determine the qualities of the algorithm we had to apply it on a real-world problem.

We chose to do this on the problem of lipreading, learning to recognize spoken words out of video-fragments. A big obstacle in this task, like all tasks involving the processing of video-data, is the high dimension of the input. Previous methods overcame this problem by doing a lot of preprocessing, or feature-extraction, usually using a lot of prior knowledge of the specific domain [56]. We used only minimal preprocessing, feeding the SNN with almost raw video-data. Thus the network had to tune a lot of parameters in order to correctly classify the training data. This brings a big risk of overfitting the data and performing miserably on the test data. The results were however very acceptable, although not as good as the systems specially designed for lipreading. There is certainly room for improvement, not only by incorporating prior knowledge of the lipreading domain but also by improving the general capability of processing video-data.

One thing that certainly needs improvement is the encoding method for transforming video-data into spike-trains. In our experiments we thresholded the analog data-streams

per pixel-location. Such a technique however does not produce very satisfying spike-trains. The ideal encoding would distribute the information of a given video-stream in an evenly manner over all spike-times of all spike-trains, respecting the spatial and temporal ordering of the data [47]. The firing-times would then be independent of each other; that is, the information they carry would be orthogonal, just like the Poisson spike-trains. A possible way to approach this goal would be to use receptive fields of spiking neurons. This is already done for static images and proved very successful [12, 4]. If this could be extended with receptive fields that are sensitive to changes in time rather then in space, it would be possible to improve the encoding of video-data. Of course this requires a lot of extra study and is an issue for future work.

Our lipreading-system shows that spiking neural networks can indeed be used for practical applications involving the processing of temporal data. These applications, no matter how simple, are usually performed by applying an expert-system specially designed for the task and thus requiring a lot of domain-specific knowledge. The SNN method however can be considered as a black-box, like conventional neural networks, that can perform general pattern recognition tasks.

6.3 Computational power

During our study on the learning of SNNs, we also investigated the computational power of SNNs in general. In addition to what was already known, namely that they are more powerful than conventional neural networks [30], we have shown that a one-layered SNN can solve the notorious XOR-problem, while this was taken to be impossible [5]. Although the XOR-function is by itself not very special, it occupies a special position in the history of neural network research. It is the simplest non-linear function that can not be computed by a one-layered conventional neural network [51]. The fact that one-layered SNNs can solve this function should be of considerable interest to the neural network community. It underlines the power of computing with the new spiking neural network models as opposed to the conventional neural network paradigms.

6.4 Future work

The use of spiking neural networks as a computational tool is a relatively recent development. It is not surprising that a lot of work still needs to be done before SNNs will become a common tool; for example, that it will be incorporated in the Matlab Neural Network Toolbox. The most important aspect that has to be improved is the coding-scheme for transforming data-streams into spike-trains, as discussed in section 6.2 regarding the lipreading application. Besides video-data, there is also need for suitable coding-schemes for other dynamic data such as positional data from tracking systems and audio data.

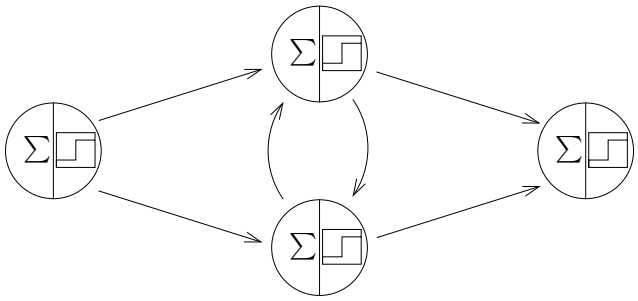


Figure 6.1: An example recurrent network architecture. The two neurons in the middle are connected in both directions, leading to a loop in the network.

Focusing on the developed learning algorithm, there are numerous possibilities of further research. For example applying the algorithm on other network architectures than the feedforward structure we used. The algorithm can in principle train all kinds of networks, including recurrent architectures, see figure 6.1. Supervised learning-rules for conventional neural networks did not have this ability, because in a recurrent structure it is impossible to perform backpropagation of the error; it would be unclear if a specific neurons state is the cause or the effect of another neurons state. For SNNs this problem does not exist because the state of a spiking neuron can only influence the state of another neuron in the future. By using a recurrent architecture the network could learn to remember certain information for as long as it needs, which could be helpful to find temporal correlations that happen on a large or variable time-scale.

Other research could aim at extending the algorithm in analogue with well-tested extensions of the conventional backpropagation-rule. For example, the efficiency could be improved by introducing a momentum term in the learning-rule, which is already attempted for the SpikeProp algorithm [61]. Or the algorithm could be changed more drastically in accordance with the QuickProp method [15], to improve the performance even more.

It is our hope that the introduction of our learning algorithm will contribute to the field of spiking neural networks and not only lead to more research as suggested here, but also to a wider use on practical applications such as lipreading.

Bibliography

[1] W. Bialek, F. Rieke, R.R. de Ruyter van Steveninck, and D. Warland. Reading a neural code. *Science*, 252:1854–1857, 1991.

[2] C. M. Bishop. *Neural Networks for Pattern Recognition*. Clarendon Press, Oxford, 1995.

[3] S. M. Bohte. The evidence for neural information processing with precise spike-times: A survey. *Natural Computing*, 3(2):195–206, 2004.

[4] S.M. Bohte. *Spiking Neural Networks*. PhD thesis, Universiteit Leiden, 2003. Available from http://homepages.cwi.nl/~sbohte/publications2.htm.

[5] S.M. Bohte, J.N. Kok, and H. La Poutré. Error-backpropagation in temporally encoded networks of spiking neurons. Technical report, CWI Technical Report SEN-R0037, 2000. Available from http://db.cwi.nl/rapporten.

[6] S.M. Bohte, H. La Poutré, and J.N. Kok. Unsupervised clustering with spiking neurons by sparse temporal coding and multi-layer rbf networks. Technical report, CWI Technical Report SEN-R0036, 2000. Available from http://db.cwi.nl/rapporten.

[7] N. K. Bose and P. Liang. *Neural network fundamentals with graphs, algorithms, and applications*. McGraw-Hill, Inc., Hightstown, NJ, USA, 1996.

[8] C. Bregler, S. Manke, H. Hild, and A. Waibel. Improving connected letter recognition by lipreading. In *Proceedings of the International Conference on Acoustics, Speech and Signal Processing (IEEE-ICASSP)*, pages 557–560, Minneapolis, MN, 1993. IEEE Press.

[9] C. Bregler and S. M. Omohundro. Nonlinear manifold learning for visual speech recognition. In *Proceedings of the Fifth International Conference on Computer Vision*, pages 494–499. IEEE Computer Society, 1995.

[10] T. Chen. Audiovisual speech processing, lip reading and lip synchronization. *IEEE Signal Processing Magazine*, pages 8–21, 2001.

[11] T. Chen and R. Rao. Audio-visual integration in multimodal communications. In *Proceedings of IEEE, Special Issue on Multimedia Signal Processing*, pages 837–852, 1998. volume 86, no. 5.

[12] A. Delorme and S. Thorpe. Face processing using one spike per neuron: resistance to image degradation. *Neural Networks*, 14(6-7):795–804, 2001.

[13] R. O. Duda, P. E. Hart, and D. G. Stork. *Pattern Classification*. Wiley, New York, 2001.

[14] J. L. Elman. Finding structure in time. *Cognitive Science*, 14(2):179–211, 1990.

[15] S. E. Fahlman. An empirical study of learning speed in back-propagation networks. Technical Report CMU-CS-88-162, Carnegie Mellon, 1988.

[16] W. Gerstner. Spiking neurons. In W. Maass and C. M. Bisop, editors, *Pulsed Neural Networks*, pages 3–54. MIT Press (Cambridge), 1999.

[17] W. Gerstner, R. Kempter, J. L. van Hemmen, and H. Wagner. A neuronal learning rule for sub-millisecond temporal coding. *Nature*, 384:76–78, 1996.

[18] W. Gerstner and W. M. Kistler. *Spiking Neuron Models: Single Neurons, Populations, Plasticity*. Cambridge University Press, 2002.

[19] D. Heeger. Poisson model of spike generation, 2000. Available from http://www.cns.nyu.edu/˜david/ftp/handouts/.

[20] J. J. Hopfield. Pattern recognition computation using action potential timing for stimulus representation. *Nature*, 376:33–36, 1995.

[21] R. Kempter, W. Gerstner, and J.L. van Hemmen. Hebbian learning and spiking neurons. *Physical Review E*, 59(4):4498–4514, 1999.

[22] C. Koch and I. Segev, editors. *Methods in Neuronal Modeling, from synapses to networks*. MIT Press, Cambridge, 1989.

[23] M.S. Lewicki. Efficient coding of natural sounds. *Nature Neuroscience*, 5(4):356–363, 2002.

[24] W. C. Lin. A space-time delay neural network for motion recognition and its application to lipreading in bimodal speech recognition. Master's thesis, National Chiao-Tung University, Taiwan, 1996.

[25] L.Torres, L.Lorente, and J. Vila. Automatic face recognition of video sequences using self-eigenfaces. In *International Symposium on Image/video Communication over Fixed and Mobile Networks*, Rabat(Morocco), 2000.

[26] J. Luettin and N. A. Thacker. Speechreading using probabilistic models. *Computer Vision and Image Understanding*, 65(2):163–178, 1997.

[27] J. Luettin, N. A. Thacker, and S. W. Beet. Locating and tracking facial speech features. In *Proceedings of the International Conference on Pattern Recognition (ICPR'96)*, volume I, pages 652–656. IAPR, 1996.

[28] W. Maass. Lower bounds for the computational power of networks of spiking neurons. *Neural Computation*, 8(1):1–40, 1996.

[29] W. Maass. Networks of spiking neurons: the third generation of neural network models. *Neural Networks*, 10(9):1541–1741, 1997.

[30] W. Maass. Noisy spiking neurons with temporal coding have more computational power than sigmoidal neurons. In M. Mozer, M. I. Jordan, and T. Petsche, editors, *Advances in Neural Information Processing Systems, volume 9*, pages 211–217. MIT Press, Cambridge, MA, 1997.

[31] W. Maass. Computing with spiking neurons. In W. Maass and C. M. Bishop, editors, *Pulsed Neural Networks*, pages 55–85. MIT Press (Cambridge), 1999.

[32] W. Maass and C. M. Bishop, editors. *Pulsed Neural Networks*. MIT Press, Cambridge, 1999.

[33] W. Maass, T. Natschläger, and H. Markram. Real-time computing without stable states: A new framework for neural computation based on perturbations. *Neural Computation*, 14(11):2531–2560, 2002.

[34] W. Maass, T. Natschläger, and H. Markram. A model for real-time computation in generic neural microcircuits. In S. Becker, S. Thrun, and K. Obermayer, editors, *Proc. of NIPS 2002, Advances in Neural Information Processing Systems*, volume 15, pages 229–236. MIT Press, 2003. Available from http://www.cis.tugraz.at/igi/maass/publications.html.

[35] W. Maass and T. Natschlger. Emulation of hopfield networks with spiking neurons in temporal coding. In J. M. Bower, editor, *Computational Neuroscience: Trends in Research*, pages 221–226. Plenum Press, 1998.

[36] W. Maass and M. Schmitt. On the complexity of learning for spiking neurons with temporal coding. *Information and Computation*, 153:26–46, 1999.

[37] R.J. MacGregor. *Neural and Brain Modeling*. Academic Press, San Diego, 1987.

[38] J. L. McClelland, D. E. Rumelhart, and the PDP Research Group, editors. *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Volume 2: Psychological and Biological Models*. MIT Press, Cambridge, MA, 1986.

[39] U. Meier, R. Stiefelhagen, J. Yang, and A. Waibel. Towards unrestricted lip reading. *International Journal of Pattern Recognition and Artificial Intelligence*, 14(5):571–585, 2000.

[40] J. Movellan. Visual speech recognition with stochastic networks. In G. Tesauro, D. Toruetzky, and T. Leen, editors, *Advances in Neural Information Processing Systems*, volume 7, pages 851–858. MIT Press, Cambridge, 1995.

[41] R. Mueller and A. V. M. Herz. Content-addressable memory with spiking neurons. *Physical Review E*, 59(3):3330–3338, 1999.

[42] T. Natschläger and W. Maass. Information dynamics and emergent computation in recurrent circuits of spiking neurons. In Sebastian Thrun, Lawrence Saul, and Bernhard Schölkopf, editors, *Advances in Neural Information Processing Systems 16*. MIT Press, Cambridge, MA, 2004.

[43] T. Natschläger and B. Ruf. Spatial and temporal pattern analysis via spiking neurons. *Network: Computation in Neural Systems*, 9(2):319–332, 1998.

[44] A. Pouget, R.S. Zemel, and P. Dayan. Information processing with population codes. *Nature Review Neuroscience*, 1(2):125–1–32, 2000.

[45] K.V. Prasad, D.G. Stork, and G.J. Wolff. Prepocessing video images for neural learning of lipreading. Technical report, Ricoh California Research Center, Technical Report CRC-TR-93-26, 1993.

[46] Lawrence R. Rabiner. A tutorial on hidden markov models and selected applications in speech recognition. *Readings in speech recognition*, pages 267–296, 1990.

[47] F. Rieke, D. Warland, R. de Ruyter van Steveninck, and W. Bialek. *Spike: Exploring the Neural Code*. Computational Neurosciences. MIT Press, Cambridge, 1997.

[48] Fitzsimonds R.M., Song H-J., and Poo M-M. Propagation of activity-dependent synaptic depression in simple neural networks. *Nature*, 388:439–448, 1997.

[49] F. Rosenblatt. *Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms*. Spartan Books, Washington, 1962.

[50] B. Ruf and M. Schmitt. Hebbian learning in networks of spiking neurons using temporal coding. In J. Mira, R. Moreno-Diaz, and J. Cabestany, editors, *Biological and artificial computation: From neuroscience to technology*, pages 380–389. Springer, Berlin, 1997. volume 1240 of Lecture Notes in Computer Science.

[51] D. E. Rumelhart, J. L. McClelland, and the PDP Research Group, editors. *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Volume 1: Foundations*. MIT Press, Cambridge, MA, 1986.

[52] B. Schrauwen. Pulstreincodering en training met meerdere datasets op de cbm. Master's thesis, Ghent University, 2002.

[53] T. J. Sejnowski and C. R. Rosenberg. Nettalk: A parallel network that learns to read aloud. In J. A. Anderson and E. Rosenfeld, editors, *Neurocomputing*. MIT Press, Cambridge, MA, 1988.

[54] J.-W. Sohn, B.-T. Zhang, and B.-K. Kaang. Temporal pattern recognition using a spiking neural network with delays. In *Proceedings of Internationl Joint Conference on Neural Network (IJCNN'99)*, pages 2590–2593, 1999. volume 4.

[55] J. Storck, F. Jkel, and G. Deco. Temporal clustering with spiking neurons and dynamic synapses: towards technological applications. *Neural Networks*, 14(3):275–285, April 2001.

[56] D. G. Stork and M. E. Henneck, editors. *Speechreading by Man and Machine: Data, Models and Systems*. NATO/Springer-Verlag, New York, NY, 1996.

[57] D. G. Stork, G. Wolff, and E. Levine. A neural network lipreading system for improved speech recognition. In *Proceedings of the 1992 International Joint Conference on Neural Networks*, pages 285–295, Baltimore, MD, 1992.

[58] S. Thorpe, A. Delorme, and R. Van Rullen. Spike-based strategies for rapid processing. *Neural Networks*, 14(6-7):715–725, 2001.

[59] S.J. Thorpe, A. Delorme, R. VanRullen, and W. Paquier. Reverse engineering of the visual system using networks of spiking neurons. *IEEE International Symposium on Circuits and Systems*, 4:405–408, 2000.

[60] J.C. Wojdel and L.J.M. Rothkrantz. Using artificial neural networks in lip-reading. In *Proceedings of 7th annual conference of the Advanced School for Computing and Imaging (ASCI 2001)*, The Netherlands, 2001. Heijen.

[61] Jianguo Xin and M. J. Embrechts. Supervised learning with spiking neuron networks. In *Proceedings IEEE International Joint Conference on Neural Networks, IJCNN01*, Washington D.C., July 2001.

[62] X. Zhang, R. M. Mersereau, M. A. Clements, and C. C. Broun. Visual speech feature extraction for improved speech recognition. In *Proceedings of the International Conference on Acoustics, Speech and Signal Processing (IEEE-ICASSP)*, Orlando, 2002.