LIACS
Leiden Institute of Advanced Computer Science

Master's Thesis
June 17, 2009

Temporal Pattern Analysis
Using Reservoir Computing

Author:
Ron Vink

Supervisor:
Dr. Walter Kosters

Contents

Chapter 1

Introduction

The goal of research in the area of *Artificial Neural Networks* is twofold. First of all we want to gain a better understanding in order to build accurate models of our brains, and to better understand what can go wrong with it. The second part is that we need better computational devices for a range of real world problems. Often these problems are (too) hard to understand, computationally too expensive with traditional methods, or the problem has to be solved with incomplete or noisy data. Neural networks are an excellent way of dealing with these issues, and much research is being done to improve the capacity, accuracy and biological realism of these connectionist models of computation.

In this thesis the application of data mining will be addressed with neural networks. The emphasis will be on neural networks as a computational device rather than a biological realistic artificial version of our brains. However biological realism is pursued. Previous work suggests that the more biologically realistic the model is, the more powerful the device is as a computational entity. The key will be the use of *temporal data*; data where the data points are dependant on other data points in the data set, i.e., have a causal relation. This means that an input x partly determines the output of the network for input $x + 1$ since the state of the network is still affected by input x when input $x + 1$ is being presented. Many real world problems are characterized by temporal data. Vision, speech, language, motor control, goal directed-behavior and planning are all examples where temporal computational models are used. In fact most human behavior and many processes are characterized by the notion of time somehow. Therefore the implicit representation of time in connectionist models is a crucial step in closing the gap between the performance of nature's evolved cognition and our artificially constructed imitations of it.

Many attempts have been made to somehow incorporate time in com-

putational models. Explicit methods where information about past states is distributed or stored in tapped delay lines or context layers have been proven to work but are problematic in terms of storage or computational complexity [10, 36]. More information on ways to explicitly incorporate the notion of time in neural networks can be found in [12, 18, 10].

A better way might be to incorporate time implicitly, thereby overcoming the problem when to discard information since the information is not actually kept. When time is being represented implicitly it means that it has an effect on processing the current state of the system, rather than having some representation of time as an additional input to, or data structure in the system.

This thesis is based on the work of Maass [21] where he introduced a new computational paradigm, the *Liquid State Machine*. The model he introduced could solve many problems that the more traditional neural networks do have, especially when dealing with temporal data. Independently, *Echo State Networks* have been introduced by Jaeger [19] which is a computational paradigm based on similar principals. This area of neural computation id referred to as *reservoir computing*. However most of the work that has been done on this model has been done with non-numerical data and moreover, no training methods have been described to train the randomly connected column of neurons. The work for this Master's thesis, has been conducted at LIACS (Leiden Institute of Advanced Computer Science, Universiteit Leiden) under supervision of dr. W.A. Kosters and prof.dr. J.N. Kok. We will examine how we can use this model for numerical temporal data mining.

In Chapters 1–4 some background information will be presented about artificial neural networks, spiking neurons, dynamic synapses and the neural code. In Chapter 5 a new computational paradigm, introduced by Maass et al. [21], will described. Then in Chapter 6 we will describe some recent methods to use spiking neurons in feed forward neural networks. In Chapter 7 We will describe the results of our experiments. we will introduce a method to control the activity in a randomly connected column of neurons by adapting the dynamic synapses which form the connections. This is a highly useful property for a desired sparse code. We will introduce a method to minimize crosstalk between patterns and with that increase the efficiency of the network. In this chapter we will also introduce a new method to use spiking neurons as a collection of perceptrons with the purpose of decoding the liquid state and to produce an output value, that is especially well suited for a liquid state machine. It produces a better resolution than the existing P-delta rule by Auer, Burgesteiner and Maass [2] and it is less computationally expensive than Spikeprop by Bothe and Kok [7], since there is no need to calculate complex derivatives by means of back propagation. It is also noise

robust due to the fact the output is distributed over a number of output neurons. We will show that a column of neurons is able to contain information of past inputs by means of recurrent connections and that our newly introduced readout mechanism is capable of extracting information from this column of neurons, even about past inputs. We will conclude with chapter 8 where we will point out areas which are well worth investigating further.

Chapter 2

Artificial Neural Networks

In this chapter Artificial Neural Networks will be explained briefly. After this introduction it will be made clear why a new computational paradigm is necessary especially for temporal data analysis. A good introduction to the traditional neural networks can be found in [17]. A more elaborate work on the same subject matter can be found in [4].

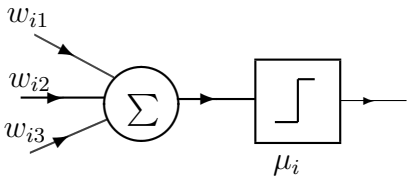
2.1 The Artificial Neuron

The general concept of an artificial neuron is an old one. In [21] it is stated that three generations of neurons exist. The first generation of neural networks consisted of *McCulloch-Pitts* neurons depicted in Figure 2.1, about fifty years old. This is the most simple model we know for a neuron. Let's consider a neuron i with threshold μ_i. The neuron i sends a binary output according to a weighted input. When the input is above the threshold μ_i the neuron sends out a binary 1, below the threshold a binary 0. This is called a Heaviside step function Θ. When a connection exists from neuron j to neuron i, a weight, w_{ij} is associated with this connection. The weight w_{ij} determines the strength of the connection from neuron j to neuron i, i.e., it regulates the amount of influence neuron j has on the output of neuron i. The output of a neuron i is denoted as V_i. The output of a neuron i with a connection from neuron j can be computed as:

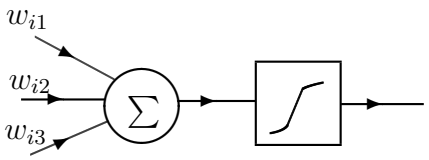
$$O_i = \sum_j \Theta_i(w_{ij}V_j - \mu_i). \tag{2.1}$$

Note that the output of neuron j serves as the input for neuron i and is being scaled by the weight, w_{ij}. The Heaviside step function is depicted in Figure 2.1(a). Although conceptually very simple these types of networks have

been successfully applied to many problems[4, 17, 35]. A network consisting of simple McCulloch-Pitts neurons with one hidden layer with sufficiently many neurons, can compute any function with binary output.



Schematic of a traditional McCulloch-Pitts neuron. This type of neuron is of the first generation and is binary. If the weighted input from neurons $j = 1, 2, 3$ for neuron i reaches the threshold i.e., $sgn(\sum_j w_{ij} S_j - \mu_i) > 0$, neuron i responds with a signal 'high'. If the weighted input stays below the threshold the output of the neuron responds with 'low'. The activation function for this type of neuron is depicted in Figure 2.1(a)



Schematic of a continuous valued neuron. This type of neuron is from the second generation. A neuron with continuous outputs uses a sigmoidal shaped activation function as depicted in Figure 2.1(b) and 2.1(c)

The second generation of neurons uses a continuous function that makes them suitable for analog in- and output, depicted in 2.1. For every input h to neuron i, an output O can be calculated according to an activation function $g(.)$. The output of a neuron i connected to neurons j can be computed as:

$$o_i = \sum_j g(w_{ij} V_j), \tag{2.2}$$

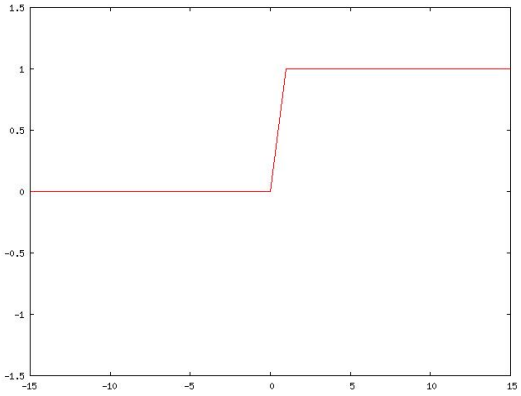
Where V_j is the output of neuron j. Common activation functions are the hyperbolic tangent or the sigmoidal function. Two examples of sigmoidal activation functions can be found in Figures 2.1(b) and 2.1(c). Networks of second generation neurons with at least one hidden layer with sufficiently many neurons can also compute any boolean function by applying a threshold function in the output layer. In fact they can do this with fewer neurons and therefore make them more more powerful than networks that consist of first generation neurons. In addition to that, networks consisting of second

generation neurons with at least one hidden layer with sufficiently many neurons can approximate any continuous function arbitrarily well. The first two generations are not believed to be biologically realistic.

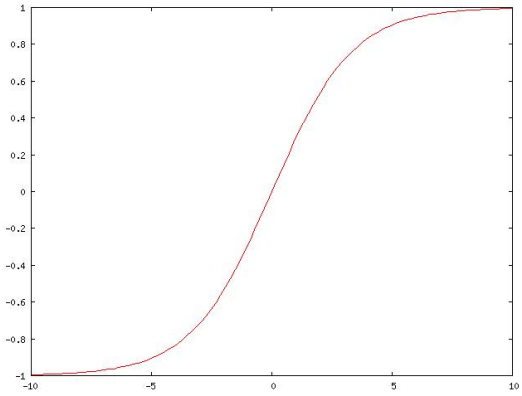
Real neurons use electric pulses to communicate information through the network instead of the input-output paradigm used by the first two generations. The third generation of neurons are the so-called *spiking neurons*. The output of spiking neurons is binary, a spike is emitted only when the inputs reach a certain threshold value. The difference with the first generation of neurons which are also binary, is that spiking neurons have complicated inner dynamics and the information spiking neurons provide is encoded in the timing of the spikes. One way to interpret the information provided by spiking neurons is to use the average rate at which it fires. The continuous output of the second generation of neurons can be interpreted as an average output. The average firing rate is then reflected by the continuous output of the neuron. Another big difference between the third generation and the first two generations is that the third generation doesn't necessarily produce an output for a particular input. Spiking neurons are described in detail in Section 3

2.2 The Feed-forward Network

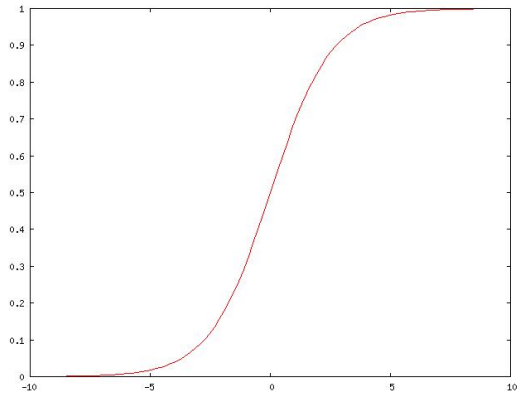
A feed-forward neural network is a network of artificial neurons which have only, as the name implies, forward connections between the artificial neurons. This type of network can only be used to approximate static functions. Static functions are functions whose output only depends on its current input and not on any of the previous inputs, in contrast to temporal data. The artificial neurons are usually called units. A feed-forward network has an input layer, an output layer and optionally several hidden layers which can be seen in Figure 2.3. Units in the input layer are denoted by ξ_k where $k = 1, 2, \ldots$, units in the hidden layer are denoted by V_j where $j = 1, 2, \ldots$ and the units in the output layer are denoted by O_i where $i = 1, 2, \ldots$. Between the layers connections exist which represent adaptable weights. When a neural network is trained to perform a specific task, the weights are adapted to better approximate the target response. There are several conventions for counting the layers in a neural network. In this work the input layer will not be counted. This way the number of layers corresponds to the number of layers of adaptable weights. Only having forward connections means that there can be no connection from a layer ℓ to a layer ℓ' with $\ell' < \ell$. The output units are denoted by O_i where $i = 1, \ldots$ denotes the specific unit in the output layer and the input units are denoted by ξ_k where $k = 1, \ldots$



(a) Heaviside step function used by the McCulloch-Pitts neuron



(b) Sigmoidal activation function,$tanh\beta x$ used by a second generation neuron. The parameter β regulates the steepness of the activation function. In this case $\beta = 0.4$.



(c) Sigmoidal activation function,$1/(1 + exp(-2\beta x))$ used by a second generation neuron. Again here $\beta = 0.4$ is used.

Figure 2.1: Activation functions used in the traditional neural networks. Note the difference in range for the two sigmoidal activation functions.

denotes the specific unit in the input layer.

2.2.1 Perceptrons

A *simple perceptron*, as depicted in Figure 2.2.1 is a feed-forward network which consists only of a single layer (an input layer does not perform any computation and is not counted). The ith output of a single layer network, called a simple perceptron, can be described by:

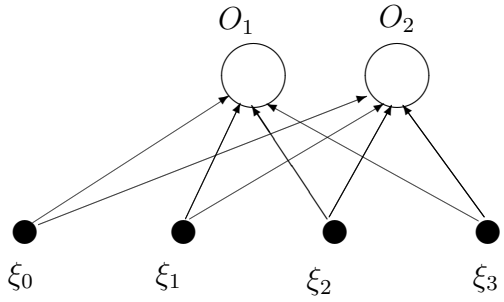


Figure 2.2: A simple perceptron consists of one layer. The input layer consists here of the filled circles and is not counted in the number of layers.

$$O_i = g(x_i) = g(\sum_k w_{ik}\xi_k), \tag{2.3}$$

where $g(x)$ is the activation function, x_i is the input to unit i, w_{ik} is the connection strength or weight from input unit k to output unit i. The activation function is usually taken to be nonlinear. There are several options but the most common function used is a non-saturated continuous sigmoid function. Two options for a sigmoid function are:

$$g(x) = \frac{1}{1 + exp(-2\beta x)} \tag{2.4}$$

as depicted in Figure 2.1(b) and

$$g(x) = tanh(\beta x) \tag{2.5}$$

as depicted in Figure 2.1(c). Here β is a parameter that regulates the steepness of the sigmoidal curve.

Often a network has an input node that is clamped to a fixed value, usually -1, which is called a *bias node*. In the case of a binary activation function a bias node represents a threshold implicitly, when the activation function is continuous the term "threshold" is not appropriate. The bias node gives the network an extra dimension or degree of freedom which makes the network computationally more powerful. If ξ_0 represents the bias node, the output O_i can be calculated as:

$$O_i = g(x_i) = g(\sum_{k=1}^{n} w_{ik}\xi_k - w_{i0}\xi_0), \tag{2.6}$$

where n is the number of input nodes. In the case of a binary activation function the threshold is then implicitly represented by $w_{i0}\xi_0$. When the activation function is continuous, $w_{i0}\xi_0$ is called a bias. In network diagrams the bias node is usually not shown but in most cases, unless explicitly stated otherwise present.

The perceptron is severely limited in its applications[17, 4]. Only linear separable problems can be solved with a perceptron. The computational power can be increased by adding hidden layers to the perceptron.

2.2.2 Multi-layer Networks

The limitations of simple perceptrons do not apply to multi-layered feed-forward networks. A simple perceptron can be transformed into a multi-layered feed-forward network by adding one or more hidden layers.A multi-layerd feed-forward network is also called a multi-layer perceptron.

2.2.3 Back-Propagation

Back-propagation is a method to change the weights in the network in order to learn a training set consisting of input-output pairs (ξ^μ, ζ^μ), where ξ_k^μ is the input to the network and ζ_i^μ is the target output of the network, for $\mu = 1, \ldots$ denoting different patterns. The learning rule will be explained by a two-layer network. For a more elaborate description please consult [10, 17, 4].

The lack of a learning algorithm for multi-layered networks has led to a decline of interest in artificial neural networks for quite some time. The back-propagation algorithm is rediscovered several times and basically is the well-known chain rule from mathematics applied to the simple learning rule used with perceptrons.

Output units will be denoted by O_i, hidden units by V_j, and input terminals by ξ_k, as depicted in Figure Ċonnections w_{jk} are present from the input units to the hidden units and connections w_{ij} from the hidden units to the output units. Different patterns are denoted by a superscript μ. The inputs can be binary or continuous valued. It has to be noted that this is somewhat an abuse of notation since the weights are not uniquely specified. We use that the output layer is referred to with subscript i, the hidden layer with subscript j and the input layer with subscript k, to distinguish the weights. However all subscripts run over $1, 2, \ldots$, which means that the true identifiers to denote the weights are not unique. However this is a generally accepted notation.

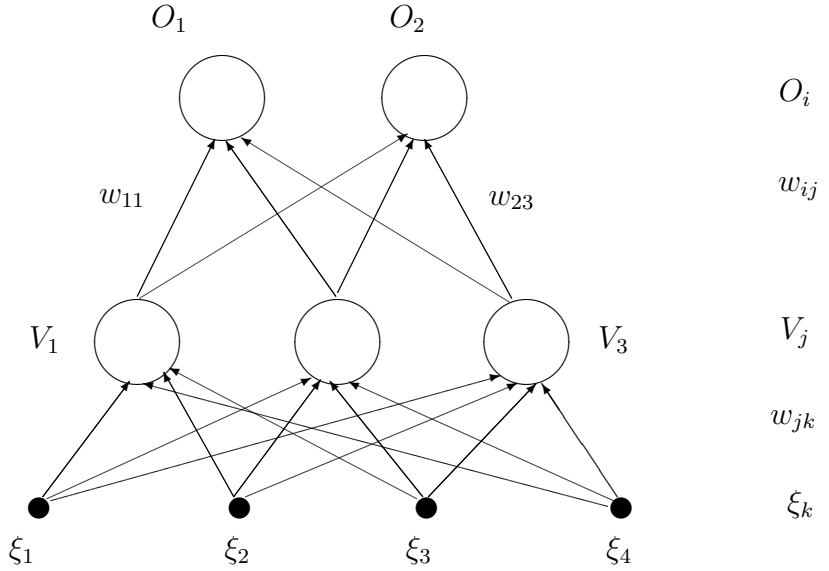


Figure 2.3: An example of a feed-forward neural network.

For a pattern μ, hidden unit j receives a net input of:

$$h_j^\mu = \sum_k w_{jk}\xi_k^\mu, \tag{2.7}$$

and produces an output by:

$$V_j^\mu = g(h_j^\mu) = g(\sum_k w_{jk}\xi_k^\mu). \tag{2.8}$$

This way the output unit receives an input:

$$h_j^\mu = \sum_j w_{ij}V_j^\mu = \sum_j w_{ij}g(\sum_k w_{jk}\xi_k^\mu), \tag{2.9}$$

and produces the final output:

$$O_i = g(h_j^\mu) = g(\sum_j w_{ij}V_j^\mu) = g(\sum_j w_{ij}g(\sum_k w_{jk}\xi_k^\mu)). \tag{2.10}$$

Determining whether or not the network produces the right output an error measure is used. This is referred to as the error or cost function:

$$E[W] = \frac{1}{2}\sum_{\mu,i}[\zeta_i^\mu - O_i^\mu]^2, \tag{2.11}$$

where W is the weight vector. The error measure becomes:

$$E[W] = \frac{1}{2}\sum_{\mu,i}[\zeta_i^\mu - g(\sum_j w_{ij} g(\sum_k w_{jk}\xi_k^\mu))]^2. \tag{2.12}$$

The error is a function of the weights. To change the weights into the right direction we can differentiate the error with respect to the weights. This is known as the gradient descent method. The amount of change of the weights after each update is regulated by the learning rate η. For the connections between the hidden and output units, the gradient descent rule gives:

$$\Delta W_{ij} = -\eta\frac{\partial E}{\partial w_{ij}} = \eta\sum_\mu[\zeta_i^\mu - O_i^\mu]g'(h_i^\mu)V_j^\mu = \eta\sum_\mu \delta_i^\mu V_j^\mu, \tag{2.13}$$

where

$$\delta_i^\mu = g'(h_i^\mu)[\zeta_i^\mu - O_i^\mu]. \tag{2.14}$$

Now we need to adjust the weights from the input units to the hidden units. These weights are more deeply embedded within the network and his is where we need the chain rule. We obtain:

$$\begin{aligned}
\Delta w_{jk} &= -\eta\sum_\mu \frac{\partial E}{\partial w_{jk}} \\
&= -\eta\sum_\mu \frac{\partial E}{\partial V_j^\mu}\frac{\partial V_j^\mu}{\partial w_{jk}} \\
&= \eta\sum_{\mu i}[\zeta_i^\mu - O_i^\mu]g'(h_i^\mu)W_{ij}g'(h_j^\mu)\xi_k^\mu \\
&= \eta\sum_{\mu i}\delta_i^\mu W_{ij}g'(h_i^\mu)\xi_k^\mu \\
&= \eta\sum_{\mu i}\delta_j^\mu\xi_k^\mu,
\end{aligned} \tag{2.15}$$

where

$$\delta_j^\mu = g'(h_j^\mu)\sum_i W_{ij}\delta_i^\mu. \tag{2.16}$$

Note that 2.16 allows us to determine the error(δ_j) for a given hidden unit V_j in terms of the δ's for the output units O_i that V_j is connected to. This is the way the error is propagated back through the network and hence the name back-propagation. A potential problem with back-propagation becomes clear now. High precision floating point numbers have to be communicated through the network, which is computational expensive and compromises the

precision when a large number of layers are used due to roundoff errors. We can see now that we need to communicate an error through the network with high precision. There are many different algorithms based on back-propagation to improve its performance and there is a lot to be said about the values of different parameters of the network.

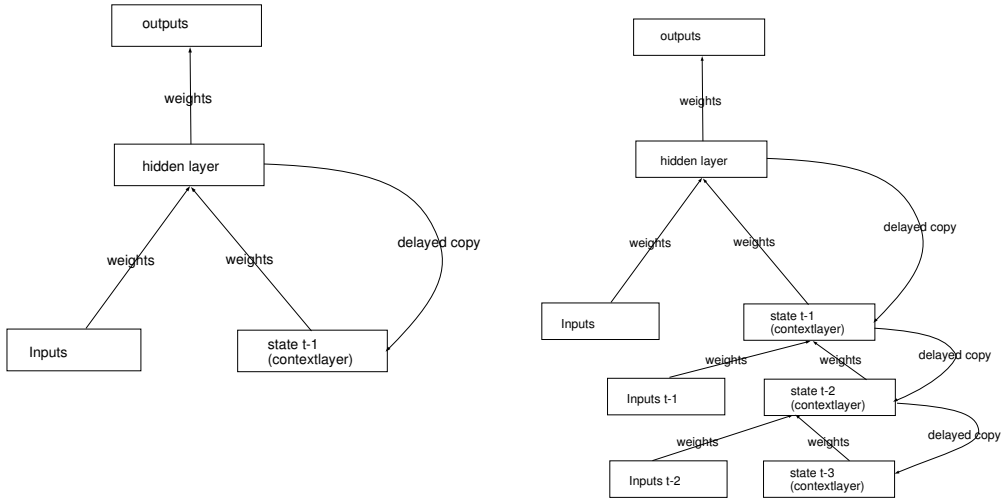
There are two common ways the network can be trained. One way is to first propagate all input-output pairs through the network and accumulate the weight changes by adding up the $\delta's$. Then after all patterns have been presented the weights are updated. The second way is incremental mode where after each input-output pair the weights are updated. It is a good idea to present the input-output pairs in a random order so the network will not learn the sequence of the data set.

2.3 Recurrent Neural Networks

In the multi-layer neural networks only forward connections exist. In a somewhat more complicated network connections backwards are allowed as well. This way we get loops in the network. These loops are called recurrent connections. There are no restrictions on the topology of the network except that there are clearly defined inputs and outputs. Any node can be connected to any node, including itself. In Section 2.2 it was stated that networks that only have forward connections are only suitable to approximate static functions. In the next section it will be explained that a recurrent neural network (RNN) does not have this restriction.

2.3.1 Increased Computational Power

A recurrent neural network can be used for temporal classification tasks that demand context dependent memory. That means this network can handle the notion of time, what a strictly feed-forward network cannot do. The ability for temporal classification comes from a so-called context-layer that acts as a memory for the network. Memory is achieved by storing the state of the hidden layer into the context layer and feeding those past observations back into the network as inputs. This context-layer, which holds the internal state (the state of the hidden layer) of time $t-1$, where t is the current time step, is then being used as input together with the input at time t which will be mapped to some desirable output as depicted in Figure 2.4. The input at time t is then being correlated with the state of the hidden layer at time $t-1$ and is thus also being correlated with the input at time $t-1$. This mechanism is also referred to as a *dynamic memory*. This way past observations are used



(a) Simple recurrent neural network (b) Unfolded recurrent neural network

Figure 2.4: Two recurrent neural network architectures. In (a) the state of the hidden layer is copied to the context-layer one time step later, a delayed copy. In (b) there are multiple context-layers for increased temporal capacity. With more context-layers it is possible to make correlations between data points that are further separated.

in the present observation, which means that the internal state is not only dependant on the current input but also on the previous internal state. This translates into a correlation between earlier inputs and the current input by reflecting previous internal states in the context of the current input, to form the current internal state. Since both the previous internal state and the current input are mapped to a desirable output the network is able to correlate between different inputs and thus encode temporal structures. The context layer is just another hidden layer whose input comes from the hidden layer and whose output is the input to the hidden layer. In 2.4(b) this method is being extended. By using more context-layers, more temporal dependencies can be represented in the network and thus being used in the calculations. It is clear that for every extra context-layer the computational complexity increases significantly. There are many variations on these two architectures [18, 12, 36].

A toy example of a situation when the notion of time is necessary to approximate a system, is the next deterministic finite automaton (DFA). The DFA will output its state every time it makes a transition. Now for the same input the DFA will produce different outputs according to which state it is in. For example when a c has just been read, the output depends on the

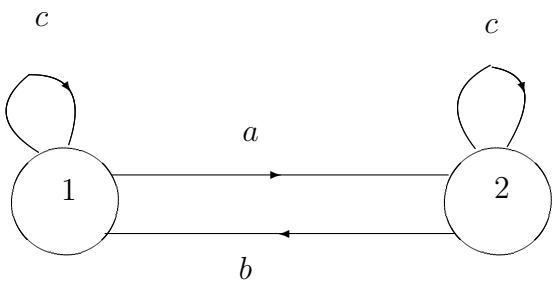


Figure 2.5: A deterministic finite automaton (DFA) which outputs its state after every transition is a toy example of temporal data.

previous read letter a or b. When we wish to make a neural network that can predict the next output of the system, it must have a memory which stores how past inputs affected the output and use this information when processing the current input. We must be able to store and use past information.

The use of a context-layer makes it possible that the network behavior is based on its past inputs, and that is why the pattern it recognizes must be seen as it happens in time. The network architecture of a RNN has a responsive mechanism to temporal structures which have an effect on processing. This is a key property of representing time implicitly. It is not an additional input, but rather it has an effect on processing the internal states of the system.

The calculations that can be done with a recurrent neural network are the same as a feed-forward neural network with the exception that standard feed-forward networks have no notion of time. The weights can be found by for example the algorithms called back-propagation through time (BPTT) [10, 18] or real time recurrent learning (RTRL) [41], which basically are extensions from the normal back-propagation algorithm. But again there are many variations of learning algorithms.

This type of network is particularly well-suited to handle noisy data. In general any temporal data mining technique handles noise gracefully. When memory is present, in this case implicitly, the ability to inspect the temporal neighborhood of the data point is present to derive a response considering the context of the data point.

Since many real world problems require the notion of time this type of network is commonly used in practice. Example applications are: speech recognition, stock market prediction, robotic control, music composition, etc.

2.3.2 Fluctuations in Error Prediction

As an example of the strength of recurrent neural networks and how the notion of time in a neural system is used, a very interesting result from the influential paper by Elman [12] will be described.

Fluctuations in the error prediction can be used to identify statistical properties in the data. A RNN was trained to predict the next character on an input character of a sequence of characters that was derived from a stream of sentences. All spaces and punctuation within the sentences were removed. The input of the network was a single character, the output of the network was the predicted next character in the stream. During training the output of the network was compared with the actual next character and the weights adjusted accordingly. Then during a test phase, the network was again presented with a stream of characters that represent sentences with all spaces and punctuation removed. The error was calculated by comparing the output character with the actual next input character.

The results are very interesting. Although for every beginning of a word the error was high, the error subsequently dropped for every new letter that was processed by the RNN in the original sentences. It is thus possible to perform word segmentation for concatenated words. This mechanism of fluctuations in error will be clarified in Section 2.3.3. The network had learned the words. This is quite remarkable since the network, at initiation, had no knowledge of language whatsoever.

The notion of time in this case can be traced back to the fact that when more of a particular word is seen, the more the network knows about it and the lower the error of prediction is. Every new input character is thus seen in the light of the previous input characters.

2.3.3 An Example of Temporal Processing

In [12] a nice example can be found as to how temporal structures can be found by a RNN. In one of the experiments described in [12] a string of 1000 characters is formed consisting of the letters b, d and g in random order. Each of these consonants were replaced by a sequence of characters according to the following rules:

$$\begin{array}{ccc} \text{b} & \rightarrow & \text{ba} \\ \text{d} & \rightarrow & \text{dii} \\ \text{g} & \rightarrow & \text{guuu} \end{array}$$

An initial sequence dbgbddg... would thus be converted into diibaguubadiidiiguu... making this sequence semi-random. The consonants are placed

	Consonant	Vowel	Interrupted	High	Back	Voiced
b	[1	0	1	0	0	1]
d	[1	0	1	1	0	1]
g	[1	0	1	0	1	1]
a	[0	1	0	0	1	1]
i	[0	1	0	1	0	1]
u	[0	1	0	1	1	1]

Figure 2.6: Encoding of the inputs. Each letter is encoded by a 6-bit vector. The underlying structure is extremely important for learning the structural dependencies (figure taken from [12]).

at random but a clear temporal pattern can be found in the vowels following the consonants. A recurrent neural network as described in Section 2.3 was trained to predict the next input. An input consisted of one letter from the sequence, it was then the task of the network to predict the next occurring letter in the sequence. The network consisted of 6 input units (each letter was encoded as a 6-bit vector, as can be seen in Figure 2.6), 20 hidden units, 6 output units and 20 context-units. During training 200 passes were made through the training sequence.

The error during part of this training phase can be seen in Figure 2.7. The target outputs can be seen in parenthesis. It can be clearly seen in Figure 2.7 that the error for predicting consonants is high and the error for predicting vowels is low. The network cannot possibly predict the consonants correctly since they were put in a sequence randomly. However not only is the network able to predict the right vowel after a consonant, the network also knows the number of vowels it can expect since the vowels follow a clear pattern. This is a very interesting feature. One might suspect that when a network knows how many vowels to expect the network can predict when the next consonant will be, however cannot predict which consonant.

Further analysis indeed revealed that the network was able to predict when the next consonant occurred. In Figure 2.6 the encoding of the letters can be seen. Note that the property "consonant" is the same for all consonants, property "high" may differ for consonants. When analyzing the prediction of the individual bits one and four representing the properties "consonant" and "high" respectively it can be seen that bit one is predicted with low error and the fourth bit is predicted with high error as depicted in Figure 2.8. From this it can be concluded the network can correctly predict

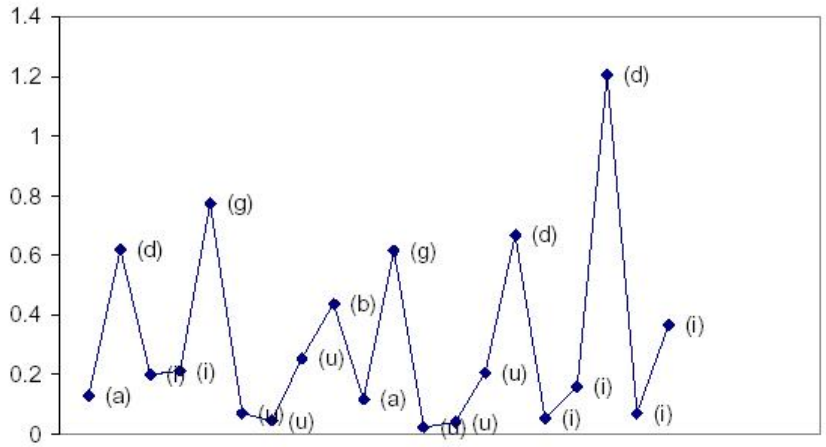
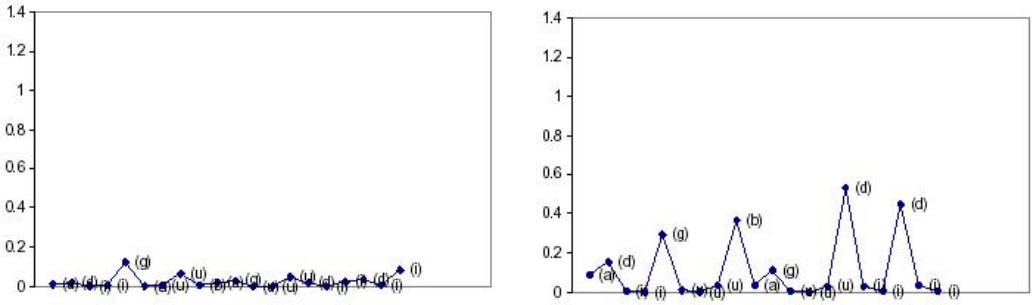


Figure 2.7: The mean squared error when predicting the next input. Note the large fluctuations in error. The network predicts with low error which vowel is the next input and how many vowels follow a consonant. The network predicts with high error the next consonant (figure taken from [12]).



(a) Error on prediction task bit 1, consonant property

(b) Error on prediction task bit 4, HIGH property

Figure 2.8: From these errors in predicting the individual bits of the encoded letters it is clear the network is able to make partial predictions. The network can correctly predict when the next consonant will occur but not which consonant (figure taken from [12]).

when a consonant follows but since the fourth bit is different for the different consonants the network cannot correctly classify which consonant.

The example just described is typical for temporal data. A feed-forward network could not learn the pattern of consonants and vowels. The reason is the sequential dependency of the input data. When the network "sees" the second vowel it needs to remember it has just seen a vowel to know the next

input will be a consonant. This can only be so if the network has some kind of memory and thus has some sort of recurrent connections. This experiment from which the results are plotted in Figure 2.8 also shows the network is able to make partial predictions when a complete prediction is not possible.

Chapter 3

Spiking Neurons

In this chapter some background will be given about spiking neurons. In Section 3.2 the general outline will be given of how a spiking neuron behaves, in Sections 3.5 and 3.6 two specific models of spiking neurons will be explained in detail. Since a full introduction could fill several books we keep it brief. A more elaborate introduction can be found in [15].

3.1 Introduction to Spiking Neurons

The Sigmoidal neuron described in Section 2.1 has proved to very successful in numerous applications and has gained a considerable amount of insight into the behavior of real neurons and connectionist systems [7, 4, 5, 17]. The Sigmoidal neuron models a rate output where the rate is a measure of the frequency at which the neuron fires electrical pulses for a certain input. However, the model where the average firing frequency is used as output, is infinitely simplistic compared to realistic neural behavior.

Real neurons do not have a simple input-output scheme as described in Section 2.2, but they behave more complicated. Real neurons emit spikes, short electrical pulses, and carry information in the timing of the spikes. It is generally believed that only the timing of the spike matters, there is no information in the shape of the electrical pulse. All spikes are similar in shape. Research provides evidence that spikes can be very reliable and precise. That is why more and more research is being conducted with regard to the computational properties of precisely timed spiking neurons. It has been shown in [22] that a spiking neural network can simulate arbitrary feed-forward sigmoidal networks which means that spiking neural networks can approximate any continuous function. In this work spiking neurons where the timing of the spikes are of importance will be used. This clearly gives a

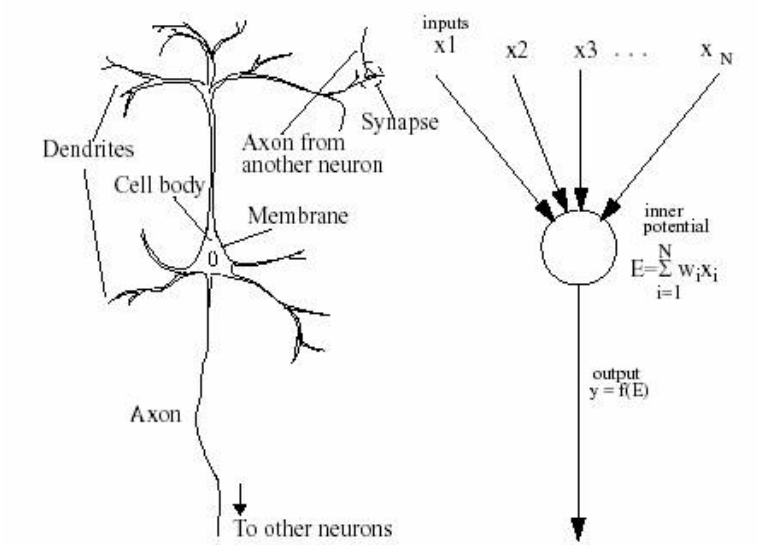


Figure 3.1: The three important parts of a spiking neuron. The soma (Cell Body) as central processing unit, the dendrites as inputs and the axon as an output device. The synapses form the connections between the axon and the dendrites. We can use the biological schematic as an example to create an artificial computational entity.

much more rich dynamic to a connectionist system than with a simple rate coding scheme.

The number of neurons a human brain is made up of is about 10 to 100 billion neurons. There exist many different types of neurons that function in different parts of the brain. Different types of neurons differ in the way they respond to inputs, the way they are connected and their function. A typical neuron has three distinct, clearly identifiable parts. These three parts are dendrites, soma and axon, and can be seen in Figure 3.1. Dendrites are the connecting wires so to speak. They have the role of carrying the input to the soma. The soma is the central processing unit. This unit performs a non-linear processing step which gives the neuron its functionality. When the inputs amount to a certain threshold the neuron emits a spike. This spike is carried over the axon to other neurons. The axon can be seen as the output device.

Another important anatomical part of the neuron is the synapse, the junction between two neurons. At the synapse the postsynaptic dendrite and the presynaptic axon meet. When a neuron fires it is said to be a presynaptic neuron. The neuron that receives the spike is called the postsynaptic neu-

ron. When a spike travelling across a presynaptic axon arrives at a synapse, a complicated bio-chemical process is started. Neurotransmitters will be released from the presynaptic terminal and will be received by the postsynaptic receptors. A single neuron is connected to a bewildering amount of other neurons, around 1000–100000 in number. Most of these connections are in the immediate neighborhood although some connections can stretch to several centimeters in the surrounding area.

3.2 Neural Dynamics

Many experiments have been performed to capture the dynamics of real neurons. The membrane potential can be measured with an intracellular electrode. By measuring this membrane potential in reaction to (artificial) stimuli from outside, a great deal is known about the dynamics of neurons. Although much is learned from experiments like this, many properties of the functioning of single neurons still remain elusive. That makes understanding a connectionist system rather complicated. However by implementing artificial neural networks with simplified dynamics, much can be learned.

The resting potential of a neuron is around $-65mV$. A spike will typically have a duration of around $1-2ms$ and an amplitude of around $100mV$. When a spike arrives the resting potential is altered and will return to its resting potential over a certain time course. When the membrane potential is altered positively, the synapse is said to be *excitatory*. When the membrane potential is altered negatively, the synapse is said to be *inhibitory*.

As an example the time course of the membrane potential of a postsynaptic neuron i, u_i will be examined. When u_i is unaffected by other neurons its value will be the resting potential, u_{rest}. When presynaptic neuron j fires a spike which is received by the postsynaptic neuron i, the membrane potential of i will be affected. The details of how the postsynaptic potential changes differs per model in both complexity and realism. Two different models of spiking neurons will be described in Sections 3.5 and 3.6. Let's say neuron j fires at time $t = 0$. neuron i responds in the following way:

$$u_i(t) - u_{rest} = \epsilon_{ij}(t). \tag{3.1}$$

Where $\epsilon_{ij}(t)$ is the response of i to an incoming spike from a presynaptic neuron j at time t which defines the postsynaptic potential (PSP) depicted in Figure 3.2(A). Exactly how this response is defined differs per spiking neuron model. When $\epsilon_{ij}(t) > 0$ the postsynaptic potential is excitatory (EPSP), when $\epsilon_{ij}(t) < 0$ the postsynaptic potential is inhibitory (IPSP). In a typical situation a neuron i receives input spikes from several presynaptic neurons

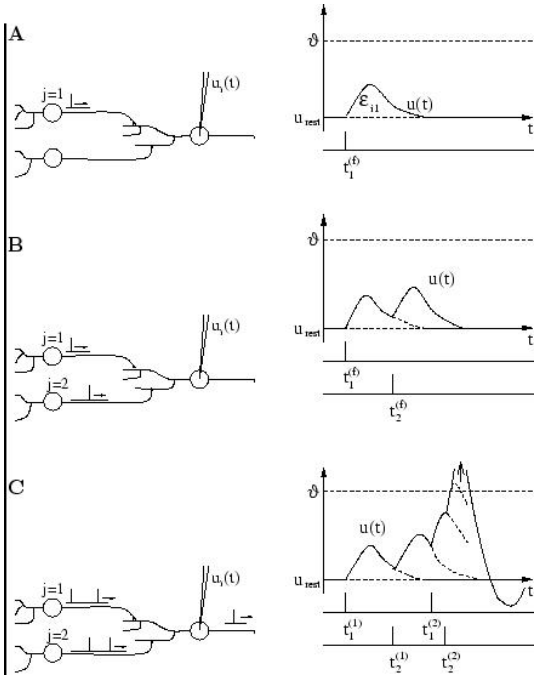


Figure 3.2: The complicated dynamics of a neuron receiving several spikes (Figure taken from [15]).

$j = 1, 2, \ldots$. The nth spike time of neuron j is denoted as $t_j^{(n)}$. When presynaptic neuron $j = 1$ fires spikes $t_1^{(1)}$ and $t_1^{(2)}$ which will be received by the postsynaptic neuron i, the postsynaptic potential ϵ_{i1} characterizes the response of neuron i to each spike, as can be seen in Figure 3.2. Typical for spiking neurons is that there is not necessarily an output of the neuron receiving incoming spikes if the threshold is not reached, and the membrane potential slowly returns to its resting state when no more spikes are received. An example can be seen in Figure 3.2(B) when another neuron $j = 2$, fires spikes $t_2^{(1)}$ and $t_2^{(2)}$. The postsynaptic potential responds linearly to input spikes and is about the sum of the individual PSP's,

$$u_i(t) = \sum_j \sum_f \epsilon(t - t_j^{(f)}) + u_{rest}. \tag{3.2}$$

Typically a PSP is around $1mV$.

The membrane potential stops to respond linearly when a threshold ϑ is reached, depicted in Figure 3.2(C). The potential will then briefly peak to an amplitude of about 100 mV which is called an action potential. This action potential is propagated through the axon and will reach other neurons as

an incoming spike. After the pulse the potential will fall beyond the resting potential, the so called *after potential*. This phenomenon of falling beyond the resting potential is called *hyperpolarization*. Typically 20–30 presynaptic spikes have to arrive before an action potential is triggered. Although a neuron cannot fire two spikes immediately after each other, due to the after potential, spiking neurons can fire spikes in succession. Successive spikes from the same neuron form a spike train.

Many different models exist to simulate the behavior of real neurons. The most detailed and most complicated is the *Hodgkin-Huxley* model which is derived from measurements of neurons of the giant squid. In this model several ion concentrations are realistically modelled to simulate the change in membrane potential. However this model is difficult to implement and computationally expensive. Another well known model is the *spike response model*, which is easier to implement.

The most widely used spiking neuron model is the *integrate and fire* model. It is a simpler model that still captures the most important features of spiking neurons. Simplified versions of these two models, the spike response model and the leaky integrate and fire model, will be used for the simulations in this work.

3.3 Synaptic Heterogenity

In Section 3.1 the three most important parts of a real neuron were described. In this section we'll look into more detail on the workings of the synapse.

The synapse is a connection between two neurons, and does not just represent a static weight between two neurons. It has been shown that synapses behave highly dynamic [23, 16, 25]. A presynaptic spike is communicated to a postsynaptic neuron through the synapse. Synaptic vesicles in the axon terminal, transfer neurotransmitter through the synaptic cleft to receptors located at the membrane of the dendrites of the postsynaptic neuron. This is illustrated in Figure 3.3. The amplitude of the postsynaptic potential depends strongly on the time since the last spike was received, which has an influence on the amount of available neurotransmitter, and the specific properties of a synapse. These two factors determine the internal state of a synapse. In [25] it is suggested that every synapse is possibly unique and that different types of target neurons underly differences in the synaptic properties. Through this heterogeneity of synapses, differential signaling emerges. Differential signalling means that, due to multiple synaptic representations, different synapses respond with different sequences of amplitudes to the same presynaptic spike train. This allows each postsynaptic neuron to interpret

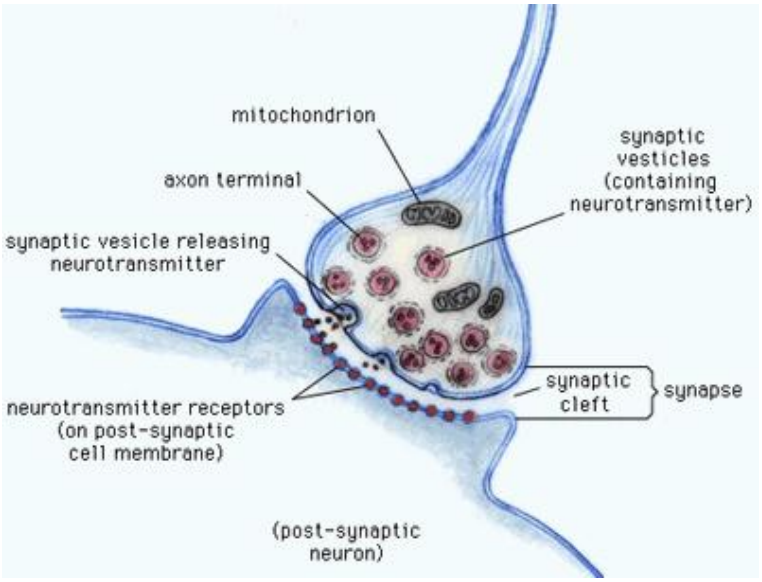


Figure 3.3: The junction between two neurons in close-up: the synapse (Figure taken from `http://www.bomi-1-gezondheid.com/Lichaam_en_geest/electro_biochemie_gevoel.htm`).

the same presynaptic spike train in a distinct fashion. The experiments conducted by [25] showed that the same spike train indeed induced different reactions in different types of neurons. Therefore they suggest that this differential signalling is a key mechanism for neural information processing.

In [23] it was shown that internal dynamic state of a synapse may serve as a transient memory buffer, that stores information about the most recent segment of the spike train it has just seen. These results hold great promise for the use of synaptic dynamics in temporal pattern recognition.

In [26] the heterogeneity of synapses was approached with the focus of the properties of an individual synapse. They defined an optimizing criterion for a spike train that will lead to a maximum response of the postsynaptic target. The resulting spike trains were interestingly enough common found spike trains patterns in biological networks of neurons. This inverse approach where the spike train is adjusted to the synapse, has lead to the idea of *preferential addressing* where the same presynaptic neuron can, by means of its firing behavior, target a specific postsynaptic neuron. This way temporal firing patterns encode information in the delays of the spike train that is decoded buy the dynamic behavior of the synapses.

Research has shown that the heterogeneity of synapses does not just con-

tribute to interpreting incoming information but retaining information from the recent past as well. From a computational point of view these results could have a great impact on the way temporal patterns could be processed.

3.4 Synaptic Dynamics

We use the model suggested in [23] for the synaptic dynamics. The properties and thus the specific behavior of a synapse is specified by four parameters $A > 0$, $U \epsilon [0, 1]$, $\tau_{facil} > 0$ and $\tau_{depr} > 0$. A can be seen as the weight of the connection between the pre- and postsynaptic connection, more precisely: A is the total amount of synaptic efficacy available. U represents the fraction of resources (neurotransmitter vesicles) that is used for a single spike. The parameters τ_{facil} and τ_{depr} represent the time constant of recovery from facilitation and the time constant of recovery from depression respectively. Facilitation means an increase in total efficacy in response to a presynaptic spike and depression means that synaptic resources are getting depleted and induce a less string postsynaptic potential.

The amplitude of a postsynaptic potential is calculated according to recursive Equations 3.3, 3.4 and 3.5. The total synaptic efficacy A_n, for the nth spike of a presynaptic spike train is calculated by:

$$A_n = A \cdot u_n \cdot R_n, \tag{3.3}$$

where the variable R_n is the fraction of synaptic efficacy available for the nth spike, and u_n represents how much of the fraction of available synaptic efficacy is actually used for the nth spike. Thus $u_n \cdot R_n$ models the fraction of synaptic efficacy that is used for the nth spike. u_{n+1} is calculated according to:

$$u_{n+1} = U + u_n(1 - U) \exp(\frac{\Delta t_n}{\tau_{facil}}), \tag{3.4}$$

where Δt_n is the time that passed since the nth spike. The factor u_n varies between U and 1 and, the initial value $u_1 = U$ and after every spike $n = 1, 2, \ldots$ the value u_n is increased ad recovers back to the value of U because of the term $\exp(\Delta t_n / \tau_{facil})$. The variable R_{n+1} is calculated according to:

$$R_{n+1} = 1 + (R_n - R_n u_n - 1) \exp(\frac{\Delta t_n}{\tau_{depr}}). \tag{3.5}$$

Note that the parameter R varies between 0 and 1. The initial value $R_1 = 1$ and is decreased after every spike. The term $\exp(\Delta t_n / \tau_{facil})$ causes R to recover back to 1.

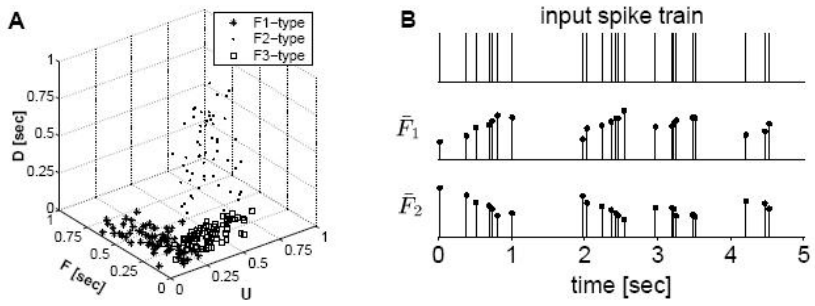


Figure 3.4: **A**) the parameter space for the parameters U, D and F is shown. Different combinations of values will produce different behavior of the synapse. **B**).Different types of synapses lead to different outputs on the same input spike train. (Figure taken from [26]).

This model of the internal state of a synapse has the effect that the first spike in a spike train causes a maximal postsynaptic potential. After that, part of the available resources to transmit incoming spikes have been used and so there will be less neurotransmitter to communicate the next spike in close succession to the postsynaptic neuron, with the result that the post synaptic potential will have a smaller amplitude. This phenomena is called depression. On the other hand there is a facilitation factor. The amount of the available neurotransmitter that is actually used to communicate a spike to a postsynaptic neuron depends on the facilitation parameter u. This regulates how fast after an incoming spike, the maximum amount of neurotransmitter used for a single spike, U will be used again. The initial values of the parameters for this model can have a severe impact on synaptic information transmission and thus on the behavior of a postsynaptic neuron as reported in [25] as a result of experiments on real neurons.

In [16] inhibitory synapses were investigated and it was found the synapses could be grouped into three major classes called facilitating ($F1$), depressing ($F2$) and recovering ($F3$). The parameters U, D and F determine to which class a synapse belongs. In Figure 3.4A the parameter space that gives rise to the classification can be seen. The different classes of synapses behave quite differently on the same input spike train as can be seen in 3.4B.

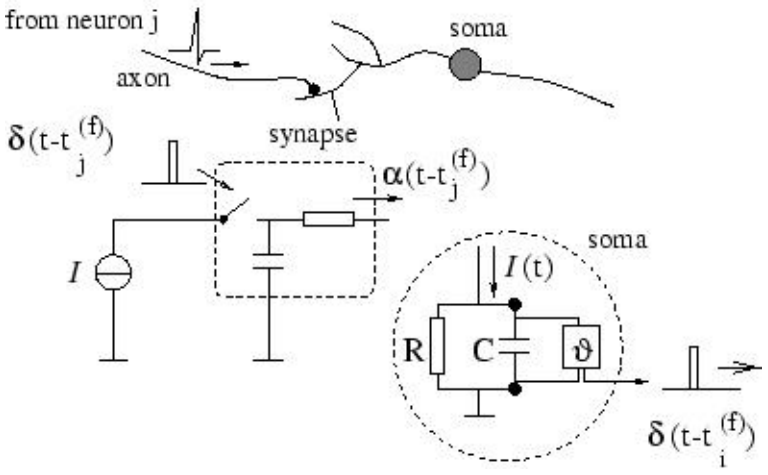


Figure 3.5: Schematic on which the leaky integrate and fire model is based. The main circuit is inside the dashed circle. The RC circuit is charged by a current $I(t)$. When the voltage $u(t)$ reaches the threshold ϑ, an output pulse is emitted. The circuit inside the square circuit models a synaptic connection.

3.5 The Leaky Integrate and Fire Model

The most widely used model is the leaky integrate and fire model. The model is simple enough for implementation but still captures the essential dynamics of real spiking neurons. The integrate and fire model is based on a circuit which consists of a capacitor C parallel to a resistor R and is driven by a current as can be seen in Figure 3.5.

The current I at time t therefore consists of $I(t) = I_R + I_C$, where I_R is the resistive current, and the component I_C charges the capacitor C. I_R can be calculated by Ohm's law $I_R = u/R$ where u is the voltage across the resistor. The capacity is defined as $C = q/u$ where q is the charge and u is the voltage. From these formulas we can derive the formula for the current:

$$I(t) = \frac{u(t)}{R} + C\frac{du(t)}{dt}. \tag{3.6}$$

This formula is multiplied by R and this way the time constant $\tau_m = RC$ is introduced. This produces the standard form:

$$\tau_m \frac{du(t)}{dt} = -u(t) + RI(t), \tag{3.7}$$

where u is the membrane potential and τ_m the membrane time constant of the neuron. The membrane time constant τ_m regulates the speed at which

the voltage of the neuron decays, i.e., leaks. In integrate and fire models the membrane voltage is not described explicitly but by the way it changes to stimuli. When the membrane potential reaches a certain threshold ϑ the membrane potential is reset to $u_r < \vartheta$. The leaky integration and reset are typical characteristics for the leaky integrate and fire model.

In integrate and fire models the form of an action potential is not explicitly defined. The fth action potential emitted by neuron j is characterized by a firing time $t_j^{(f)}$ which in turn is defined by the time the membrane potential reaches the threshold ϑ:

$$t_j^{(f)} : u(t_j^{(f)}) = \vartheta \quad and \quad \frac{du(t_j^{(f)})}{dt} > 0. \tag{3.8}$$

The second term means that the threshold has to be reached from below in order for neuron j to emit an action potential. When the threshold is reached and a spike emitted the membrane potential is reset to a new value $u_r < \vartheta$:

$$\lim_{t \to t_j^{(f)}, t > t_j^{(f)}} u(t) = u_r. \tag{3.9}$$

Usually the membrane potential is reset below the resting potential. For $t > t_j^{(f)}$ the membrane potential is again calculated as Equation 3.7. Because of the reset mechanism the membrane potential will never sustain a value larger than the threshold and so the condition $du(t_j^{(f)})/dt > 0$ from Equation 3.8 can be dropped. Some versions also incorporate an absolute refractory period. When the membrane potential reaches a certain threshold and fires at time t^f, the dynamics is interrupted for a period of Δ_{abs} and integration is restarted at $t^f + \Delta_{abs}$ with initial condition u_r.

3.5.1 Implementing the Leaky Integrate and Fire Model

When we want to use the leaky integrate and fire model we can use the nonlinear model. The nonlinear model is a generalization of the leaky integrate and fire model where the parameters are made voltage dependant. In real neurons the rate at which the membrane potential "leaks away" denoted by τ_m and the resistance denoted by R, are dependant on the membrane potential. The membrane potential of neuron i at time t is found by replacing Equation 3.7 by

$$\tau \frac{du_i(t)}{dt} = F(u_i) + G(u_i)I, \tag{3.10}$$

where $F(u_i)$ is a voltage dependant decay constant for neuron i and $G(u_i)$ is a voltage dependant input resistance for neuron i which comes down to:

$$\frac{du_i(t)}{dt} = \frac{1}{\tau_m(u_i)} + \frac{R(u_i)}{\tau_m(u_i)} I_i(t), \tag{3.11}$$

where u_i is the membrane potential of neuron i at time $t-1$. The current I at time t for a neuron i is found by evaluating the stimulation by other neurons connected to i. When a presynaptic neuron j fires at time $t_j^{(f)}$, a postsynaptic neuron will be influenced by the current of this spike over a time course $\alpha(t - t_j^{(f)})$. We can then find the total input current to neuron i according to

$$I_i(t) = \sum_j w_{ij} \sum_f \alpha(t - t_j^{(f)}), \tag{3.12}$$

where w_{ij} is the strength of the connection from neuron j to neuron i. A common way to define the postsynaptic current from Equation 3.6 $\alpha(s)$ is by

$$\alpha(s) = \frac{q}{\tau_s} exp(-\frac{s}{\tau_s})\Theta(s), \tag{3.13}$$

where $\Theta(s)$ is the Heaviside step function which returns 1 when $s > 0$ and 0 otherwise. The Heaviside step function basically ensures that a neuron cannot influence another neuron when the difference in time is negative, τ_s represents the time constant that regulates the decay of the post-synaptic current and q is the charge. A more realistic version would also incorporate a finite rise time τ_r with $0 < \tau_s < \tau_r$ of the postsynaptic current and a transmission delay Δ^{ax},

$$\frac{q}{\tau_s - \tau_r} \left[exp\left(-\frac{s - \Delta^{ax}}{\tau_s} \right) - exp\left(-\frac{s - \Delta^{ax}}{\tau_r} \right) \right] \Theta\left(s - \Delta^{ax}\right). \tag{3.14}$$

The integrate and fire model used for our experiments uses Equation 3.11 with a constant τ_m and a constant R non-dependant on the membrane potential. To calculate the synaptic current Equation 3.13 is used.

3.6 The Spike Response Model

The spike response model will now be briefly explained. First a simplified version will be described. In Section 2.2 a somewhat modified model of this simplified version will be used. Then some phenomena that are missing from the simplified version of the spike response model will be described and how they are incorporated in the more complicated version of the spike response model. For a more elaborate description please consult [15, 14].

3.7 The Simplified Spike Response Model

The major difference between the spike response model (SRM) and the integrate and fire model is in the formulation of the equations. Integrate and fire models are usually expressed in terms of differential equations, the SRM expresses the membrane potential as an integral over the past. In addition, the threshold ϑ is not fixed but may depend on the time since the last spike fired, $t - t_i$ where t is the current time. There is also a clear relation between the integrate and fire model and the SRM model. The SRM model can be mapped to the integrate and fire model as a special case [15].

The spike response kernel has so called responsive kernels that describe how the membrane potential is changed by incoming spikes. In the simplified version there are two kernels: η that describes the form of the spike and the spike after-potential, and the kernel ϵ_{ij} that describes the response of neuron i to presynaptic spikes from a neuron j. The membrane potential of neuron i is described by:

$$u_i(t) = \eta(t - \hat{t}_i) + \sum_j \sum_f \epsilon_{ij}(t - t_j^{(f)}) + u_{rest}, \tag{3.15}$$

where $\hat{t}_i = max\{t_i^{(f)} | t_i^{(f)} < t\}$, the last firing time of neuron i. just like in the spike response model a neuron i fires whenever the threshold is reached from below:

$$u_i(t) = \vartheta \ \ |\text{and}| \ \ \frac{du(t_j^{(f)})}{dt} > 0 \quad \Rightarrow \quad t = t_j^{(f)}. \tag{3.16}$$

This is a simplified version that does not fully capture the rich dynamics of real neurons. The most essential limitations in this simplified spike response model is that all postsynaptic potentials have the same shape, regardless of the state the neuron is in. In addition to that, the dynamics of neuron i only rely in its most recent firing time $\hat{t}_i$. In the next section an example will be given of a postsynaptic potential which shape is dependant on the state the neuron is in. More detailed descriptions can be found in [15].

3.8 More Realism

The Spike Response Model is just like the nonlinear integrate and fire model, a generalization of the Leaky integrate and fire model. However, the nonlinear integrate and fire model has parameters that are voltage dependant, whereas with the spike response model the parameters depend on the time since the last spike. This dependence of the membrane potential on the last output

spike time facilitates the possibility to model refractoriness, temporary loss of responsiveness to incoming spikes due to recent activity, in terms of three components. First of all a reduced responsiveness after an output spike, second an increase of the threshold after firing and third a hyperpolarizing spike after-potential.

There are three important kernels that define the spike emission behavior and the response of the neuron's membrane potential to incoming spikes. The response of the membrane potential to incoming spikes is defined by the three kernels that are illustrated in Figure 3.6. The kernel η describes the form of the action potential and after-potential. Again the shape of the action potential does not contain information. The action potential is a function of the time since the last firing time $\hat{t}_i^{(f)}$. The kernel ε describes the time course of the response of the membrane potential to an incoming spike also called the postsynaptic potential. Note that the shape of the postsynaptic potential is dependant on two parameters, the time since the last spike emitted, and the time since the last incoming spike. When a neuron i has fired its last spike time is denoted as: $t_i = max\{t_i^f < t\}$, where t is the current time. The kernel κ describes the time course of the response to an input current pulse after a postsynaptic firing time t_i^f. The time course of the membrane potential of neuron i, u_i, is given by:

$$\begin{aligned} u_i(t) =& \eta(t - t_i) + \sum_j w_{ij} \sum_f \varepsilon(t - t_i, t - t_j^f) \\ &+ \int_0^\infty \kappa(t - t_i, s) I^{ext}(t - s) ds, \end{aligned} \tag{3.17}$$

where t_j^f are spikes from pre-synaptic neurons j, and w_{ij} is the strength of the connection between neuron i and neuron j. The last term, I^{ext}, is an external (background)current. The second term runs over all pre-synaptic neurons j and all pre-synaptic firing times t_j^f of neuron j. Note that all terms depend on the time of the last fired post-synaptic spike $\hat{t}_i$.

In Figure 3.6 the time course of the membrane potential u_i of neuron i is shown, and it can be seen exactly what the individual kernels do. Up to the firing time of neuron i $\hat{t}_i$ the responsiveness of kernel ϵ_{ij} can be seen. The responsiveness is due to the impinging spikes of other neurons j. When u_i reaches the threshold a spike is emitted. The shape of the spike is described by kernel η as a function of the time passed since the last spike. Note that the threshold is also changed at the time of the spike and moves back towards its original value as a function the time since $\hat{t}_i$. This is a way to incorporate a mechanism to make sure a neuron cannot emit two successive spikes

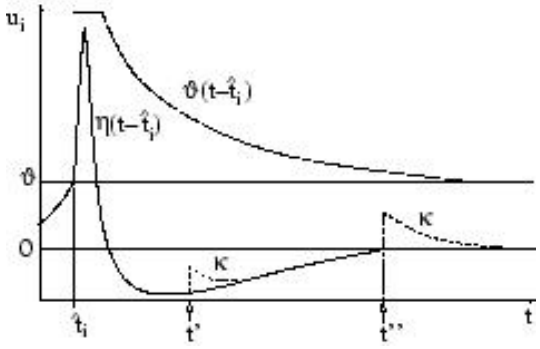


Figure 3.6: The responses of the different kernels to the membrane potential u_i of a neuron i. Note the three components that are used to model refractoriness. First, after a spike is emitted the threshold is set at a high value so a neuron can fire two spikes immediately after each other. Second there is a reduced response just when a spike has been emitted and third the hyperpolarizeing after-potential.

immediately after each other. When the membrane potential overshoots the resting potential, and reaches the so called after-potential, the kernel κ starts to respond. Note that shortly after a spike is fired the response of an incoming spike at time t' is a lot smaller than when an incoming spike arrives at time t''a long time after the spike was emitted. This phenomena is called refractoriness.

Chapter 4

Interpreting the Neural Code

An issue neuroscientists have been dealing with for quite some time is how to interpret the neural code. Through sophisticated equipment neuroscientists have been able to record the brain activity but interpreting this activity proved to be hard. For example, what information is contained in a pattern of pulses, how is this information communicated and how is this information decoded by other neurons and what are the building blocks of computation are key issues. Since we wish to build a computational model we need to look at some of these issues. First of all we need to interpret the output of our system, something neuroscientists have studied extensively. Second, we need to encode input data into a pattern of pulses. This latter issue has not been studied by neuroscientists. After all, they wish to understand the neural code, not produce a new one. In this section we will give a brief overview of how, according to the literature, information can be extracted from or encoded into spike trains.

4.1 Spike Trains

When the membrane potential reaches the threshold a neuron emits a short action potential. The time of this action potential is referred to as the firing time which coincides with the time the membrane potential crossed the threshold for the first time. The firing time of a neuron i is denoted as t_i^f where $f = 1, 2, 3, \ldots$ is the number of the spike. Formally, a spike train is defined as a sequence of pulses:

$$S_i(t) = \sum_f \delta(t - t_i^f)$$

where $\delta(x)$ is the Dirac function, $\delta(x) = 0$ for $x \neq 0$ and $\int_{-\infty}^{\infty} \delta(x)dx = 1$. This means that every pulse is taken as a single point in time.

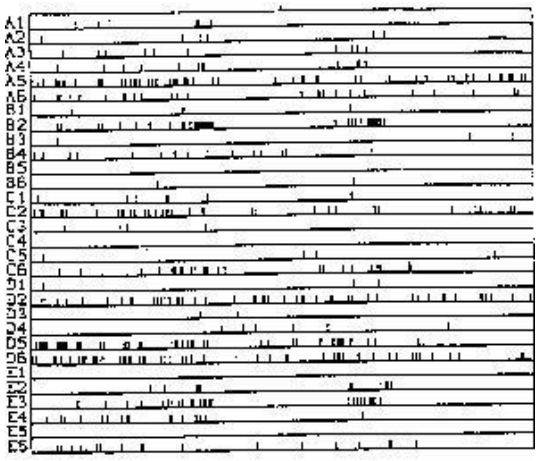


Figure 4.1: Spatio-temporal pulse pattern. Activity of 30 neurons, named A1-E6, are plotted along the vertical axis as a function of time. The firing times can be identified by the short vertical bars. A key issue in neuroscience is how to interpret such neuronal code. From [15]

4.2 Rate Codes

Traditionally it was believed that all of the relevant information was contained in the mean firing rate of the neuron. However, there are several different definitions of rate codes. One definition deals with averaging over time, another with averaging over several repetitions over the experiment or an average over a population of neurons.

Rate as a Spike Count

The number of spikes over a certain time window T, say $500ms$, is divided by the length of the time window. The mean firing rate is then defined as:

$$\nu = \frac{\eta_{sp}(T)}{T} \tag{4.1}$$

This definition comes from several classic experiments where a current is applied to a receptor. The most well known example is from Adrian [1] where a current is applied to a receptor in a muscle spindle. Other experiments have been conducted with the touch receptor in a leech [20]. From these experiments an observer could classify the the neuronal firing by a spike count measure, an increase in current resulted in an increase in spikes. It is questionable wether brains really use such an averaging mechanism. An

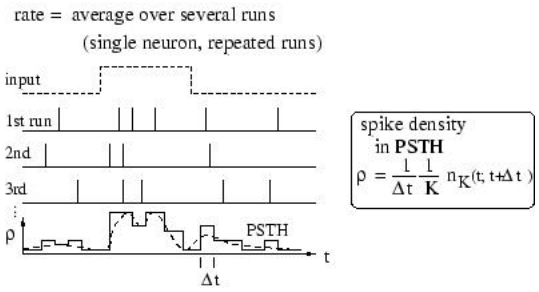


Figure 4.2: Example of an Peri Stimulus Time Histogram as an average over several runs over the experiment (From [15])

important counter arguments comes from behavioral experiments. Reaction times seem to be too fast to count the spikes and average them over some long time window. Note that this averaging mechanism is used with the traditional second generation neural networks.

Rate as a Spike Density

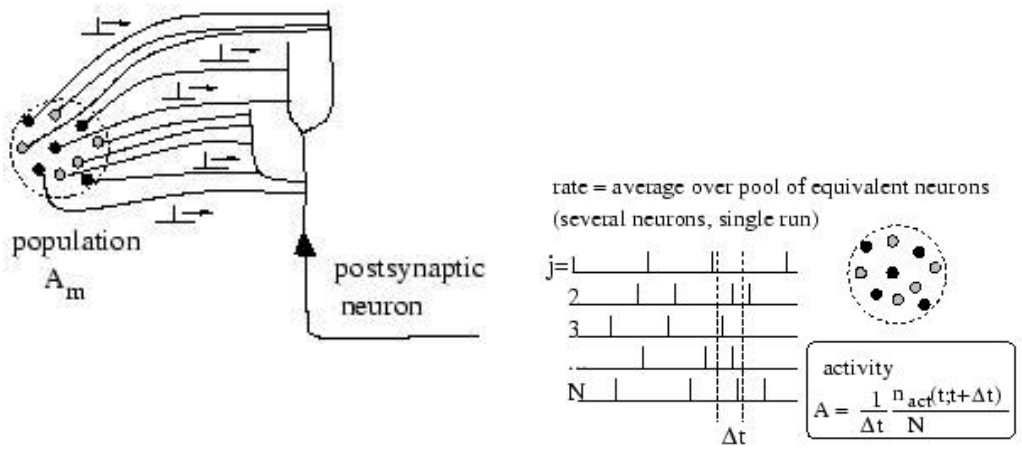
This definition of rate works for both stationary as for time-dependant stimuli. The same stimulation sequence is applied several times and reported in a so called Peri Stimulus Time Histogram(PSTH). The number of spikes between time t and Δt is then summed up over all experiments and divided by the number of K repetitions. This results in a number that represents typical activity of a neuron between time t and Δt. Spike density is then defined by:

$$\rho(t) = \frac{1}{\Delta t} \frac{\eta_k(t, t + \Delta t)}{K} \tag{4.2}$$

An important argument against this method is that it can hardly be biologically plausible. A classic counterargument against rate as a spike density is a frog that wants to catch a fly, it can not wait for the insect to fly repeatedly along the same trajectory. It could make sense when it involves a population of neurons whose response is averaged in a single run.

Rate as a population Activity

A good example of rate as a population activity is rate as a population activity are the columns of neurons with similar properties found in the primary visual cortex of various animals. Note that this is exactly the idea on which the liquid state machine(see chapter 5) has been based. Unlike



(a) A population of neurons (b) population activity

Figure 4.3: Rate as a Population activity does not suffer from the disadvantages of temporal averaging. Here a postsynaptic neuron receives input fro a pre-synaptic population of neurons. This population is able to reflect instantaneously changes in the input it receives.

temporal averaging at a single neuron level, the population of neurons is able to reflect changes rapidly with changing stimuli.
The activity of a population of neurons can be defined as:

$$A(t) = \frac{1}{\Delta t}\frac{\eta_{act}(t, t+\Delta t)}{N} = \frac{1}{\Delta t}\frac{\int_t^{t+\Delta t}\sum_j\sum_f \delta(t-t_j^f)dt}{N} \tag{4.3}$$

where N is the number of neurons in the population. Although rate codes have been successfully applied in numerous models, a common belief that not just the rate of a neuron contains information but also the timing of the spikes matter is rapidly emerging. For a survey see [8].

4.3 Population Coding

In this section we will look at how a the activity of a population of neurons can encode information. First it will be stated what is required in a computational model, then some neuroscience literature will be reviewed. A standard model for population coding will be explained as well as how this model was derived. Finally the model will be analyzed to establish how this model fits our requirements.

In the previous section 4.2 some neural coding schemes were described. These coding schemes are inefficient, slow and lack biological realism as well as the properties that would be required in a computational model. Neuroscience research suggests that information in the brain is encoded not by single cells but by populations of cells [28, 27]. In recent years the mechanism of encoding information by populations of neurons is better understood. Much research has been done to what information can be extracted from a population of neurons and how the brain is capable of interpreting such a neural code. Many sensory and motor variables are encoded by populations of neurons, such as the perception of movement where the angle of the moving object is encoded as to interpret the direction the object is moving in, or the perception of wind and the orientation in an environment among others. Each neuron in the population responds to a certain part of the domain of that encoded variable, the neuron is said to be sensitive to this part of the domain. This will be explained in detail in Section 4.3.1 The many neurons in the population have overlapping sensitivity profiles which means that many neurons are active at a given time. The advantage is that damage to a single cell will not have a critical effect to the information that is encoded.

Actually those factors that would make a neural coding scheme successful for computational purposes would also make that coding scheme biologically successful. First of all such a coding scheme needs to be noise robust. Noise is that part of the output that cannot be accounted for by the stimulus[28], either caused by stochastic fluctuations or by interference from other parts of the neuronal network. The purpose of a network is to make generalizations based on a training set and thus has to be able to correctly classify noisy data. Second, the scheme has to be efficient. In biological terms this means using a minimal amount of energy while functioning competitively in an environment. From a computational point of view it is required that a minimal amount of spikes has to be processed in order to classify noisy data correctly. The last but certainly not the least requirement is that the the coding scheme has to be able to represent a nonlinear mapping i.e. has a high enough resolution to encode a particular variable.

4.3.1 The Population Coding Model

A model[43] has been devised that uses bell shaped tuning curves to calculate the response of a neuron as can be seen in Figure 4.4 a. The model is inspired by measuring the responses of neurons during the perception of a motion stimulus in the visual area MT of a macaque monkey. Activity was monitored of neurons which are sensitive to visual movement in a small area of the visual field. The response of a neuron is defined as the number of action potentials

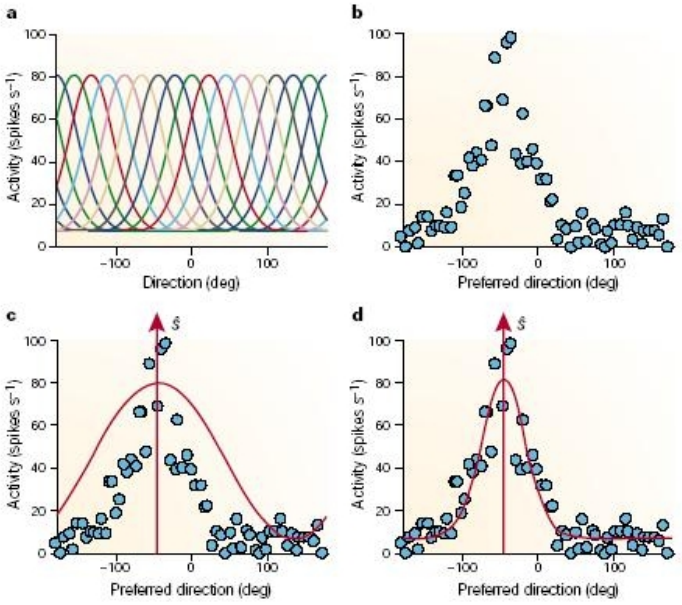


Figure 4.4: **a**) Tuning curves for a population of neurons. The tuning curves define the sensitivity for each neuron in the population to a perceived direction. **b**) Activity of each neuron plotted against the preferred direction. **c**) Template matching with the population vector method. **d**) Template matching with the maximum likelihood method

per second measured over a few hundred milliseconds during the presentation of the stimulus[43]. The response of a neuron can formally be described by

$$r_i = f_i(s) + n_i, \tag{4.4}$$

$f_i(s)$ describes the tuning curve and is the average response of neuron i to stimulus s and n_i is the noise in the response to stimulus s. The tuning curve is commonly written as:

$$f_i(s) = ke^{-(s-s_i)^2/2\sigma^2}, \tag{4.5}$$

where s_i is the mean of the gaussian which, when presented with a stimulus s where $s = s_i$, triggers the maximum response. The mean s_i of neuron i is often called the preferred direction. In the example of an object which is moving in a certain direction at an angle s, the stimulus $s - s_i$ represents

the angular difference at which the moving object is presented to neuron i with respect to the preferred direction of neuron i. This neuron will then respond with activity $f_i(s - s_i)$. Note that the inclusion of noise in Equation 4.4 makes the response non-deterministic. In computational models this is an undesirable property, however this gives us the opportunity to analyze the robustness of information encoding of a population and compare different readout methods.

4.3.2 Decoding a Population Code

The model of a population code as described in Section 4.3.1 realistically defines the response of a population of neurons as is experimentally recorded during in vivo experiments. Just as important is the question what information can be extracted from such a population of neurons. In this Section we'll assume a population P exists and responds as described in Section 4.3.1.

Noise

A population P of neurons responds with a noisy hill as can be seen in Figure 4.4. When the activity of each neuron is plotted against its preferred orientation the result is not a smooth hill. The presence of noise poses a difficulty. When a neuron responds with a certain firing rate, it is not known how much the stimulus contributed to this firing rate and how much is due to noise. Even for a computational model it is important to be able to deal with noise even though in the response of the neuron the amount of noise can be controlled and even reduced to zero with Equation 4.4. Noise can arise from noisy input data or an imprecise response of the population due to insufficient training or a difficult training set. To determine the output we could look at the neuron that has the maximum activity and assume that the estimated output is the preferred direction of the neuron. However this method is not noise robust and will lead to large fluctuations. It would be better to reduce noise by averaging over all neurons. To determine the exact output of P we would like to know the exact contribution of the stimulus to each tuning curve.

A common method for decoding is a bayesian model. In this case it is assumed that noise has a normal distribution with a mean of zero. The probability of a stimulus s can be determined by the mean responses $f(s)$ given the observed response r. The probability distribution $P(s|r)$ can then be determined which states how likely a stimulus s is for a given response r. The probability distribution $P(s|r)$ can be calculated as:

$$P(s|r) = \frac{P(r|s)P(r)}{P(r)}. \tag{4.6}$$

For a more detailed description see [28]. Once a probability distribution has been determined a value for the direction for each cell is the result. But only one estimate of s is needed. Several possibilities exist to select one estimate of s. The first method is to choose the estimate with the highest probability density $P(s|r)$. This is known as the *maximum a posteriori estimate*(MAP). Another option is to choose the direction that maximizes the likelihood function $P(r|s)$. This is known as the maximum likelihood estimate (ML).

Both ML and MAP require a tremendous amount of data and analysis before they can be put to use. The probability distributions of the responses r given a stimulus s have to be known for each neuron. This is the method used in neuroscience research. Often however, we wish ro read the output in one trial. In such a method the preferred direction of each cell is assumed to be known. Methods that rely on the preferred direction of a cell are called voting methods. The name arises from the fact that in these methods the activity of each neuron counts as a vote for the stimulus s. A well known voting method is the population vector estimator [13]. With this method each neuron is assigned a vector where the length of the vector is based on its activity and the direction of the vector corresponds with the value of the stimulus the neuron encodes. A weighted average over all the vectors gives the estimated stimulus. The population vector method is known not to give the optimal accuracy [28, 11, 31]. Whether a read-out method performs with optimal accuracy can be determined by a statistical construct called the Cramer-Rao bound. This bound produces the minimal trial to trial error at a certain noise level. Voting methods and ML are template matching procedures. Template matching procedures slide a response curve across the output of the population and for the best fit the peak position of the response curve is used as the estimate of the stimulus. In Figure 4.4c and d the templates of the population vector and the maximum likelihood can be seen. The difference in accuracy of the estimate lies in the template used. The template used in the population vector method is a cosine whereas the template used in the ML method is based on the tuning curves and is proven to be a better estimator based on this fact. In fact, the ML fit is often assumed to equal the Cramer-Rao bound.

A method to computationally determine the stimulus from a population of neurons is proposed in [31]. They investigate an optimal way to decode the information present in a population of neurons, and they show their method is equivalent to the ML method, thereby stating that their method is optimal. They propose to convolve the noisy response of the population with a filter

that is based on the average tuning curve of the population. The maximum of the filtered response is taken to be the estimated stimulus. This filter is called the matching filter and has the shape of the average tuning curve. They show that the standard deviation of the estimated stimulus approaches the cramer-Rao bound when the matched filter has the width that is the same as the response curves of the neurons in the receptive population. However the Cramer-Rao limit is only reached for small noise amplitudes and starts to stride away from the optimal bound for higher noise levels, which are stated to be non-realistically high.

A different method has been taken by [11] where a close approximation of a ML estimator is implemented in a biologically plausible network of neurons. Their network consists of a pool of neurons that have lateral connections. Such a network is said to be an ideal observer. they show that for neuronal noise independent of the firing rate the variance achieved by the estimator is equal to the minimum variance as is produced by the ML estimation. For neuronal noise dependant on the firing rate, such as generated in poisson-like distributions, the achieved variance is close to the ML estimation. The noisy hill that is the result of the activity in the population is being evolved into a smooth hill by a 2D-gaussian filter. The peak of the resulting smooth hill is taken as the estimated output value. Interestingly they managed to produce an optimal readout with a neural implementation. It is speculated that it is not unlikely the brain has implemented a maximum likelihood estimation through evolution.

4.4 Encoding Continuous Data into Spike Times

It was shown in [24] that spiking neurons are computationally more powerful than artificial sigmoidal neurons. However the encoding of information in the spike times is more complicated than is the case with the artificial neurons. Where a simple activation function translates an input value into an output value, spiking neurons use a differential equation to determine when to fire a spike. Somehow we need to encode the input information in a form we can feed into a spiking neural network. Since the only thing a spiking neuron can process are spikes from other neurons we have to translate the input data into spikes.

A common method for decoding a spike train is the *spike information interval coding*(SIIC). This method uses a convolution,

$$x(n) = \sum_{k=0}^{N} f(k)u(n-k)$$

where x is the reconstructed waveform, f is the convolution filter modelled after the post-synaptic potential and u is the spike train. This method is described in detail in [29], [34]. The idea is to use the form of the action potential to reconstruct the analog signal from a spike train. The *Hough Spiker Algorithm* is designed to do the exact opposite. A drawback with these methods is that, since they use the form of the action potential, these methods are only usable for waveforms. Another drawback is that for encoding an analog signal into a spike train we need a filter that is optimized for the particular function to be converted. That makes these techniques ill suited for a general temporal pattern analysis purpose. After all, we don't know in advance what the data looks like. We need a way to translate every data point in the input data into a unique spike train. No two numbers can be translated into the same spike train and in order to train the network we also need to be able to reverse the process. The output spike train needs to be translated into a real value. This mapping from the real data-points to the spike trains needs to be a bijection.

A widely studied method to use real numbers with spiking neurons is to use input neurons that use an activation function to translate the input into a firing rate [44]. These input neurons are so called receptive fields. The input variable is then distributed over a population of input neurons with graded and overlapping activation functions. This method of population encoding is used to produce a firing rate for every input neuron and thus encode the information distributed over the network.

It is well known by now that the information a spiking neuron encodes is not just by the rate that it fires at. An increasing amount of evidence reveals that information is also encoded in the timing of the spikes. This is the reason that the method of translating a real number by population encoding is being used to translate the data value into a firing time. This method was introduced by Bohte[6]. With their method an optimally stimulated neuron will fire at $t = 0$ and a minimally stimulated neuron will fire at $t = t_{max}$. A threshold can be introduced as a minimal output value of the gaussian i.e receptive field, that has to reached in order for the input neuron the receptive field is associated with, to fire. In [6] population encoding was used to increase the temporal distance between the data points, which will result in increased resolution once presented to a network of spiking neurons like the liquid of a liquid state machine.

In [6] it was found that it is least expensive to independently encode each of the input dimensions by a separate 1-dimensional array of receptive fields. This suggests that in advance the dimensionality of the 1-dimensional array has to be set. In [44] it was found that the encoding accuracy of a particular variable is increased when the dimensionality of the receptive fields is

increased and the activation functions of these input neurons are sharpened.

In [6] the activation functions used are Gaussians. First the data range is determined and for each input dimension n the minimum, Min_n and maximum, Max_n are determined. For the i^{th} neuron encoding a variable in dimension n the center c of the Gaussian was determined by

$$c = Min_n + \frac{2i-3}{2} \cdot \frac{\{Max_n - Min_n\}}{m-2} \tag{4.7}$$

where m is the number of neurons with Gaussian receptive fields used. The width σ of the Gaussians was set to

$$\sigma = \frac{1}{\gamma} \frac{\{Max_n - Min_n\}}{m-2}, m > 2 \tag{4.8}$$

Each receptive field is then defined by the following equation:

$$F\phi\left(\frac{(x-c)^2}{\sigma^2}\right) \tag{4.9}$$

where x is an input dimension and $\phi(z) = exp(-z/2)$.
In [6] it was stated that the the use of a mixture of receptive fields with varying receptive field widths greatly enhanced the level of detectable detail.

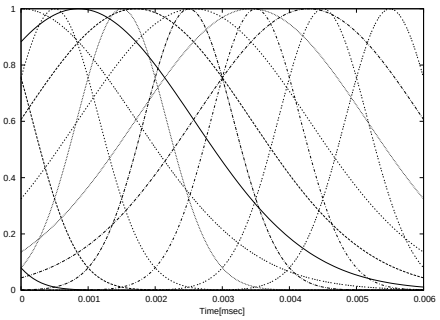


Figure 4.5: A sensor is made up of an array of receptive fields. Each receptive field is a gaussian defined by Equation 4.9. This way an input value is translated into an array of output values which represent firing times of the input neuron the sensor is connected to.

It is clear it is desirable that the distance between the data points is high. This will increase the resolution since the further the data points are separated in the encoding space the easier it will be for the neural network to discriminate between them.

The maximum resolution is determined by F. Every receptive field will have a range between 0 and F. With this encoding scheme for each data point we get $N \times m(n)$ values between 0 and 1. These values were then converted to delay times between $t = 1$ for a 0 and $t = 10$ for a 1. This type of coding has two desirable properties. It produces a sparse coding which allows only to process a small amount of active neurons. By encoding each variable separately coarse coding is achieved. This allows each variable to be encoded by an optimal number of neurons.

4.4.1 Inspecting Population Coding by Receptive Fields

In this section the usability of encoding real numbers into a vector of spike times, by means of population coding using multiple receptive fields with varying field widths, will be investigated.

To investigate the useability of this encoding technique we will encode some data into spike trains and analyze whether the results fit our goals. The data we will be using is simply the domain $[0 : 10]$ of real numbers with a resolution of 0.1. This gives a total of 100 data points. The data was normalized to values between 0 and 1. To encode this data an array of receptive fields is used. A receptive field is defined here by a neuron which has a sensitivity area in the form of a gaussian as described in Section 4.4.

Several criteria have to be met in order for this encoding method to be useful. First of all, the number of spikes and the interval these spikes are distributed on are an important factor for the interpretation of this spike train by the neural network. If not enough spikes are generated the concerning input will not be able to induce a spike at the postsynaptic neurons since the postsynaptic neuron's potential will not reach threshold . Secondly, if the spikes are distributed on a short interval then the learning function of a network will also be limited to a short interval and therefore will not be able to produce late firing times as the output. A network has to be as flexible as possible so it is important the responses of the output neurons can be adjusted over a wide interval. This requires an interval that is as long as the output domain. In [6] it was not defined how to convert the output of the gaussian function to a spike time other than that a lightly excited receptor was assigned a late firing time and a highly excited receptor was assigned an early firing time. The output values of the receptor are confined to $[0 : 1]$ and converted to a value that is within the defined spiking interval I. The interval I is the duration an encoded input value is being presented to the postsynaptic neural structure. We convert the result R of the receptor's sensitivity area defined by Equation 4.9 to a firing time as

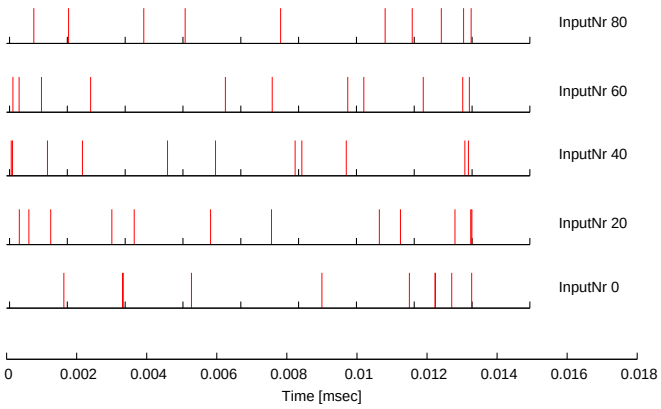


Figure 4.6: Few spikes are generated per input value.

$$T_{min} + (1.0 - R)(T_{max} - T_{min}), \tag{4.10}$$

where T_{min} is the minimum firing time and T_{max} the maximum firing time of I.

Another property we wish from the encoding mechanism is that a neuron can emit a spike train as a result of an input instead of a single spike time. This can be achieved simply by assigning several receptive fields i.e. gaussians to an input neuron. This will produce a spike per gaussian where the level of activation exceeded the threshold.

The most important property is to be able to regulate a certain amount of distance between encoded spike time vectors in order for a neural network of any kind to be able to discriminate between inputs. How much distance this has to be is impossible to state, however the importance here is it is possible to regulate and control the distance between inputs. When this is possible the discriminatory property of the real input values to be encoded can be increased at a temporal level. All the information in such an encoded vector of spike times is embedded in the time difference of successive spike times. The discriminatory property of the resulting spike trains will be investigated from the receiving neuron's point of view. The distance between two spike trains S_i and S_j will be defined as as the difference Δ_{S_i,S_j} over all the input neurons n in postsynaptic potential over time t and where $t \geq 0$ and $t \leq T_I$ where T_I is the length of the encoding interval. The discriminatory property

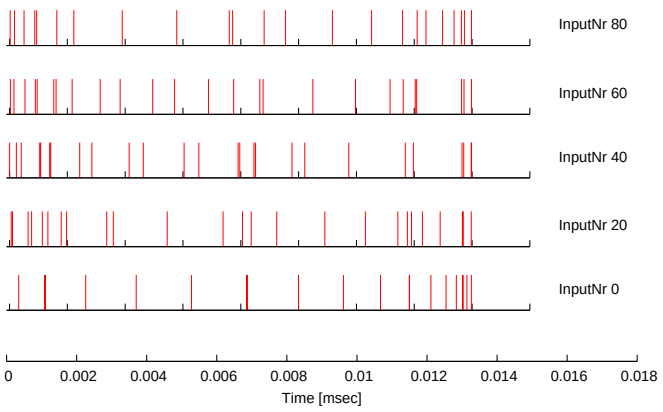


Figure 4.7: Many spikes are generated per input value.

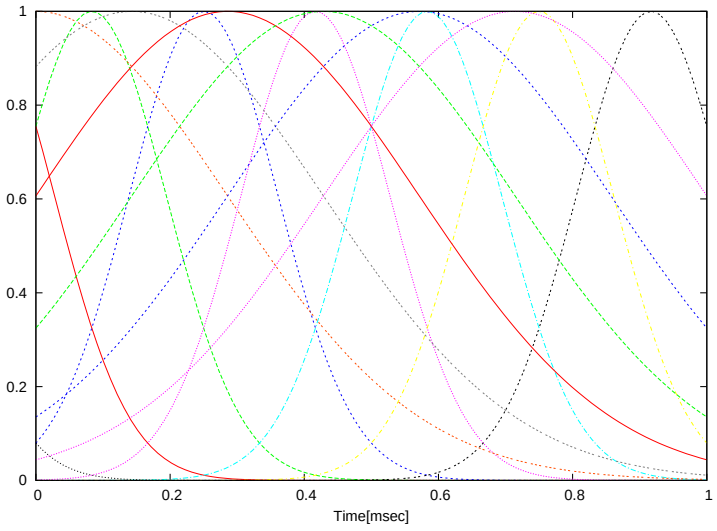


Figure 4.8: Narrow Gaussians produce few spikes.

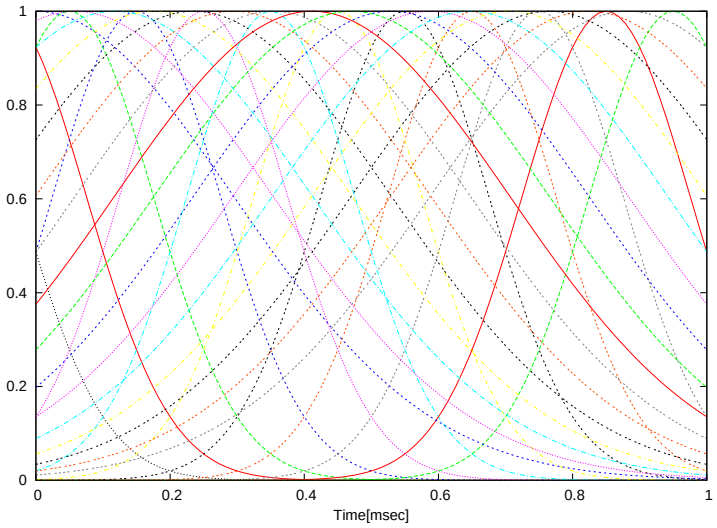


Figure 4.9: Wide Gaussians produce many spikes.

Δ_{S_i,S_j} will be defined as:

$$\Delta = \sum_n \sum_\ell |y(t - t_n^{\ell_{S_i}}) - y(t - t_n^{\ell_{S_j}})|, \tag{4.11}$$

where $y(t)$ is the postsynaptic potential and $t_n^{\ell_{S_i}}$ is the ℓth spike emitted by neuron n that belongs to spike train S_i.

4.4.2 Decoding a Spike Train Obtained by a 1-Dimensional Receptive Field

In Section 4.4 it was described how to encode an input variable into a spike train. In the field of numerical data mining recognizable patterns are sought in large amounts of data. When seeking for patterns in data one tries to predict the next input based on what is seen so far. A pattern can be found this way by correcting any error that is present in the output of the network. This implies that we also need a way to reverse the coding.

We need two important properties. It was already stated that a bijection is needed. Every input is to be encoded uniquely into a spike train and every spike train is to be decoded back into this same number without error. Another important property is the scaling in between two encoded spike trains. A small error in the output of firing times by the output neurons must give rise to only a small error in the predicted value presented by the

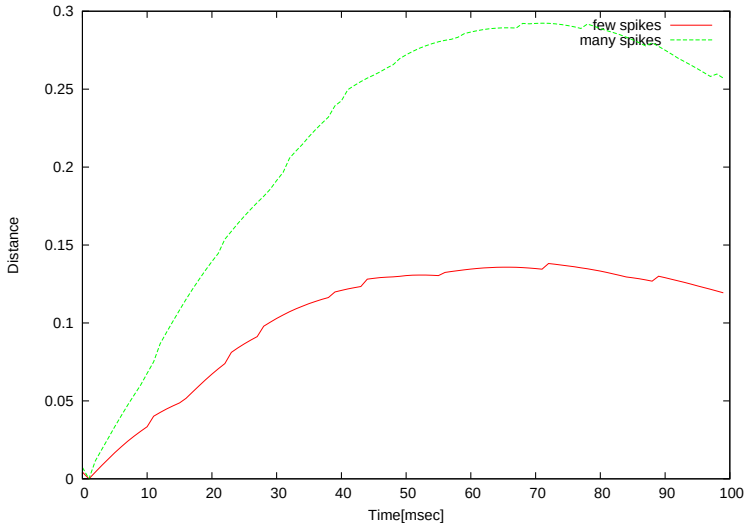


Figure 4.10: Distances for spike trains with different spike densities.

network.

Lets first look at a way to decode an encoded spike train. A spike train is encoded by a 1-dimensional array of receptive fields. Every receptive field yields a firing time and these firing times determine at which points in time the input neuron fires. The output of the network is of the same form. A 1-dimensional array of spike times. We must simply solve for every receptive field r the input value that produced firing time $t_r^{(f)}$. When we look at the equation for a single receptive field,

$$t_r^{(f)} = F\phi\left(\frac{(x-c)^2}{\sigma^2}\right) \tag{4.12}$$

we have to solve x for $t_r^{(f)}$. The derivation is as follows:

$$\begin{aligned} y &= \exp\left(\frac{-(x-c)}{2\sigma^2}\right) \\ \ln y &= \frac{-(x-c)}{2\sigma^2} \\ \ln y &= \frac{-x^2+2xc-c^2}{2\sigma^2} \\ \ln y \cdot 2\sigma^2 &- \left(-x^2+2xc-c^2\right) = 0 \end{aligned} \tag{4.13}$$

Equation 4.13 can simply be solved by the ABC-formula which yields the final Equation that will yield our answer.

$$x_{1,2} = \frac{-2c \pm \sqrt{2c^2 - 4 \cdot (-1) \cdot (-c^2 - \ln y \cdot 2\sigma^2)}}{-2} \tag{4.14}$$

The only problem is then that for a single firing time results in two solutions and only one is the correct solution which will yield our input. This problem can be solved by the fact that every receptive field produces two values, but every receptive field actually should produce the same value since they each decode the same input value. So by looking at which value is produced by every receptive field we know the correct input value we are looking for.

This yields the desirable property that no matter how many receptive fields we use to encode an input value, we only need to solve Equation 4.14 twice to find the encoded input value.

Chapter 5

Liquid State Machines

5.1 The Analogy

The common analogy of the inner workings of the brain is a pool of water. The relaxed state of the water is a smooth surface. When looking at this of the point of view of neural networks with an energy function this is the only attractor of the error landscape. When the surface of the water is disturbed this disturbance will fade out and the surface of the water will return to its resting state. As a computational device this has not much to offer. But in the event of a continuous stream of disturbances the water surface holds all information of present and past disturbances(inputs). This means that there are no stable states except for the resting state.

To use this as a computational device we need a medium that represents the water surface in the analogy just described. Since the state of this medium is a result of the past inputs we can use this to extract information and analyze some dynamical system. The perturbations in the medium must react to different inputs in different ways that is recognizable and must be non-chaotic. The medium can be any analyzable and excitable entity. However different mediums will have different characteristics. For example, the water surface will have strictly local interactions. A medium of neurons is an ideal medium since the structure and dynamics can be defined by properties such as the degree of connectivity, which will determine range of interactions and recurrent structure, and variety responses by the individual neurons which is determined by the definition of the neuron model.

The most important aspect is that in this computational paradigm computation is performed without stable states. This results in the ability to analyze rapidly changing inputs of a dynamical system.

5.2 The Computational Paradigm

The computational paradigm called "the liquid state machine" is introduced by Maass et al. [21]. The computational device is a neural network whose architecture is not dependent on a particular application, rather the neural structure is biologically realistic. There is an increasing amount of evidence for a neural structure described by the liquid state machine in the neocortex. The model is designed to process a continuous stream of data in a rapidly changing environment.

There has been an increasing amount of evidence of a particular form of neural structure in the brain that is surprisingly stereotypical. An intricately connected column of neurons acts as a fading memory which seemingly has the ability for complex universal computational power. The intricately connected structure of the column of neurons form a great amount of recurrent loops which are capable of retaining information and have the ability to make correlations between data points. The neuronal structure is referred to as the "liquid" in the computational paradigm. Inputs to this neuronal structure are captured in the dynamics of the system. The dynamics of the liquid forms a dynamic path of internal transient states. The recurrent loops in the liquid cause past inputs to be captured in the perturbations of the system and gradually fade out. That is why this neural structure forms a fading memory. Studying a neuronal structure like this is highly complex since biologically realistic neuronal models have dynamics that are not easily understood. Especially considering that each neuron adds a degree of freedom and keeping in mind the complicated dynamics of recurrent loops, such a system high dimensional. Therefore it has been a great challenge to study such complicated dynamics and utilize the great potential of a such a neuronal structure as a computational device.

In [21] it is stated that the connectivity of the column of neurons can be found randomly. Their experiments have been conducted with a neuronal structure, illustrated in Figure 5.2 that has been generated randomly adding to the evidence of its great versatility since the structure is not application dependant. Another option would be to evolve the structure. This holds great promise since this could be done real time, leading to improved performance.

The complex high dimensional dynamics of the neuronal structure can be used to extract information by a so called read-out mechanism. Reading out information from the liquid can be done by a neural network that is trained to analyze the information that is provided by the liquid. Since the liquid has many recurrent loops it can very well be used for spatio-temporal information. It is well suited for spatio-temporal information since the liquid acts as a fading memory. The transient states of the liquid are very dynamic.

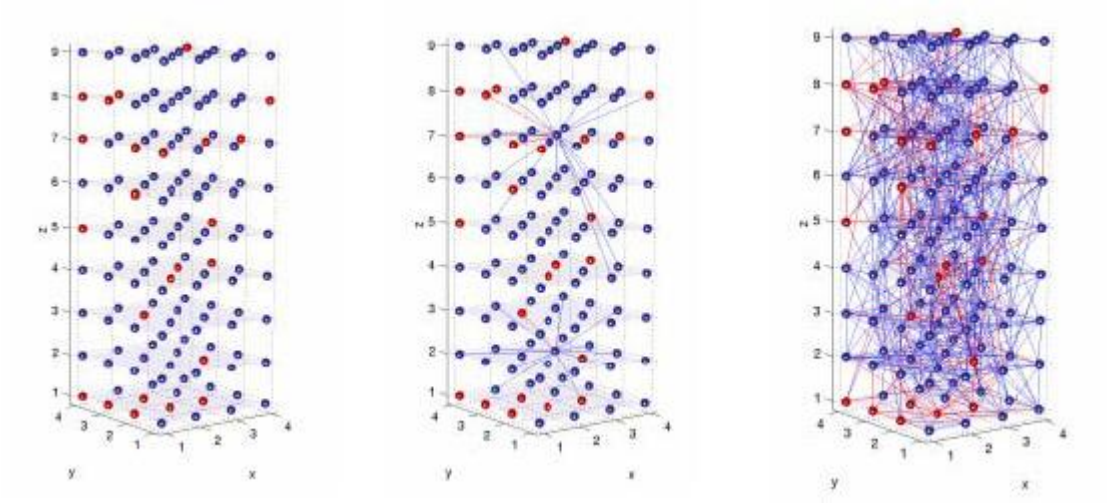


Figure 5.1: Structure of a column of neurons. A column like this, that is randomly connected forms an intricate web of recurrent connections. Another option is to evolve the connections(like in a real brain) as to get a structure optimized to retain information for a specific pattern. These recurrent connections are capable of retaining information of past inputs and acts as a fading meory. All the information of past inputs can be read out of this structure by special readout neutworks that are trained to recognize a particular pattern. (Graphics by Maass)

The transient liquid never goes back to the same state yet the read-out neural network can be trained to recognize dynamical states. In [21] this is noted as an unexpected result; "each readout can learn to define its own notion of equivalence of dynamical states within the neural microcircuit, and can then perform its task on novel inputs". they define it as "the readout-assigned equivalent states of a dynamical system". In fact, different read-outs can be trained to interpret the same state in a different way providing the ability for parallel computation.

5.3 The Computational Model

In [21] it is stated that under idealized conditions the Liquid state machine is capable of universal computational power. They provide a mathematical framework to prove this. The difference with a Turing machine however, is that the liquid state machine computes on real-time input in continuous time with a fading memory. Another difference is that where a Turing machine

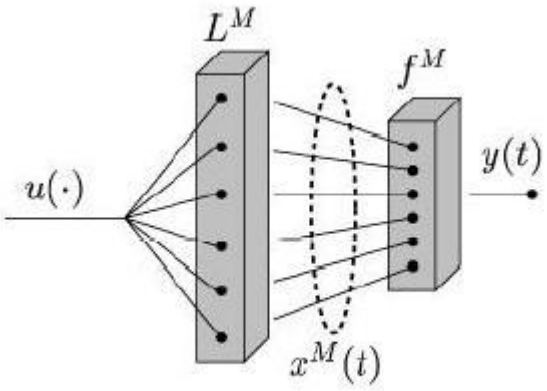


Figure 5.2: A function $u(t)$ acts as a continuous stream of disturbances to excite the medium L^M. At time t a liquid state $x^M(t)$ is created. This liquid state is read by a memory-less readout unit to interpret the information provided by the liquid state. The memory-less readout unit produces the output $y(t)$. (Graphics by [21])

performs a state transition for a pre-specified task, the states of the liquid change continuously over time.

The input to the liquid state machine can be characterized by a continuous stream of disturbances $u(t)$ at every time t. Lets denote a liquid state machine by M, and lets denote a state of the excitable medium by $x^M(t)$ at every time t. The liquid is also called a Liquid filter, L^M. The state $x^M(t)$ of the Liquid filter contains all information about a the past disturbances. Its state is a result of the most recent input as well as a result of the perturbations within the liquid filter from the history of inputs. The information about the preceding inputs is continuously decaying, and therefore the liquid filter is a fading memory. Lets denote the output of the liquid, $x^M(t)$, as a result of the input $u(t)$ as:

$$x^M(t) = (L^m u)(t) \tag{5.1}$$

To extract information from the liquid filter in order to analyze the inputs to this liquid filter, memory-less readout maps are used. They are called memory-less since in contrast to the liquid filter they do not possess any mechanism to retain information about previous states. These readout maps are adaptable and therefore contribute to the overall memory of the system. Readout map are, as opposed to the liquid filter, chosen for a specific task. In fact, different task-specific readout maps can be used to analyze the same

state of the liquid filter and potentially perform different tasks in parallel. The memory-less readout maps, f^M transform the liquid state into the output $y(t)$ at every time t by the following definition:

$$y(t) = f^M(x^M(t)) \tag{5.2}$$

Note that more traditional systems use some location to store the information about past stable states. For example in buffers or tapped delay lines. However in connectionist models it is a good idea to use past states implicitly. In [21] it is stated that it is important to focus on two crucial properties; *the separation property* and *the approximation property.*

The separation property means that for some time point t two significant different inputs $u(t)$ and $v(t)$, there are noticeably different liquid states x_u^M and ℓ_v^M. When this separation property is met in combination with a good readout map, the need to store information in some location until a decision is made it can be discarded, is circumvented.

The approximation property means that the liquid states can be decoded by the readout maps in a proper way. There needs to be sufficient information in the liquid state for a readout map to transform the liquid state into a certain target value. The approximation can be seen as the plasticity of the readout maps when analyzing the liquid state.

Chapter 6

Supervised Learning for Spiking Neural Networks

For a long time a supervised learning algorithm has been lacking for spiking neural networks. Learning with networks of spiking neurons was possible but only in an unsupervised manner. Around 2002 two supervised learning algorithms were introduced. The p-Delta learning rule for parallel perceptrons by Maass [2] and Spikeprop by Bohte [7]. In Section 6.1 we will describe the Spikeprop algorithm and in Section 6.2 some problems of Spikeprop will be reviewed. In Section 6.2.1 an extension on spikeprop will be described and in Section 6.3 we will review the model of [2]. Then, in Section 7.3 we will introduce a new learning algorithm for parallel perceptrons. This new algorithm will enhance the resolution and flexibility of the encoded output by taking the temporal information into account that is present in the delays of the presynaptic spike trains. For convenience and clarity we will adopt the notation of [7] for the neural dynamics.

6.1 Multilayer Supervised Learning for Spiking Neural Networks

In this section the SpikeProp algorithm devised by [7] will be explained. The Learning algorithm is modelled akin to the traditional error-backpropagation methods. It was found that networks of spiking neurons, with biologically realistic action potentials, could perform complex non-linear classification tasks with fast temporal coding just as well as the more traditional sigmoidal networks. The information communicated through the network is done only by the timing of the spikes. In [7] it is shown that with temporal coding less spiking neurons are needed to solve the interpolated XOR problem than with

instantaneous rate coding. This credits their newly proposed temporal coding technique, described in section 4.4. The algorithm spikeProp is extended by Schrauwen [33] where several improvements are proposed. He shows it is possible to learn several constants used in the network and by this severely reduces the number of weights needed to solve the interpolated XOR problem.

6.1.1 SpikeProp

The Neuron model used is the Spike response model(SRM) introduced by Gerstner [14], the algorithm can be adapted to numerous spiking neuron models depending on the choice of the spike response function. Let Γ_j define the set of pre-synaptic neurons for a neuron j. Neuron j receives spikes at times t_i from neurons $i \in \Gamma_j$. A neuron fires at most one spike during a simulation interval(presentation of one input variable), and fires when the internal state variable reaches threshold ϑ. The internal state variable is defined by

$$x_j(t) = \sum_{i \in \Gamma_j} w_{ij} \varepsilon(t - t_i) \tag{6.1}$$

where w_{ij} is the connection efficacy or weight of the connection. The spike response function is defined by:

$$\varepsilon(t) = \frac{t}{\tau} e^{1 - \frac{t}{\tau}} \tag{6.2}$$

where τ is the membrane potential decay time constant which regulates the rise and decay time of the PSP. In the network used in [7] a single connection between a neuron i and a neuron j consists of a fixed number of m synaptic terminals. Assume a pre-synaptic neuron i fires at time t_i. A synaptic terminal is a sub-connection that associates a different delay time d^k and weight w_{ij}^k with the pre-synaptic spike t_i. The delay d^k is the difference between the time the presynaptic neuron fires and when the postsynaptic neuron receives the spike and thus affects the postsynaptic potential. In [7] a presynaptic spike at a synaptic terminal k is described as a PSP of standard height with delay d^k. The unweighed contribution of a pre-synaptic spike from neuron i for synaptic terminal k is then defined by:

$$y_i^k(t) = \varepsilon(t - t_i - d^k) \tag{6.3}$$

where $\varepsilon(t)$ is the spike response function determining the shape of the PSP. Furthermore $\varepsilon(t) = 0$ for $t < 0$.

Equation 6.1 is extended for multiple terminal synapses per connection and the internal state variable x_j receiving stimuli from all other connected

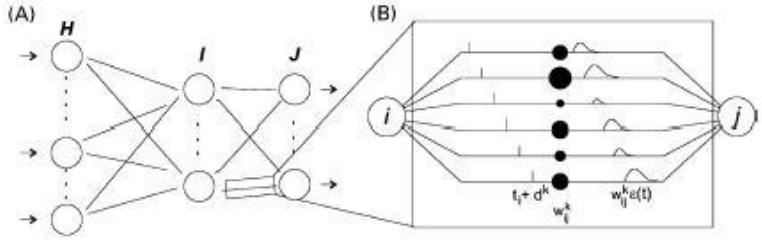


Figure 6.1: (A) is a Feedforward spiking neural network. (B) A connection between neurons consists of multiple sub-connections called synaptic terminals. Every synaptic terminal has a delay d^k and a weight w_{ij}^k. The delay determines the time between the spike fired by the pre-synaptic neuron and the change in PSP of the post-synaptic neuron. The weight regulates the amount of change the pre-synaptic spike will make to the membrane potential of the post-synaptic neuron. (graphics by [7]).

neurons $i \in \Gamma_j$ where Γ_j is the set of all immediate predecessors for neuron j. Then the internal state variable x_j can be described by the weighted sum of the pre-synaptic spikes.

$$x_j(t) = \sum_{i \in \Gamma_j} \sum_{k=1}^{m} w_{ij}^k y_i^k(t) \tag{6.4}$$

where weight w_{ij}^k is the weight associated with the synaptic terminal k. The firing time is determined by the first time the internal state variable crosses the threshold $\vartheta : x_j(t) \geq \vartheta$. The firing time is thus a non-linear function of the state variable $x_j : t_j = t_j(x_j)$. In [7] the threshold is constant and equal for all neurons in the network. Because time to first spike encoding is used for the feed-forward network it is not important what happens to the neuron

after it fires. Keeping in mind that a neuron is only allowed to fire once per simulation interval(presentation of one input variable), all the information the neuron provides is in the first and only spike it produces during that simulation interval. This means that all the information neuron i provides is based on its inputs, γ_i, until neuron i emits a spike. The inputs a neuron i receives after it has emitted a spike do not influence the postsynaptic neuron i, nor the dynamics of the network. This also makes sure that the information the neuron provides us with is only for the present input variable. We do not need to to reset the neuron nor incorporate a refractory period. For an in depth description of the spikeProp algorithm see [7].

6.2 Problems with SpikeProp

For a long time a supervised learning method has been lacking for networks of spiking neurons and the SpikeProp algorithm eagerly fills this gap. Since this algorithm is the first in its kind it is not surprising some initial problems with this algorithm exist.

So far, the network has not been tested on true numerical data, but merely on categorical data sets. This raises the question whether the network is suitable for regression tasks, although this is also strongly dependant on the encoding of the variables that act as input for the network.

Some problems with this algorithm have been addressed by Schrauwen [33]. In [7] it was already noted that a problem could arise with "silent" neurons. This is due to the fact that if a neuron no longer fires for any input there is no mechanism to prop up the weights again so as to make this neuron fire again. Weight initialization is a difficult problem, but is even more crucial with the spikeProp algorithm. The weights have to be initialized exactly in such a way that there is at least one input pattern a neuron responds to. If not, the network will not be able to learn the input data. With the traditional sigmoidal networks weight initialization was always difficult as well. It can severely affect the learning speed of the network. However in the case of SpikeProp it is crucial since not just the performance of the network depends on it, it is the deciding factor whether the network will learn at all. In [33] it is suggested to actively lower the threshold by a factor of 0.9 whenever a neuron stops firing. This corresponds to scaling all the weights up. Another way to interpret the mechanism to make a neuron start firing again.

Another problem, or at least a factor that severely compromises the performance of the network, addressed in [33], is that with the network described by [7] there is a tendency to an overly determined architecture. As described in section 6.1 we need to enumerate the delays of the synaptic terminals since

for every synaptic terminal and thus for every weight we need a different delay over some specified interval. The enumerated delays remain fixed over the entire run of the network. This causes an overspecification of the network and it hardly seems biologically plausible that these parameters of the network are so rigidly determined. So in [33] it is suggested to adapt these delays as well as the other parameters. The adaptation of the delays will reduce the number of synaptic terminals necessary and thus the weights used in the network. The adaptation is very similar as the weight update rule. Just use the gradient error method with respect to a certain parameter. The parameters suggested to update in [33] are the delays d_{ij}^k, the membrane time constant τ and the threshold ϑ. Updating the threshold for spiking neurons is nothing new however. It seems biologically plausible and has been done before by many researchers. To use this in a supervised adaptive manner with a gradient error method is new and seems biologically plausible. However it does raise the question whether we do not get too many degrees of freedom in the network and if the order in which these adaptations are performed make a difference. In [33] the actual algorithm and thus the order in which to make the adjustments are not described.

6.2.1 Adapting the Parameters of the Network

In this section the derivation of the learning rules for the parameters will be explained. The parameters that are being adapted in [33] are the weights w_{ij}^k, the synaptic time constants τ_{ij}^k, the synaptic thresholds ϑ_i and the delays d_{ij}^k. This also means that there will be four different learning rates $\eta_w, \eta_\tau, \eta_\vartheta$ and η_d respectively.

Learning Rule for The Synaptic Time Constants

The synaptic time constant τ models the decay rate of the membrane potential and determines the shape of the PSP. A higher value for τ will decrease the decay rate of the membrane potential. This clearly has a big influence of the firing time of the neuron. It is necessary to calculate how a change in the synaptic time constant changes the error E. This can be expressed this as:

$$\frac{\partial E}{\partial \tau_{ij}^k} = \frac{\partial E}{\partial t_j}(t_j^a)\frac{\partial t_j}{\partial x_j(t)}t_j^a\frac{\partial x_j(t)}{\partial \tau_{ij}^k}(t_j^a) \tag{6.5}$$

where τ_{ij}^k is the membrane constant for synaptic terminal k between neuron i and j. The last term in 6.5 can be written as:

$$\frac{\partial x_j(t)}{\partial \tau_{ij}^k}(t_j^a) = w_{ij}^k \frac{\partial \varepsilon_{ij}^k}{\partial \tau_{ij}^k}(t_j^a - t_i^a - d_{ij}^k)$$
$$= w_{ij}^k \varepsilon_{ij}^k (t_j^a - t_i^a - d_{ij}^k) \left(\frac{(t_j^a - t_i^a - d_{ij}^k)}{(\tau_{ij}^k)^2} - \frac{1}{\tau_{ij}^k} \right) \tag{6.6}$$

where ε_{ij}^k is the spike-response function for which the delay d_{ij}^k and the membrane time constant τ_{ij}^k for synaptic terminal k are used. Now we have:

$$\frac{\partial E}{\partial \tau_{ij}^k} = w_{ij}^k \varepsilon_{ij}^k (t_j^a - t_i^a - d_{ij}^k) \left(\frac{(t_j^a - t_i^a - d_{ij}^k)}{(\tau_{ij}^k)^2} - \frac{1}{\tau_{ij}^k} \right) \delta_j \tag{6.7}$$

The update rule for the membrane time constant then becomes:

$$\Delta \tau_{ij}^k = -\eta_\tau \frac{\partial E}{\partial \tau_{ij}^k} \tag{6.8}$$

Where η_τ is the learning rate for the membrane time constant.
When adapting the delays to a hidden layer, the change in error with respect to the synaptic time constants can be expressed like:

$$\frac{\partial E}{\partial \tau_{hi}^k} = \frac{\partial x_i(t_i^a)}{\partial \tau_{hi}^k} \frac{\partial t_i^a}{\partial x_i(t_i^a)} \frac{\partial E}{\partial t_i^a}$$
$$= \frac{\partial x_i(t_i^a)}{\partial \tau_{hi}^k} \delta_i$$
$$= w_{hi}^k \varepsilon_{hi}^k (t_i^a - t_h^a - d_{hi}^k) \left(\frac{(t_i^a - t_h^a - d_{hi}^k)}{(\tau_{hi}^k)^2} - \frac{1}{\tau_{hi}^k} \right) \delta_i \tag{6.9}$$

The update rule for the membrane time constant to the hidden layer then simply becomes:

$$\Delta \tau_{hi}^k = -\eta_\tau \frac{\partial E}{\partial \tau_{hi}^k} \tag{6.10}$$

Learning Rule for The Synaptic Delay Times

The delay times are a very important factor to determine the firing time of a spiking neuron. To learn these delay times instead of enumerating them has the advantage the network can contain less weights because the network is no longer overly specified. Now it is necessary to know how the delay time from neuron i to neuron j at synaptic terminal k influences the error:

$$\frac{\partial E}{\partial d_{ij}^k} = \frac{\partial E}{\partial t_j}(t_j^a)\frac{\partial t_j}{\partial x_j(t)}(t_j^a)\frac{\partial x_j(t)}{\partial d_{ij}^k(t)}(t_j^a) \tag{6.11}$$

Then the last term can be rewritten as:

$$\begin{aligned}\frac{\partial x_j(t)}{\partial d_{ij}^k(t)}(t_j^a) &= -w_{ij}^k\frac{\partial \varepsilon_{ij}^k}{\partial t}(t_j^a - t_i^a - d_{ij}^k) \\ &= -w_{ij}^k\varepsilon_{ij}^k(t_j^a - t_i^a - d_{ij}^k)\left(\frac{1}{(t_j^a - t_i^a - d_{ij}^k)} - \frac{1}{\tau_{ij}^k}\right)\end{aligned} \tag{6.12}$$

Then for the output layer using the term δ_j equation 6.12 can be rewritten into:

$$\frac{\partial E}{\partial d_{ij}^k} = -w_{ij}^k\varepsilon_{ij}^k(t_j^a - t_i^a - d_{ij}^k)\left(\frac{1}{(t_j^a - t_i^a - d_{ij}^k)} - \frac{1}{\tau_{ij}^k}\right)\delta_j \tag{6.13}$$

Which makes the final update rule:

$$\Delta d_{ij}^k = -\eta_d\frac{\partial E}{\partial d_{ij}^k} \tag{6.14}$$

Where η_d is the learning rate for the delays.
For adapting the delays to a hidden layer the change in error with respect to the delays can be expressed like:

$$\begin{aligned}\frac{\partial E}{\partial d_{hi}^k} &= \frac{\partial x_i(t_i^a)}{\partial d_{hi}^k}\frac{\partial t_i^a}{\partial x_i(t_i^a)}\frac{\partial E}{\partial t_i^a} \\ &= \frac{\partial x_i(t_i^a)}{\partial d_{hi}^k}\delta_i \\ &= -w_{hi}^k\varepsilon_{hi}^k(t_i^a - t_h^a - d_{hi}^k)\left(\frac{1}{(t_j^a - t_i^a - d_{hi}^k)} - \frac{1}{\tau_{hi}^k}\right)\delta_i\end{aligned} \tag{6.15}$$

The update rule for the delay time d_{hi}^k between neuron h and i where neuron i is in the hidden layer then simply becomes:

$$\Delta d_{hi}^k = -\eta_d\frac{\partial E}{\partial d_{hi}^k} \tag{6.16}$$

Learning Rule for The Synaptic Threshold

The change in error with respect to the change in threshold for a neuron in the output layer can be described by:

$$\frac{\partial E}{\partial \vartheta_j} = \frac{\partial E}{\partial t_j}(t_j^a)\frac{\partial t_j}{\partial \vartheta_j}(t_j^a) \tag{6.17}$$

The last term in eaquation 6.17 can be described by:

$$\begin{aligned}\frac{\partial t_j}{\partial \vartheta_j}(t_j^a) &= \frac{1}{\frac{\partial x_j(t)}{\partial t}(t_j^a)} \\ &= \frac{1}{\sum_{i\in\Gamma_j}\sum_{\ell} w_{ij}^{\ell}\frac{\partial \varepsilon_{ij}^k}{\partial t}(t_j^a - t_i^a - d_{ij}^k)}\end{aligned} \tag{6.18}$$

since the first term evaluates to $(t_j^d - t_j^a)$ Equation 6.17 evaluates to $-\delta_j$:

$$\frac{\partial t_j}{\partial \vartheta_j}(t_j^a) = -\delta_j \tag{6.19}$$

The update rule for the threshold of a neuron in the output layer then becomes:

$$\Delta\vartheta_j = -\eta_\vartheta \frac{\partial E}{\partial \vartheta_j} \tag{6.20}$$

where η_ϑ is the learning rate for the neuron threshold. For the hidden layer the change in error with respect to the threshold can be written as:

$$\frac{\partial E}{\partial \vartheta_i} = \frac{\partial t_i^a}{\partial \vartheta_i}\frac{\partial E}{\partial t_i^a} \tag{6.21}$$

The last term will become:

$$\frac{\partial E}{\partial t_i^a} = \sum_{j\in\Gamma^i}\delta_j\sum_k w_{ij}^k\frac{\partial y_i^k(t_i^a)}{\partial t_i^a} \tag{6.22}$$

The first term of Equation 6.21 has been calculated in Equation 6.18

$$\frac{\partial t_i^a}{\partial \vartheta_i} = \frac{1}{\sum_{j\in\Gamma^i}\sum_{\ell} w_{hi}^{\ell}\frac{\partial \varepsilon_{hi}^k}{\partial t}(t_i^a - t_h^a - d_{hi}^k)} \tag{6.23}$$

and so the change in error with respect to the change in neuron threshold becomes:

$$\begin{aligned}\frac{\partial E}{\partial \vartheta_i} &= \frac{\sum_{j\in\Gamma^i}\delta_j\{\sum_\ell w_{hi}^\ell \frac{\partial \varepsilon_{hi}^k}{\partial t}\partial(t_i^a - t_h^a - d_{hi}^k)\}}{\sum_{j\in\Gamma^i}\sum_\ell w_{hi}^\ell \frac{\partial \varepsilon_{hi}^k}{\partial t}(t_i^a - t_h^a - d_{hi}^k)} \\ &= -\delta_i \end{aligned} \tag{6.24}$$

Now the update rule for the hidden layer can be written as:

$$\Delta\vartheta_i = -\eta_\vartheta \frac{\partial E}{\partial \vartheta_i} \tag{6.25}$$

6.3 Feed Forward Learning with Perceptrons

Back-propagation networks pose a problem when being implemented in hardware. The need for a bi-directional network and the communication of high precision floating point numbers make it difficult to implement this type of network in hardware. Other problems with back-propagation are the need for a precise derivative of the activation function and the cascading of multipliers in the backward pass results in a reduced efficiency. Because of these reasons it is questioned whether back-propagation is a biologically plausible method to adapt the behavior of a connectionist system. A method that uses a pool of perceptrons is introduced by Auer in [2]. Such a pool of perceptrons is referred to as *parallel perceptrons* and used as an alternative for the bi-directional networks needed by back-propagation systems. They use perceptrons that consist of gates that use the Heaviside activation function, or spiking neurons. The method to train a parallel perceptron is called *the parallel delta rule*(p-delta rule). The performance is said to be comparable with multi-layer back-propagation networks using sigmoidal units. This method is said to be able to approximate any continuous function arbitrarily well with values between $[0, 1]$.

The combined output of every perceptron in the pool is added up to form the output O and then a squashing function s_ρ applied to this output O. This pool of perceptrons is described as a group of voters. For every voter the output is -1 or 1. The majority vote can then be seen as the binary output of the system. When using spiking neurons this output is binary as well. This method is said to have been successfully applied with spiking neurons in a number of applications using liquid state machines [21, 39]. The architecture of a parallel perceptron combined with the p-delta rule to adapt its behavior, is claimed to be a new hypothesis regarding the organization of learning in

biological networks of neurons that overcome the deficiencies of approaches that were based on back-propagation.

6.3.1 Space Rate Code

Space rate code encodes information by means of the fraction of active neurons. During a time window T, an input will be presented to a pool of perceptrons P. At the end of the input time window T, the potential $x_j(t)$ of all neurons j is calculated to determine if the neuron emitted a spike. The potential $x_j(t) \in P$ for a neuron j in a pool of parallel perceptrons P at time t, is calculated as:

$$x_j(t) = \sum_{i \in \Gamma_j} w_{ij}\, \varepsilon(t - t_i), \tag{6.26}$$

where Γ_j is the set of all predecessors of neuron j, $\varepsilon(t - t_i)$ is the shape of the postsynaptic potential with the current time t, and where t_i represents the firing time of neuron i. This postsynaptic potential ε at time t is defined as:

$$\varepsilon(t) = \frac{t}{\tau}\, e^{1-t/\tau}, \tag{6.27}$$

with τ the membrane constant. We will maintain the notation used in [7] and define the contribution of neuron i to the postsynaptic potential of neuron j as:

$$y_j(t) = \varepsilon(t - t_i). \tag{6.28}$$

Then the postsynaptic potential for each neuron $j \in P$ can be calculated as:

$$x_j(t) = \sum_{i \in \Gamma_j} w_{ij}\, y_i(t). \tag{6.29}$$

The postsynaptic potential will be calculated for each neuron $j \in P$ at the end of time window T. The output of a single neuron j for input z, which is propagated through the pool of perceptrons during a time window T, is then defined as:

$$f_j(z) = \begin{cases} 1 & \text{if } x_j(t) \geq \upsilon, \\ -1 & \text{otherwise,} \end{cases} \tag{6.30}$$

where υ is the threshold. When the postsynaptic potential $x_j(t)$ for a neuron j at time t exceeds the threshold, the output is 1, otherwise the output is 0.

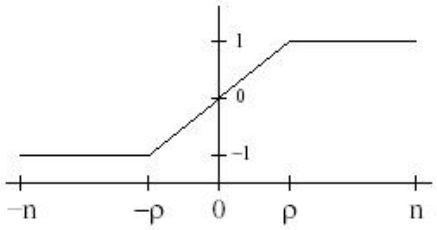


Figure 6.2: The squashing function to determine the output for a parallel perceptron.

To obtain the output of the pool P for input z, we first sum the outputs of the individual neurons as:

$$p = \sum_{j=1}^{n} f_j(z), \tag{6.31}$$

where n is the number of neurons. Then, secondly, the output of P is calculated as the fraction of the neurons in the pool that are active, and is denoted as $s_\rho(p)$, where $s : \mathbb{Z} \to \mathbb{R}$ is a squashing function that scales the sum of the outputs of all the perceptrons into an output in a specific range, e.g. $[-1 : 1]$ defined as:

$$s_\rho(p) = \begin{cases} 0 & \text{if } p < -\rho, \\ p/\rho & \text{if } 0 \leq p \leq \rho, \\ 1 & \text{if } p > \rho, \end{cases} \tag{6.32}$$

The resolution ρ is derived from the tolerable error ϵ and is defined as:

$$\rho = \frac{1}{2\varepsilon}. \tag{6.33}$$

It is important that the number of neurons is greater than the resolution in order to obtain the intended output range.
In [2] it is stated that every continuous function $g : \mathbb{R} \to [-1 : 1]$ can be approximated within any given error bound ε on any given bounded subset of $\mathbb{R}^d$.
The adaptation process for the pool of perceptrons is summarized in Equation 6.41

$$\alpha_j \leftarrow \alpha_j - \eta(\|\alpha_j\|^2 - 1)\alpha_j + \eta \begin{cases} (-y_j(t)) & \text{if } \hat{o} > o + \varepsilon \text{ and } x_j \geq \upsilon, \\ (+y_j(t)) & \text{if } \hat{o} < o - \varepsilon \text{ and } x_j < \upsilon, \\ \mu(+y_j(t)) & \text{if } \hat{o} \leq o + \varepsilon \text{ and } \upsilon \leq x_j < \upsilon + \gamma, \\ \mu(-y_j(t)) & \text{if } \hat{o} \geq o - \varepsilon \text{ and } \upsilon + \gamma < x_j < \upsilon. \end{cases} \tag{6.34}$$

The presynaptic weights are normalized by the factor $\alpha_j - \eta\left(\|\alpha_j\|^2 - 1\right)\alpha_j$, where $\|\alpha_j\|^2$ is the Euclidian norm, and η represents the learning rate. The weights are adjusted when the output $\hat{o}$ of P differs by an amount more than the tolerable error ε of the target output o. Just as described in Section 6.3.2, we use that every perceptron is encouraged to have the right sign by a margin of γ. The parameter μ determines the importance of actually reaching this margin.

In our experiments we will use an output range of [0:1]. Because of this we made some changes to the manner an an output for a single neuron is calculated, the squashing function and the resolution. The output of a single neuron is defined as:

$$f_j(z) = \begin{cases} 1 & \text{if } x_j(t) \geq \upsilon, \\ 0 & \text{otherwise.} \end{cases} \tag{6.35}$$

The squashing function we will use is then defined as:

$$s_\rho(p) = \begin{cases} 0 & \text{if } p < \rho_{low}, \\ p/\rho & \text{if } 0 \leq p \leq \rho, \\ 1 & \text{if } p > \rho_{high}, \end{cases} \tag{6.36}$$

where ρ_{low} is the lower bound of neurons in pool P that have to respond and is defined as:

$$\rho_{low} = \frac{(n - \rho)}{\varsigma}, \tag{6.37}$$

where ς is a parameter > 0. The upper bound of neurons to respond is defined by rho_{high} which is defined by:

$$\rho_{high} = n - \rho_{low}. \tag{6.38}$$

The function of the parameters ρ_{low} and ρ_{high}, is to serve as a buffer to give the pool of neurons more flexibility. This way, in case of the maximal target value, not all neurons in the pool have to fire, and in case of the minimal output target value, it is ensured that the output does not depend on just a single neuron.

6.3.2 The Perceptrons

The behavior of a perceptron(also referred to as a threshold gate or McCuloch-Pitts neuron) can be characterized as:

$$f(z) = \begin{cases} 1, & \text{if } \alpha \cdot z \geq 0 \\ -1, & \text{otherwise} \end{cases} \tag{6.39}$$

$$O_i = s(\sum_{i=1}^{n} f_i(z)) \tag{6.40}$$

6.3.3 The p-Delta Rule

Now a summary of the p-delta rule will be given. the output $\hat{o}$ of the parallel perceptron is defined as $\hat{o} = s_\rho \sum_{i=1}^{n}(f_i(z))$, where $f_i(z) = +1$ when $\alpha_i \cdot z \geq 0$ and $f_i(z) = -1$ if $\alpha_i \cdot z < 0$. For every perceptron $i = 1, 2, \ldots, n$:

$$\alpha_i \leftarrow \alpha_i - \eta(\|\alpha_i\|^2 - 1)\alpha_i + \eta \begin{cases} (-z) & \text{if } \hat{o} > o + \varepsilon \text{ and } \alpha_i \cdot z \geq 0 \\ (+z) & \text{if } \hat{o} < o - \varepsilon \text{ and } \alpha_i \cdot z < 0 \\ \mu(+z) & \text{if } \hat{o} \leq o + \varepsilon \text{ and } 0 \leq \alpha_i \cdot z < \gamma \\ \mu(-z) & \text{if } \hat{o} \geq o - \varepsilon \text{ and } -\gamma < \alpha_i \cdot z < 0 \end{cases} \tag{6.41}$$

Where γ is a margin by which a perceptron has to have the right sign, α_i - $\eta(\|\alpha_i\|^2\text{-}1)\alpha_i$ is a normalizing factor for the weight vector α_i that penalizes a deviation from unity length, η is the learning rate, μ is a measure of importance for the margin γ, o is the target output and z is the input vector.

One of the drawbacks of this learning method can be seen in Figure 6.2. In order to decrease the error bound ε and thus increase the precision of the function approximated, we need more perceptrons. This problem is due to the lack of computational power of a pool of perceptrons since a parallel perceptron can be reduced to a feed-forward network with only one layer of adaptable weights. According to the authors of [2] it can approximate any continuous function $g : \mathbb{R} \rightarrow [-1 : 1]$ with an arbitrary error bound ε. Theoretically this is true, but in [2] it has only been tested on classification problems. The goal of our computational model is pattern recognition in temporal data and not a classification task. This puts strict requirements of computational power and approximation accuracy on our readout mechanism of our liquid state machine. Since we do not intend to implement our model in hardware and it is well known that networks that consist of two layers of adaptable weights are more powerful than one layer of adaptable weights, we

choose to use another supervised learning technique for networks of spiking neurons explained in the next section.

Chapter 7

Experiments

In this chapter we will demonstrate the results of our methods concerning the computational capabilities of a randomly populated and connected column of neurons. Our concepts lean heavily on [21] and [19] with a few drastic modifications. In Section 7.1.1 we describe the liquid and the parameters used. In Section 7.2.1 we describe how sparseness can be measured which is considered common knowledge. Our major contribution in an attempt to make a practical computational tool out of this concept, is to evolve a sparse neural code and to establish a framework of learning rules which creates a trainable computing reservoir. In order to do this we redefine sparseness to different time scales which is described in Section 7.2.2 and Section 7.2.3. We then describe how to adapt the liquid weights to meet the sparseness constraints of the liquid in Section 7.2.5 and Section 7.2.6. For each input only a fraction of the neurons is allowed to be active. The selection of which inputs are active for a certain input stimulus is done algorithmically such that all neurons are utilized efficiently. In Section 7.4.2 we will show that this concept leads to a tremendous ability to scale up the liquid dimensions without adding to the computational load. Since a sparse neural code is present within the liquid, the scaling ability also applies to the storage capability of the network. Originally in [21] and [19] only the weights to the readouts were trained. In Section 7.3.3 we will introduce a set of local learning rules which will adapt the firing times of the neurons within the randomly connected population of neurons in order to reduce the readout error. These learning rules rely both on a gradient descent method and a method where an attempt is made to control how specialized a neuron responds to the presynaptic spikes it receives. In Sections 7.4.4 and 7.4.5 we describe methods to efficiently use the great number of connections within the liquid, since initially the spiking nature of the neuronal model causes some connection to remain ineffective. In Section 7.3.1 we introduce a mechanism to encode the liquid state into an

output value by averaging the firing time of a so called readout structure. Finally in Section 7.4.7 we will demonstrate the network's ability to act as a fading memory.

7.1 The Liquid

In Section 7.1.1 the liquid parameters will be described and these values will be used for the experiments unless stated otherwise.

7.1.1 The Liquid Parameters

We will use a liquid that is closely modeled after the column of neurons as described in [21], however there are differences. We use encoded inputs by an encoding method as described in [6]. Inputs are encoded into spike times by an array of neurons that each has a sensitivity profile shaped as a Gaussian. The Gaussian is used to calculate the spike time(s) as is also described in 4.4. We set γ to be 0.1, which makes a fairly broad Gaussian, for 30 neurons per input variable. A spike train is calculated for every neuron within the sensor. The initial spike t_{init} is calculated as:

$$t_{init} = t_{min} + (1.0 - r)\,(t_{max} - t_{min}), \tag{7.1}$$

where t_{max} and t_{min} are the maximum and minimum firing time of each sensor neuron respectively and r is the resulting value calculated by Equation 4.9. The inter spike interval t_{isi} is calculated as:

$$t_{isi} = \frac{1.0}{r\, t_{maxrate}}, \tag{7.2}$$

where $t_{maxrate}$ is a constant used to scale the inters pike interval. We set t_{min} to be 0 and t_{max} to be 0.07. The inputs are being propagated through the liquid during a certain time interval T_p before the next input is being encoded and propagated through the liquid of neurons. The start of the time interval will be denoted as T_p^s and the end of the time interval will be denoted as T_p^e. This time interval is important since the effect of an input still needs to be present when the next input is being presented to the liquid. The length of the time window needs to be considered with the membrane time constant τ_m of the neuron since both these parameters will partly create a temporal effect. However we believe that the greatest influence on the temporal effect, which makes the liquid act as a fading memory, are the recurrent connections form the context layer. Connections between neurons i and j are generated

according to the probability

$$C \cdot exp(-D(i,j)^2/\lambda^2),$$

where $D(i,j)^2$ is the Euclidean distance between neurons i and j, C scales the probability for creating a connection, λ is a parameter which controls the connectivity density as well as the average distance between the connections. We chose to set $\lambda = 2$. Each connection between a neuron i and j has a delay $d_{i,j}$ which is determined according to

$$T_p\left(\left(D(a,b)/D_{max}\right)/\varpi\right),$$

where D_{max} is maximum distance possible within the liquid of neurons, and ϖ is a parameter which scales the delay. We set ϖ to 0.7. The neuronal dynamics are a simplification of the spike response model as described in Section 3.2. We model the postsynaptic potential of neurons in the liquid as

$$\varepsilon(t) = \frac{t - t_i}{\tau} e^{1 - \frac{t - t_i}{\tau}}$$

like Equation 6.2, where τ_m is the membrane potential and t_i and incoming spike at time t from neuron i. We model the neurons refractory by:

$$-\eta_{refr} \cdot exp(-(t - t_i)/\tau_{refr}),$$

where η_{refr} is a parameter which scales the refractory, and τ_{refr} is the refractory time constant. In addition each neuron has an absolute refractory time denoted by t_{abs}. We set η_{refr} to be 0.005, τ_{refr} to be 1.0 and τ_{abs} to be 0.003. Note that the absolute refractory time is very small since a large absolute refractory time would make a low sparse target trivial and a high sparse target impossible. A small absolute refractory time does not restrict the activity and poses a greater challenge to the activity controlling algorithm. We set 20 percent of the liquid neurons to be inhibitory.

7.2 Evolving the Activity of the Liquid

In this section we will describe the learning rules in order to evolve a sparse liquid response. A sparse liquid response will dramatically improve computational performance. On top of that a sparse code will reduce the crosstalk when training the network and will associate portions of the liquid with specific input values. We believe this improves the learning ability of the network. In Section 7.2.1 we will explain how to measure sparseness. Then in Sections 7.2.2 and 7.2.3 we will explain how to differentiate between population sparseness and lifetime sparseness. Finally in Sections 7.2.5 and 7.2.6 we will define the learning rules to achieve lifetime and population sparseness respectively.

7.2.1 How to Measure Sparseness

We wish to take advantage of a sparse neural code both for the liquid response as well as the readout activity. The advantage of this sparse neural code is the computational efficiency and the great discriminatory property a true sparse neural code provides. However this raises the question of how to define a sparse code and how to assign a value to the sparseness of a neural code. In fact several notions of how to measure sparseness exist as is described in [27].

A commonly used solution to measure sparseness for a single neuron can be found in [30] which was defined as

$$a = \frac{(1/n \sum_{s=1}^{n} r_s)^2}{1/n \sum_{s=1}^{n} r_s^2}, \tag{7.3}$$

where r_s is the response to the s^{th} stimulus and n is the total number of stimuli a single neuron exhibits. When for all inputs the number of spikes is about the same then the sparseness is 1, when all spikes of a neuron are concentrated as a result of a single stimulus then the result is near 0. The output of Equation 7.3 was conveniently scaled into a value between 0 and 1 by [38] which was used for experiments with cells from the visual cortex. An experiment was set up where controlled stimuli in the form of a sequence of images stimulating the spatial and temporal patterns in and around the Primary Visual Cortex when an animal freely views a natural scene. The measure of the sparseness S is then defined as:

$$S = \frac{(1 - a)}{(1 - a_{min})}, \tag{7.4}$$

where a_{min} is the minimum value Equation 7.3 can produce. Equations 7.3 and 7.4 do not make a distinction between lifetime sparseness and population sparseness. Equations 7.3 and 7.4 measure how selective a neuron is in its responses to input stimuli, however if a neuron responds to every input with a single spike, except for one input where it responds with a lot of spikes, the response is still considered sparse according to Equations 7.3 and 7.4. We disagree with this definition and would like to differentiate between population sparseness, where it is measured what fraction of neurons of a population responds and to lifetime sparseness, where we measure how active a neuron is over a period of time. The main reason for this is to reduce crosstalk between different patterns when training the network.

7.2.2 Lifetime Sparseness

When measuring lifetime sparseness we would like to know how active a neuron is over a given period of time. An interesting question arises about the duration of that time as well as the resolution of that time. We can make a distinction between sparseness for the duration of a single input, which has been defined as a *Time window*, and sparseness over a past number of inputs.

When measuring the sparseness of a single neuron over a past number of inputs we are not interested in how many times a neuron fired for a given input. Whenever a neuron responds to a given input by emitting one or more spikes we want to take into account that the neuron has responded. In the case of lifetime sparseness we define the time bin b_L as the smallest possible amount of time for which the activity of a neuron is relevant. In this case b_L equals the duration of a single time window. We define the occurrence of activity for a neuron i within the timebin b_L as:

$$F_L(i) = \begin{cases} 1 & \text{if } \vartheta : x_i(t) \ \geq \ \vartheta \text{ and } T_S \leq t_i \leq T_S + b_L, \\ 0 & \text{otherwise,} \end{cases} \tag{7.5}$$

where T_s is the start time of the time window the sparseness is being calculated for and t_i is the first firing time of neuron i within the current time window. Measuring the sparseness S_L across the past P inputs then becomes:

$$S_L = \left(1 - \left(\frac{\left(\sum_{p=1}^{P} \ F_L(i) \right)^2}{P \ \sum_{p=1}^{P} \ F_L(i)} \right) \right) / \ (1 \ - (1 \ / \ P)). \tag{7.6}$$

In the course of a time window, i.e., the neuronal response as a result of a single input, it is possible for a neuron to emit several spikes. Note that it is important to acknowledge the difference between the number of times a neuron has been active over the past number of inputs and the number of spikes a neuron emits as a result of a single input. The definition of the activity of a single neuron as a result of a single input is:

$$F_T(i) = \begin{cases} 1 & \text{if } \vartheta : x_i(t) \ \geq \ \vartheta \text{ and } T_S \leq t_i \leq T_S + b_T, \\ 0 & \text{otherwise.} \end{cases} \tag{7.7}$$

The number of time bins,which is the smallest possible amount of time in which a single spike by neuron i can occur, is defined as:

$$B = \frac{b_L}{b_T}, \tag{7.8}$$

where the sparseness S_T for the duration of a single input then becomes:

$$S_T = \left(1 - \left(\frac{\left(\sum_{b=1}^{B} F_T(i) \right)^2}{B \sum_{b=1}^{B} F_T(i)} \right) \right) / \left(1 - (1 / B)\right). \tag{7.9}$$

where b is the smallest possible amount of time in which a spike can occur.

7.2.3 Population Sparseness

In Equations 7.6 and 7.9 it was measured how selective a neuron is in its responses to a set of input stimuli and a single input respectively. However, these equations measure the sparseness of a single neuron. we would like to be able to control the activity of the population of neurons as a whole as well. In our experiments we investigate the response of the liquid when excited by an input. Is a neural code of a population sparse when every neuron in the population contributes to the neural code of the population, but every neuron by itself produces a sparse code? We would like to argue this is not the case. We want to use a sparse neural code to construct a computationally efficient algorithm which reduces to amount of spikes that have to be processed, as well as severely reduce the crosstalk when learning. On top of that we want to create a large discriminatory property where it is easy for a readout to recognize a liquid response and associate this with an output. It is not hard to illustrate that a neural code where for every input a single neuron responds and no two inputs activate the same liquid neuron, has a large discriminatory property. This notion where for every entity in the world is recognized by a single neuron is also known as the *grandmother cell*. This notion is not represented by the definition of sparseness in Equations 7.6 and 7.9. In [42] the sparseness measure as defined by Equation 7.3 is called *lifetime sparseness*, and the sparseness of a population is called *population sparseness*. In [40] Equation 7.4 is used for both lifetime sparseness and population sparseness. However we use, to simplify the notion of population sparseness, simply by the percentage of neurons within the liquid that respond to an input. The definition of population sparseness we will use in our experiments is:

$$S_{pop} = \frac{\sum_{i=1}^{N} F_L(i)}{N}, \tag{7.10}$$

where N is the total amount of neurons within the liquid and $F(i)$ is a function that return a 1 if neuron i emitted a spike and is defined by Equation 7.5.

7.2.4 Adapting the Activity of the Liquid

In this section a set of learning rules will be described that will enable the ability to control the amount of activity of a randomly connected pool of neurons. In Section 7.2.2 and 7.2.3 a distinction was made between lifetime and population sparseness. Similarly, this distinction is made for the learning rules. In Section 7.2.5 it is described how to adapt the lifetime sparseness of a neuron and in Section 7.2.6 it is described how to adapt the population sparseness of a randomly connected population of neurons.

7.2.5 Adapting for Lifetime Sparseness

In Section 7.2.2 it was described how to measure the lifetime sparseness of a single neuron. Since we are aiming for a sparse neural code in our experiments we will enforce that each neuron is only allowed to fire a single spike without setting hard constraints to achieve this. Equation 7.9 defines how to measure the lifetime sparseness of a single neuron. When only a single spike is allowed during a time window, Equation 7.9 results in:

$$\begin{aligned} S_T &= \left(1 - \left(\frac{\left(\sum_{b=1}^{B} F_T(i)\right)^2}{B \sum_{b=1}^{B} F_T(i)}\right)\right) / (1 - (1 / B)). \\ &= \frac{1 - (1 / B)}{1 - (1 / B)} \\ &= 1. \end{aligned} \tag{7.11}$$

The result 1 calculated in equation 7.11 will therefore always be the target lifetime sparseness for the duration of a single input. The lifetime sparseness error will therefore result in:

$$E_T = 1 - S_T, \tag{7.12}$$

where S_T is defined in 7.2.2. To adapt the lifetime sparseness of a single neuron j during a single input we determine the change in presynaptic efficacy of a weight w_{ij} by an amount proportional to the error as defined in Equation 7.12 and the potential contribution the presynaptic neuron i imposed on the postsynaptic neuron j, which is $y_{ij}(t)\, w_{ij}$. Note that when neuron i has not spiked, the amount of current being used to proportionally alter the weight is the last time the potential of neuron i has been calculated. When a neuron has spiked one or more times the potential contribution for a weight is defined as:

$$Y_{ij} = \sum_{f=1}^{F} y_{ij}(t_i^f), \tag{7.13}$$

where F is the total number of spikes neuron i emitted and $y_{ij}(t_i^f)$ is the potential contribution of the weight associated to the connection from neuron j to neuron i at the the time neuron i emitted a spike for the fth time. The adaptation of the presynaptic weight is then defined as:

$$\Delta w_{ij} = -\eta_T \, Y_{ij} \, w_{ij} \, E_T, \tag{7.14}$$

where η_T is the learning rate used when adjusting for the lifetime sparseness S_T measured for a single input.

7.2.6 Adapting for Population Sparseness

In Section 7.2.3 the concept of population sparseness was described. To control the number of neurons that are active for a single input we naturally increase sparseness and increase computational efficiency. However that does not mean that all the resources in the liquid are being utilized *effectively*. When training the network to have only a few neurons active for every input we need to make sure that different neurons are active for different inputs. In order to do this we employ lifetime sparseness across a large timescale. Equation 7.6, which computes the lifetime sparseness across several inputs basically assigns a score regarding to the activity of the neuron. When not enough neurons within the population are active, neurons which have a high sparseness score will be encouraged to start firing spikes. When too many neurons are active within the population, only those neurons that have a low sparseness score will be encouraged to stop firing. This way, efficient use of the neurons is established. Equation 7.10 defines how to calculate the sparseness of a population of neurons. The population sparseness is simply calculated as the fraction of the neurons within the population which are active. The population sparseness error E_{pop} is defined as:

$$E_{pop} = S_{pop} - D_{pop}, \tag{7.15}$$

where D_{pop} is the target population sparseness. The adaptation of the presynaptic weights is subject to constraints with regard to the lifetime sparseness score S_L. When E_{pop} is positive, too many neurons are active. Since we wish to make efficient use of all the neurons within the liquid, we will select only those neurons whose sparseness score S_L lies at most a distance C_{min}^{ℓ} from the minimal sparseness score available S_{max} within the population of neurons.

Similarly,when the error E_{pop} is negative and therefore too few neurons are active, some neurons need to be encouraged to emit a spike. We only select those neurons to adapt their presynaptic weights, when the sparseness score lies no further than C^{ℓ}_{max} from the maximal sparseness score which is present in the population of neurons. This ensures that when too few neurons fire, only those neurons are encouraged to fire that have been the least active for other inputs. The adaptation for population sparseness is then summarized as:

$$\Delta w_{ij} = \begin{cases} \eta_{pop}\, Y_{ij}\, w_{ij}\, E_{pop} & \text{if } S_{max} - S^i_L > C^{\ell}_{min} \text{ and } E_{pop} > 0 \\ & \text{or } S^i_L - S_{min} > C^{\ell}_{max} \text{ and } E_{pop} < 0, \\ 0 & \text{otherwise,} \end{cases} \tag{7.16}$$

where C^{ℓ}_{min} is a constant used when too many neurons are active, C^{ℓ}_{max} is a constant which is used when too few neurons is active, η_{pop} is the learning rate when adapting for population sparseness, S^i_L is the lifetime sparseness for neuron i and $Y\, w_{ij}$ computes the potential contribution from neuron i to neuron j where Y_{ij} is defined by Equation 7.13.

7.3 Enhancing the Resolution of the Readout Population

In this section we will describe the readout we used to compute the output from a liquid state. First in Section 7.3.1 we describe the neural code the readout uses to calculate the output value, then in Section 7.3.2 the learning rules are described which will reduce the output error of the readout population and finally in Section 7.3.3 it will be described how the firing times within the liquid will be adapted in order to reduce the readout error.

7.3.1 Reading the Output

In Section 6.3.1 a single-layer feed forward net called a parallel perceptron using spiking neurons is described. This method calculates the fraction of neurons that emit a spike, and returns that fraction as the final output. This type of neural code is called a *space rate code*. For some inputs a very large number of neurons has to be active to reflect the output. The amount of cross-talk is fairly large since for every input-output pair no constraints are set to which weights should be adapted what could result into a negative effect on the total error of the output. The temporal information present

within the firing time of spike is not being utilized, which could mean a loss of information. The activity of the population provides the output. On top of that, the resolution of the outputs is dependant of the number of neurons in the readout. We wish to make the output code more sparse. Sparse codes have many advantages, the most important ones being less energy used and faster learning due to less crosstalk between patterns [27].
To make the code more sparse we make use of the temporal information contained in the spikes. A small adaptation of the space rate code described in Section 6.3.1 will give the advantage of temporal information provided in the firing time of the spike. Instead of the fraction of the pool of neurons in the readout structure which emit a spike, we use the average firing time of the neurons within the readout structure. To encourage a sparse code we only average over the N_d earliest firing times of the readout. We will define this set of neurons as Γ_o, and only the neurons within the set Γ_o participate in the output. When fewer than N_d readout neurons emitted a spike, the number of elements in the set Γ_o will thus be fewer than N_d. When this occurs we call Γ_o *incomplete*. This does not mean that the other neurons were inactive. The main objective is to reduce crosstalk between patterns to speed up learning. For this reason the presynaptic weights of the readout neurons that do not participate in the output i.e., those neurons $i \notin \Gamma_o$, will not be adapted in order to stop that neuron from firing, since doing so would introduce more crosstalk between the patterns.
The output of a single readout neuron i is simply defined as:

$$t_i = \begin{cases} t & \text{if } i \in \Gamma_o, \\ 0 & \text{otherwise,} \end{cases} \tag{7.17}$$

where the firing time $t^i > t^j$ for all neurons $i \in \Gamma_o$ and for all neurons j where $i \notin \Gamma_o$. The output of the readout is then defined as:

$$\hat{O} = \sum_{i=1}^{N} t_i^a / N, \tag{7.18}$$

where N is the number of neurons in Γ_o and t_i^a is the actual firing time of neuron i. Note that when using this neural code, early firing times are being favored over later firing times. This makes sense with respect to the behavior of a postsynaptic neuron. A neuron which receives presynaptic spikes, and responds with an increasing postsynaptic potential, will emit a postsynaptic spike when the postsynaptic potential reaches the threshold. The potential rises with each excitatory incoming spike and therefore later presynaptic spikes will be less likely to participate in the output. The error for the entire

readout is defined as:

$$E = \frac{1}{2}(\hat{O} - t^d)^2, \tag{7.19}$$

where t_j is the target firing time of the readout. The average firing time $\hat{O}$ as defined by Equation 7.18 is translated to the actual output value of the readout by:

$$O = \frac{\hat{O} \; - \; t^d_{min}}{t^d_{max} \; - \; t^d_{min}}, \tag{7.20}$$

where t^d_{min} and t^d_{max} are predefined constants and represent the minimum and maximum target firing time respectively. The actual output value O results then in a value between 0 and 1. The relationship we defined between lifetime sparseness and population sparseness gives rise to specialization of neurons regarding their response to inputs.

7.3.2 Adapting the Weights

The weights cannot be adapted in a similar manner as the Spikeprop algorithm as described in [7]. A major drawback of the algorithm in [7] is that the spike time of a neuron can only be adjusted when the neuron actually spiked. When achieving a sparse code for pool of neurons obviously we sometimes want to activate a neuron that did not emit a spike, or sometimes we want a neuron to remain inactive. Another reason why we cannot make use of a back propagation style algorithm is that we cannot propagate the error back since the hidden layer is a population of randomly, and therefore recurrently, connected neurons. That is why we will introduce here a Hebbian-like adaptation style which will provide us with more flexibility and which is suitable for a sparse code. The target firing time of a neuron j is calculated as:

$$t_j = t_{min} + \; (t_{max} - t_{min}) \, \xi, \tag{7.21}$$

where ξ is the normalized target value as being presented in the input data. The output neurons have a minimum and maximum firing time denoted as t_{min} and t_{max} respectively. It is important to choose the values of t_{min} and t_{max} carefully. When t_{min} is chosen too small there may not be enough presynaptic spikes with $t \; < \; t_{min}$ which means the neuron can not meet the target firing time. This is a result of the input spike train. When all spikes t emitted by the sensor $t > t_{min}$, the liquid neurons will not be able to emit a spike $t < t_{min}$. As a result of that, the neurons in the readout will not be able to either. When t_{min} is chosen too large the difference between t_{min} and t_{max}

may be too small which negatively impacts the capacity of the readout to represent output values. Note that individual neurons within the readout can have a firing time different that the target firing time. As long as the average firing time of all neurons $j \in \Gamma_o$ equals the target firing time t_j^d. Adaptation of the presynaptic weights for a neuron j is done like:

$$\Delta w_{ij} = \begin{cases} \eta_P \, E \, w_{ij} \, Y_{ij} & \vartheta : x_i(t) \; \geq \; \vartheta \text{ and } i \in P, \\ \eta_P \, (t_{max} - t_{min}) \; w_{ij} \, Y_{ij} & \vartheta : x_i(t) \; < \; \vartheta \text{ and } \Gamma_o \text{ is incomplete,} \end{cases} \tag{7.22}$$

where η_P is the learning rate, $\vartheta : x_i(t) \; \geq \; \vartheta$ means that neuron i emitted a spike and $w_{ij} \, Y_{ij}$ is the contribution of presynaptic neuron j to the potential of postsynaptic neuron i where Y_{ij} is defined by Equation 7.13. Besides the advantage of being able to adapt for neurons that did not emit a spike, this method also has the advantage of not having to calculate computationally expensive derivatives.

7.3.3 Adapting the Liquid Firing Times

Adapting the presynaptic weights for all neurons j in the readout P will not ensure the correct output values after a number of iterations. When all firing times of $t_i < t_j^d$ for neurons $i \in L$, the target firing time can not be realized by adaptation of the presynaptic weight for readout neurons alone. For that reason the presynaptic weights of neurons within the Liquid will be adapted as well. We will not use the technique of back propagation due to the recurrent connections within the Liquid. We will use only local adaptation rules. This has the advantage of being easy to understand and leads to greater computational efficiency. However, it has the drawback of introducing some inaccuracy since a liquid neuron which firing time is adapted will also influence the liquid neurons it is connected to. Despite the inaccuracy the learning rules perform quite well. We think the reason for this is that we rely on a sparse code. The event of a spike emitted by a neuron in the liquid does not only hold information in its firing time, but also holds information by which neuron it is emitted.

When establishing the adaptation which is needed to reduce the error of the output produced by the readout as defined in Equation 7.19, we need to calculate whether an increase or decrease of presynaptic current is necessary. The amount of adaptation solely relies on the readout error which is defined in Equation 7.19. The direction of the adaptation is determined by several factors. First of all, the *sign* of the presynaptic spike with respect to the postsynaptic spike is important. The slope of the postsynaptic potential, as

a result of t_i^a, determines whether a later firing time of neuron i will result in less or more postsynaptic current. If $(t_j^a - t_i^a) - \tau_j < 0$ than the slope is rising, therefore a later firing time of neuron i will result in less postsynaptic current caused by neuron i. This factor will be called the sign S of the postsynaptic slope caused by a presynaptic neuron i and is defined as:

$$S = (t_j^a - t_i^a) - \tau_j. \tag{7.23}$$

The second factor for determining the direction of the adaptation is whether or not neuron i is inhibitory or excitatory. When a neuron is inhibitory we simply need to adapt the firing time in the opposite as when the neuron would be excitatory. Therefore the inhibitory factor I_i of a neuron i is defined as:

$$I_i = \begin{cases} -1 & \text{if neuron } i \text{ is inhibitory,} \\ 1 & \text{otherwise.} \end{cases} \tag{7.24}$$

The third and final factor in calculating the direction of adaptation of a neuron i is determined by the fact whether or not the neuron i actually contributed to the postsynaptic potential to the postsynaptic neuron j. When the firing time $t_i > t_j$, neuron i did not contribute to the postsynaptic current of neuron j. In this case, only when a postsynaptic neuron j needs an increase in postsynaptic current, we need to adjust the firing time of a presynaptic neuron i. The reason for this is simple. When increasing the firing time of a presynaptic neuron, the possibility exists that the firing time of the presynaptic neuron becomes greater than the postsynaptic neuron. When no mechanism exists to increase the number presynaptic neurons i where $t_i < t_j$ early target firing times become impossible. In fact our own experiments confirm this is an actual problem. Note that if it wasn't for the tapped delay lines, this problem could not be overcome by the Spikeprop algorithm as described in [7]. We call this the *orientation* of the presynaptic spike and is defined as:

$$R_i = \begin{cases} 1 & \text{if } (t_j - t_i) < 0 \text{ and } I_i = 1, \\ 0 & \text{otherwise,} \end{cases} \tag{7.25}$$

note that the orientation of a presynaptic neuron i depends on the inhibitory value of neuron i. In short the orientation of a presynaptic neuron i equals 1 when it did not contribute to the postsynaptic potential of a neuron j and is not inhibitory. To combine the factors 7.23, 7.24 and 7.25,we define δ_i to indicate the direction and the amount of adaptation for a neuron i as:

$$\delta_i = \eta_L \left(t_j^a - t_j^d\right) S \, R_i \, I_i. \tag{7.26}$$

The change in efficacy of the presynaptic weights of a neuron i is then defined as:

$$\Delta w_{ij} = \eta \, \delta_i \, Y_{ij} \, r, \tag{7.27}$$

where Y_{ij} is the potential contribution defined by Equation 7.13, r is a number $\in [0.7, 1.0]$, which introduces some noise. We found it improves the performance of the learning process to add some randomness to the adaptation.

7.4 Exploring the Learning Framework

In this section we will explore several properties of the network and its learning rules. in Section 7.4.2 we will investigate the learning ability of the network. Then in Section 7.4.3 we will describe how to prevent the occurrence of very large weights and in Section 7.4.4 we will describe a method to diversify the neural code. In Sections 7.4.5 and 7.4.6 we will describe the favorable properties of specialization in the network. Finally, in Section 7.4.7 we will describe a method to incorporate an implicit time dimension in the network.

7.4.1 Mackey-Glass Data Set

In this section we will describe the data we will present to the network of neurons. We initially use a data set with few data points as a toy problem and is meant to illustrate some key properties of the network's behavior.

The data we will use in the experiments described in Sections 7.4.2 and 7.4.3 is known as the Mackey-Glass time series. This time-series is commonly used to demonstrate the performance of recurrent connected neural networks and is defined as:

$$\frac{dy(t)}{dt} = -by(t) + a\frac{y(t-\tau)}{1+y(t-\tau)^{10}}, \tag{7.28}$$

where a, b and τ are parameters which determine the behavior of the time-series. We set $a = 0.2$, $b = 0.1$ and $\tau = 17$, which results in the time-series which can be observed in Figure 7.1. We have created a small subset of this data to present to the neural network. We have integrated Equation 7.28 at 30 different time points which results in a data set of 30 data points.

7.4.2 Gradient Descent Learning

In this section the results of the methods described in Sections 7.2.5, 7.2.6, 7.3.2 and 7.3.3 will be presented. We wish to demonstrate in this experiment

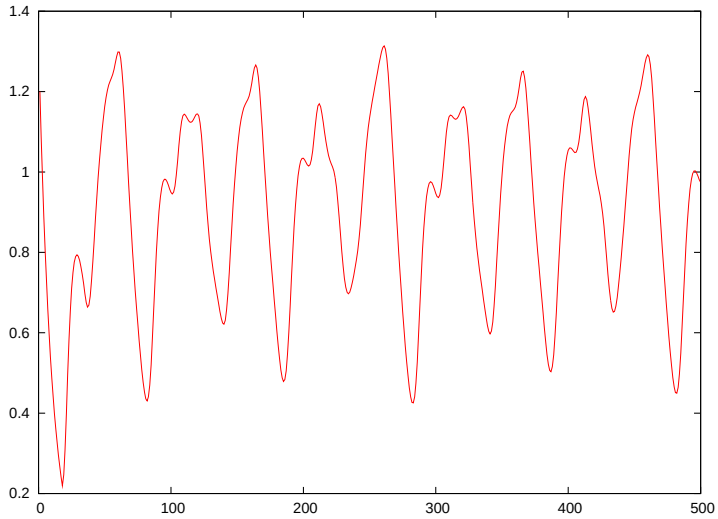


Figure 7.1: Mackey-Glass time-series where parameters $\tau = 17$, $a = 0.2$ and $b = 0.1$

that with a set of learning rules which have been defined locally, i.e., on the level of a single neuron, an input pattern can be learned by the network. Despite the fact the network can be successfully trained on the input data set, a number of drawbacks to this method become apparent.

In Figure 7.2 the responses of the sensor and the liquid is depicted for different liquid and sensor sizes. It can be seen that for different inputs different areas of the liquid are excited and therefore the liquid's resources are being utilized efficiently. When inspecting Figure 7.2 closely, subtle variations of liquid responses can be observed for different inputs. Furthermore the Gaussian shape of the sensor responses is apparent in Figure 7.2(f). As described in Section 4.4 the sensor neuron's response is determined by the output of Gaussian function as a function of the normalized input.

In Figure 7.3 the error is depicted for the readout for each epoch. The error tolerance is 0.01 and when the tolerance is reached the algorithm stops. It is clear that an increase in the number of neurons greatly enhances the performance of the network. For more traditional neural methods this is also the case, however there is an upper limit for the amount of neurons when increasing the network size no longer yields a beneficial effect or can even be harmful to the performance of the network. With a sparse liquid response this harmful effect due to upscaling the network size is confined to the number of neurons active for a single input. However, the size of the liquid can be scaled up indefinitely. This is due to the sparse response of the liquid. Only

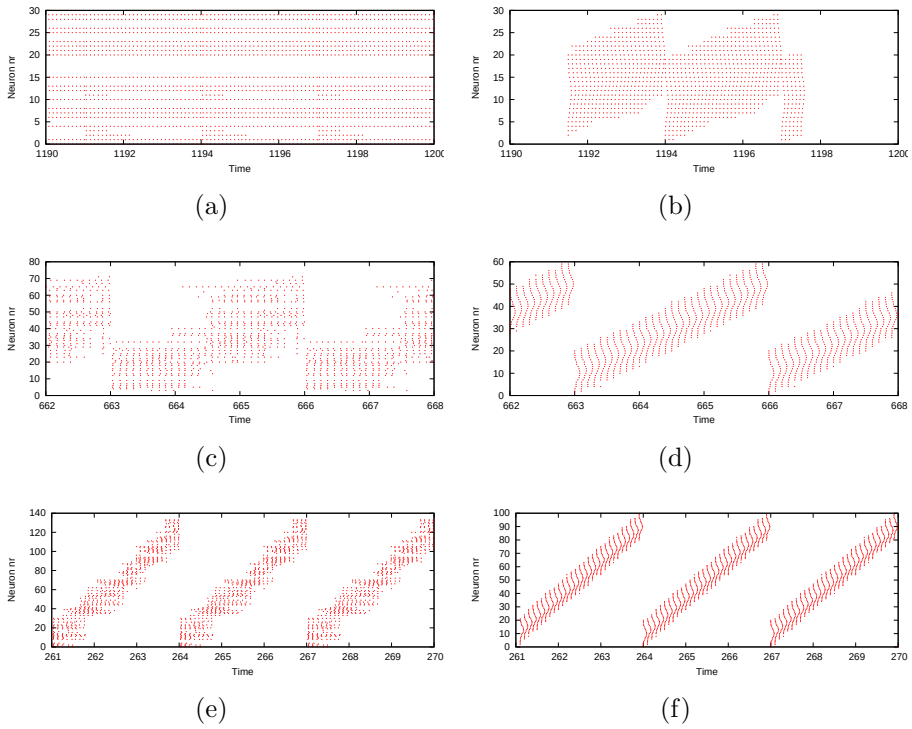


Figure 7.2: The responses of the sensor and liquid for several different sensor and liquid sizes. Figure 7.2(a) depicts the liquid response with a size of 36 neurons and Figure 7.2(b) shows its corresponding sensor response with a sensor size of 30 neurons. Figures 7.2(c) and 7.2(e) show the liquid responses with a liquid size of 72 and 135 neurons respectively and Figures 7.2(d) and 7.2(f) show the sensor responses with sizes 60 and 72 neurons respectively. The x-axis represents the time scale. Neuronal activity is propagated through the network for the duration of 0.1 ms. The y-axis represents the neuron number.

a small part of the liquid responds to an input which is unique to that input. This makes the liquid a highly scalable storage medium.

In Figure 7.4 the error of the population sparseness is depicted. It can be seen that the activity is not within the specified bounds. This is due to the fact that a lot of adaptations are being done to the liquid weights at any given time. Not only are the weights being adapted to adjust for the liquid population activity, the weights are also adjusted to reduce the readout error. Despite this activity error, the amount of liquid activity still allows for all the advantages of a sparse neural code. Note that this error is absolute. This means that the total number of spikes the liquid emits is either below or

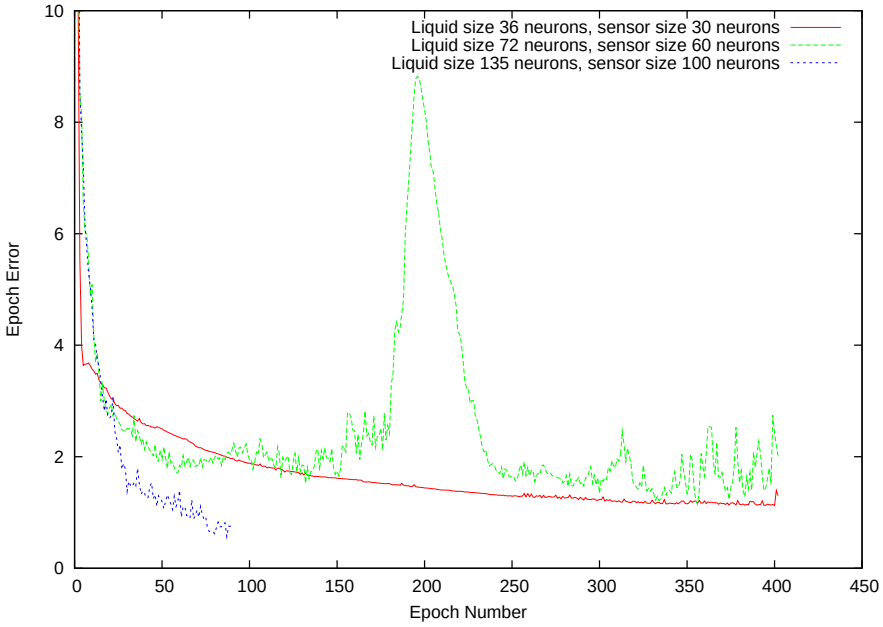


Figure 7.3: The epoch error is plotted against the epoch number. The x-axis displays the epoch number, the y-axis displays the error for each epoch.

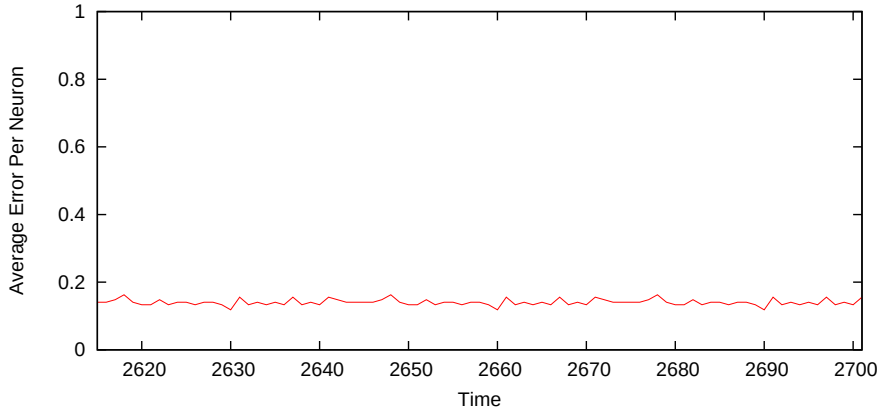
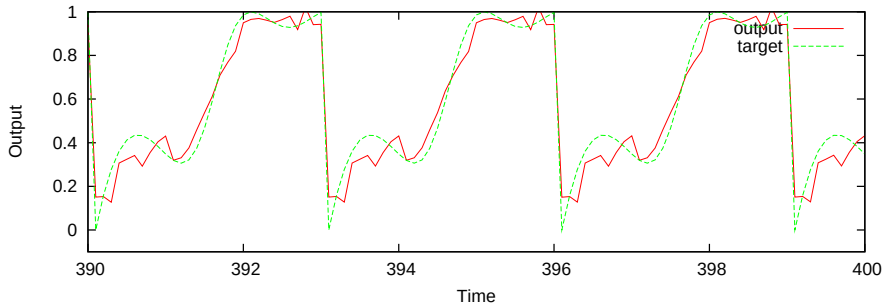
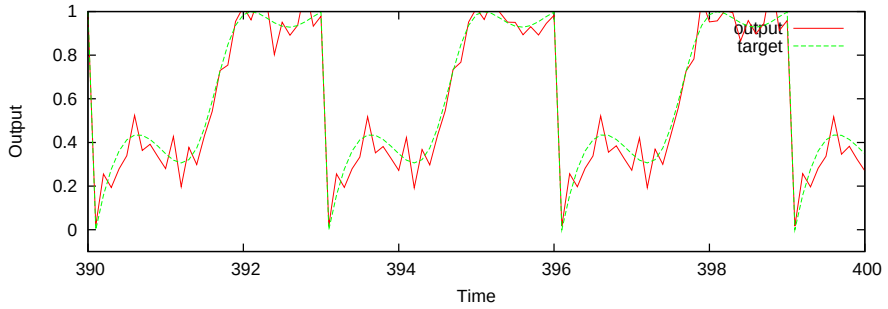


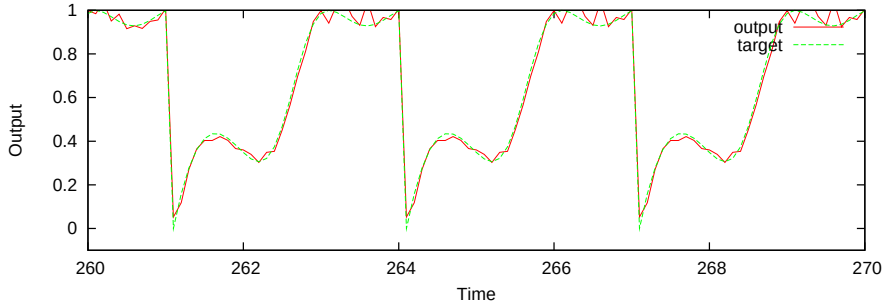
Figure 7.4: Error activity with a liquid size of 135 neurons



(a)



(b)



(c)

Figure 7.5: The actual output of the readout and the target output are plotted for different network sizes. In Figure 7.5(a) the total number of neurons is 126, in Figures 7.5(b) and 7.5(c) the number of neurons is 172 and 275 respectively.

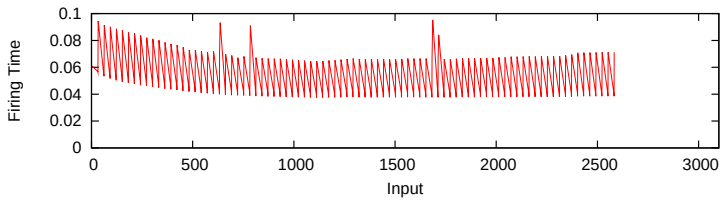
above the target. In Figure 7.5 the outputs of the readout is plotted against the target output. It can be seen that increasing the number of neurons in the liquid and sensor makes for different behavior of the final output by the readout. In Figure 7.5(a) it can be seen that after 400 runs of the entire data set, the lowest possible output is difficult. This is due to a high Euclidean distance of the output while the Euclidean distance of the corresponding input values is very small. Furthermore in Figures 7.5(b) and 7.5(c) some overfitting can be seen. It could be argued this is partly due to a somewhat low resolution of the input data. The data set consists of only 30 data points and makes this a toy problem.

In Figure 7.6 several traces of neuron's firing times can be seen. Notice the variation in responses of the individual neurons. This diversity in neuronal behavior is very important for the readout to efficiently learn the correct output values. The plots span the entire training process and a variety of adaptation directions of the firing times can be noticed as well. The relative firing time is plotted against the time. The relative firing time can have a value in [0,0.1], since the duration of a time window is set to be 0.1 ms.

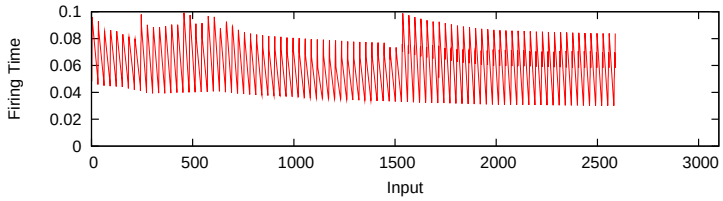
In Figure 7.7 the epoch error for several values of the connection parameters λ_S and λ_L can be seen. In Section 7.1.1 it was described two different connection parameters exist. The connection parameter for connections from the sensor to the liquid λ_S was set to 1.5, 2.0 and 2.5. The connection parameter which determines the connections within the liquid λ_L was also set to 1.5, 2.0 and 2.5 which resulted in 4447, 6657 and 8339 connections within the liquid respectively. In Figure 7.7 it can be seen that increasing the number of connections is a means to increase the networks capacity and with that the performance of the network of neurons. Increasing the number of neurons as well as increasing the number of connections improves the performance of the network. However, when the number of connections are increased we are not sure if all the connections are used efficiently. This issue will be addressed in Section 7.4.4 and Section 7.4.5.

7.4.3 Weight Distribution

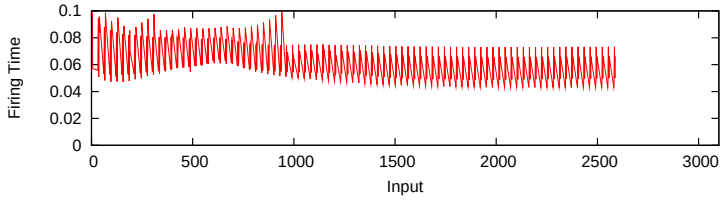
When inspecting the resulting weights after training the network we noticed that a small number of weights were responsible for most of the presynaptic input. This is not surprising since in our learning rules we did not set any constraints as to how the weights evolved as long as the readout error is reduced. We believe that these great differences in value which exist between the weights, do compromise the ability of the network to learn. A relatively high presynaptic weight will always dominate the postsynaptic potential no matter how the input spike train is arranged. We do not believe



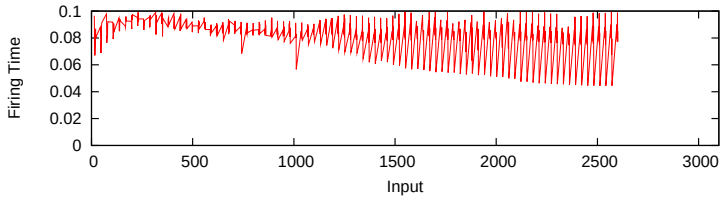
(a) Neuron Number 9



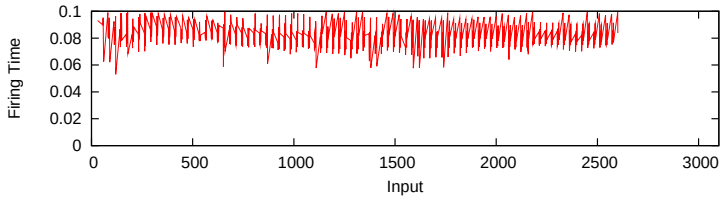
(b) Neuron Number 13



(c) Neuron Number 30



(d) Neuron Number 78



(e) Neuron Number 105

Figure 7.6: Several traces of neuronal responses within the liquid. A great diversity of neuron activity can be seen

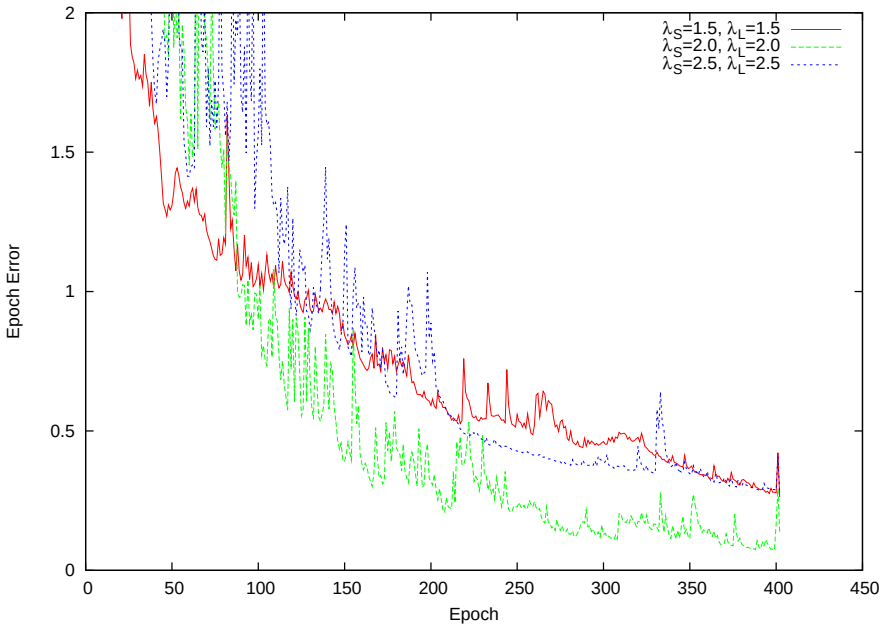


Figure 7.7: The epoch error for several values of the connection parameters λ_S and λ_L.

that the difference in presynaptic timing and order of the presynaptic spikes will compensate for the dominance of a single weight and therefore the possible variation on liquid responses for different inputs is reduced. A fixed limit on a weight did not improve the training results. We therefore introduce a constraint on the set of all presynaptic weights as a whole, for any postsynaptic neuron. We set a limit to the difference between the maximal and minimal presynaptic weight which have participated in a postsynaptic spike. Note that this constraint does not apply to all presynaptic weights, but only to those which have contributed to the postsynaptic potential on the onset of a postsynaptic spike. We call this set the *effective presynaptic weights* Γ_{eff} which applies to a single postsynaptic spike. The constraint C_{eff} on the set of effective presynaptic spikes is defined as:

$$C_{\mathit{eff}} > w_{min} - w_{max}, \tag{7.29}$$

where $w_{min} \in \Gamma_{\mathit{eff}}$ and there is no $w \in \Gamma_{\mathit{eff}}$ for which $w < w_{min}$, and $w_{max} \in \Gamma_{\mathit{eff}}$ and there is no $w \in \Gamma_{\mathit{eff}}$ for which $w > w_{max}$, and C_{eff} is a constant which defines the minimal Euclidean distance between any effective presynaptic weights. To enforce this constraint we increase weights $w < w_{max} - C_{\mathit{eff}}$ by:

$$\Delta w = H(w - w_{max} - C_{\mathit{eff}})\, \eta_{\mathit{eff}}\, |E_P|, \tag{7.30}$$

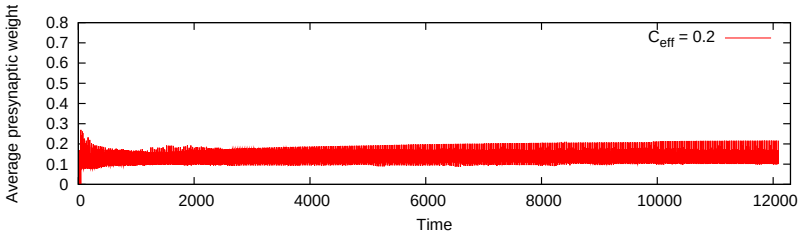
where η_{eff} is the learning rate, E_P is the error of the readout pool P and $H(x)$, also known as the Heaviside function, is defined as:

$$H(x) = \begin{cases} 1, & \text{if } x \geq 0 \\ 0, & \text{otherwise.} \end{cases} \tag{7.31}$$

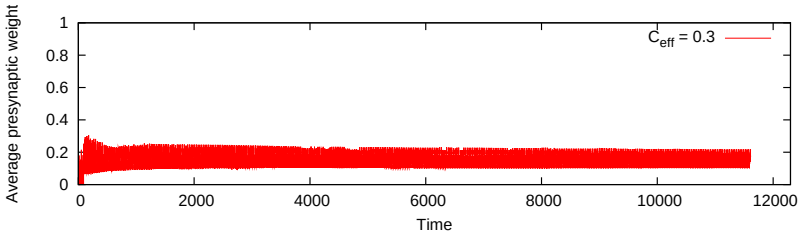
Note that in order to enforce the constraint the maximum value is not adapted and all weights which violate the constraint as defined in Equation 7.29 are increased. This increases the overall postsynaptic potential and the learning rules as defined in Sections 7.2.5, 7.2.6, 7.3.2 and 7.3.3 will ensure that too much presynaptic weight is reduced in order to reduce the sparseness and readout error.

In Figure 7.9 it can be seen that the value of C_{eff} is of some influence on the performance of the network. We trained a network with a reservoir of 90 neurons, a sensor of 60 neurons and a readout of 40 neurons. Higher values of C_{eff} result in a greater precision. This came as a surprise. The expectation was that for a large variance, i.e., a few weights which account for most of the postsynaptic potential, compromises the performance of the network. However, when analyzing what happens we found that due to the difference in the order and timing of the presynaptic spikes for different input values, the higher weights do not always contribute to a postsynaptic spike. A value of 1.0 for C_{eff} seems quite large yet the results are better than for lower values. In Figure 7.8 it can be seen that C_{eff} is enforced on the presynaptic weights of the liquid. For greater values of C_{eff} The average presynaptic weight tends to be higher when the value of C_{eff} smaller. This can be explained by investigating what happens when weight violates the constraint defined by Equation 7.29. This weight is increased until it falls within the constraint defined by Equation 7.29 which is defined on all the presynaptic weights which participated in the postsynaptic spike. The assumption was that this results in too much postsynaptic potential and Equations 7.27, 7.22, 7.14 and 7.16, would decrease all the weights in order to reduce the sparseness and readout error. However, lower values of C_{eff} merely restrict the ability of a neuron to respond uniquely to distinct inputs and therefore lacks precision.

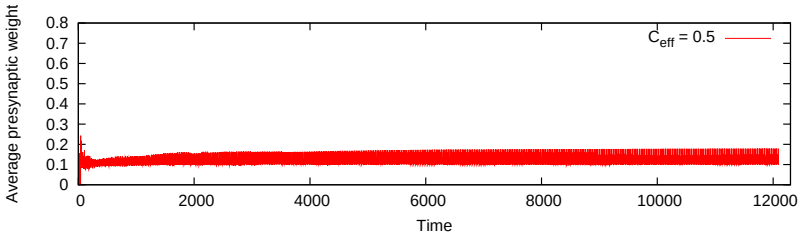
Another prediction we made was that lower values of C_{eff} would lead to more presynaptic spikes for each postsynaptic spike. We assumed the reason for this would be that for low values of C_{eff} a lower presynaptic weight would emerge per connection. However this was not the case. In fact, in Figure 7.10 it can be seen the opposite is the case. For higher values of C_{eff} a slightly lower number of incoming spikes per neuron can be seen averaged over all the neurons within the liquid. We believe this is due to the presence of inhibitory neurons. We also believe this is the reason a fixed limit for any presynaptic



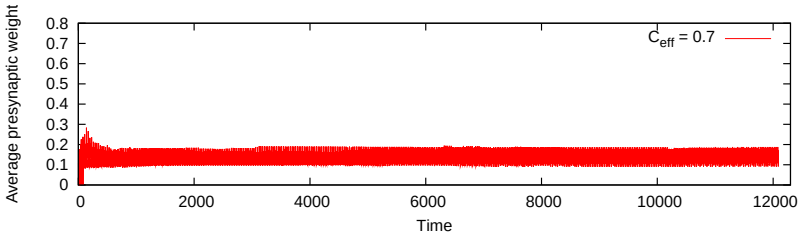
(a)



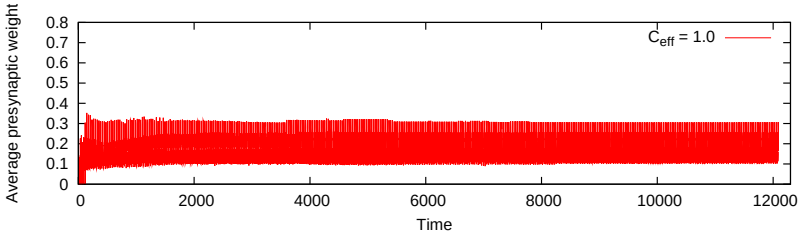
(b)



(c)



(d)



(e)

Figure 7.8: Average Weight difference between the minimum and maximum presynaptic weight within the liquid

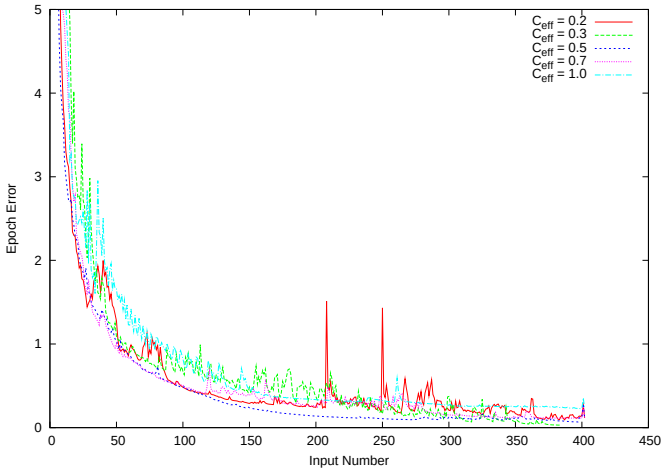
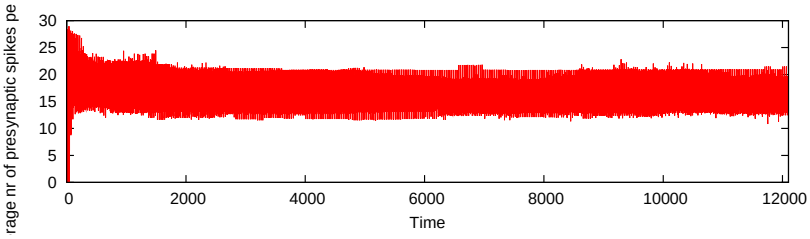
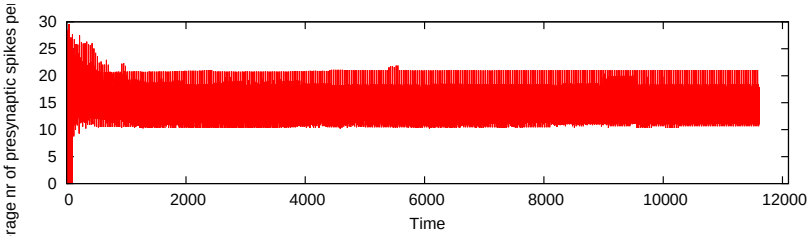


Figure 7.9: Epoch error for several values of the parameter C_{eff}

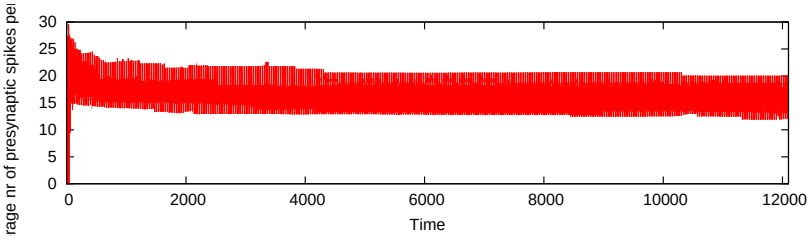
weight does not have a favorable effect on the learning performance. It seems that a quite large variance in presynaptic weight does enhance the learning ability of the network. We believe this introduces a form of specialization for a particular input where the order and timing of the incoming spike train does compensate for these high presynaptic weights.



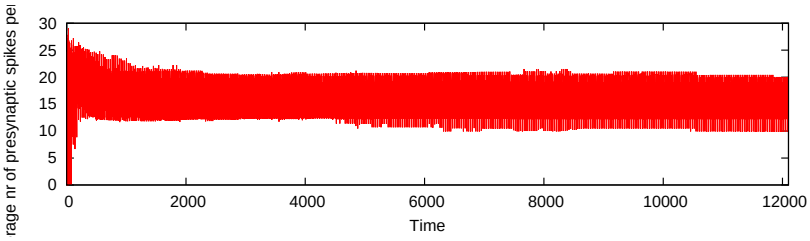
(a)



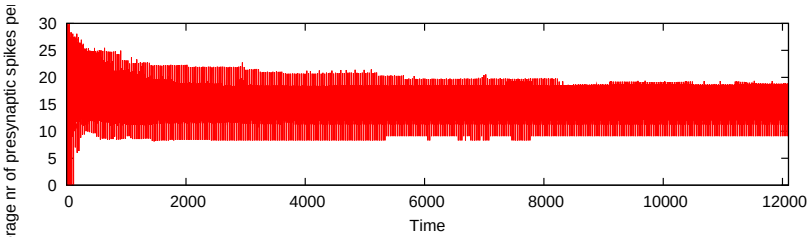
(b)



(c)



(d)



(e)

Figure 7.10: Average number of incoming spikes per postsynaptic neuron within the liquid.

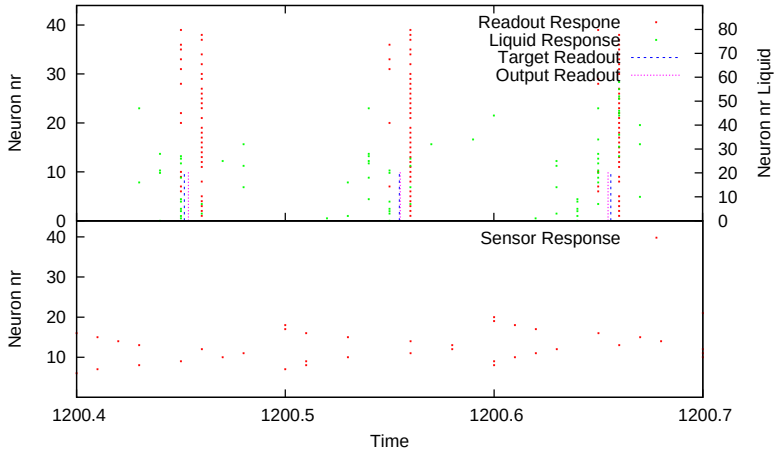


Figure 7.11: An overlapping plot of the liquid firing times and the readout firing times at the top. The vertical bars indicate the readout target and the readout output. At the bottom of the figure the sensor activity can be seen.

7.4.4 Short Term Dynamic Plasticity

In Section 7.4.2 we improved the learning ability of the network by increasing the number of connections. In this section we would like to investigate how efficiently these extra connections are used. It can be seen in Figure 7.11 that the firing times of the readout neurons do not vary a lot. It was described in Section 7.3 that the output of the readout pool is determined by averaging a number of the earliest firing times. It turns out that, although the readout performs quite well, almost all of the readout neurons fire at the same time. Note that through the spiking nature of this neuronal model timing is of the essence. This is not surprising since the output is determined by the firing time of the readout neurons, which is determined by the firing time of the liquid neurons which is in turn determined by the sensor neurons. This is depicted in Figure 7.11. At the bottom the sensor firing times can be seen. Then on top of that the firing times of the liquid as well as the firing times of the readout are depicted. The vertical axis on the left displays the neuron number of the readout pool, while the right axis displays the neuron number of the liquid. The readout output and readout target are also displayed in the

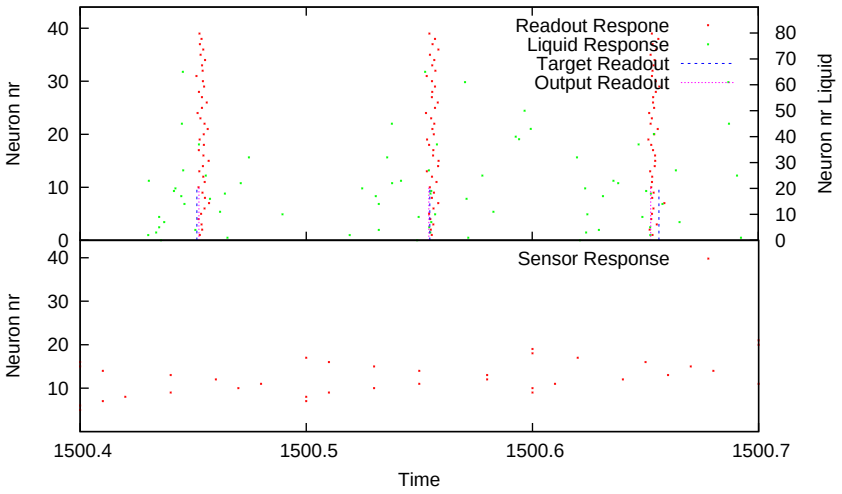


Figure 7.12: An overlapping plot of the liquid firing times and the readout firing times at the top. The vertical bars indicate the readout target and the readout output. At the bottom of the figure the sensor activity can be seen.

figure as the vertical bars. As described in Section 7.3 the readout neurons fire only once. When a readout neuron i fires at time t_i, any presynaptic spike with a spike time $t > t_i$ will not contribute to the output. This means that when the target spike time of the readout is early, fewer liquid and sensor spikes can contribute to the output. The set of effective presynaptic spikes Γ_{eff} is therefore smaller or more specialistic to the input than a late firing time. In Figure 7.11 any sensor or liquid spike with a greater firing time that the readout output, which is indicted by the vertical line, does not contribute to the output value.

We would like to investigate the effect of this causality of this spiking behavior by diversifying the firing times of the readout neurons. When a subset of readout neurons fires at a later time than the target spike time, then this has to be compensated by other neurons which fire at an earlier time. However, the average would reflect the correct output value, while a greater amount of information contained in the liquid and sensor firing times can be utilized in calculating the output value by the readout. We believe this will increase the learning ability of the network. The method which we

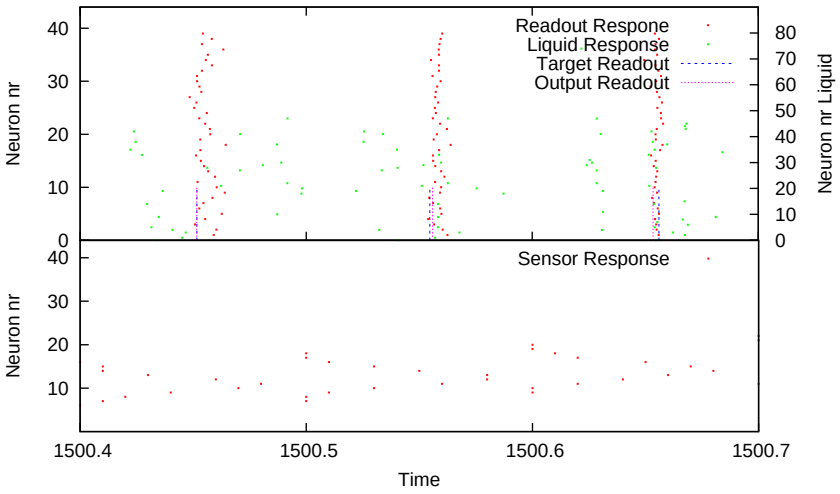


Figure 7.13: An overlapping plot of the liquid firing times and the readout firing times at the top. The vertical bars indicate the readout target and the readout output. At the bottom of the figure the sensor activity can be seen.

use to create more diversity in the firing times of the readout neurons is modeled after the phenomenon called *short term dynamic plasticity* which is described by [32] and [9] amongst others. We implement a variation of this phenomenon by increasing the firing time of those neurons which belong to the upper range of firing times. This means that only high firing times are increased. To correct for this in the output all the neuron's firing times should be decreased again which will give us the desired result. The presynaptic weight change ${\Delta w_i}^{STDP}$ for a neuron i is primarily based on the firing time t_i of that neuron i, but is also based on whether or not presynaptic spikes exist with a greater firing time than t_i. Since the overall goal is to reduce the error of the readout, ${\Delta w_i}^{STDP}$ is also dependant on the readout error E_p of the readout pool P as defined in Equation 7.19. The presynaptic weight change ${\Delta w_i}^{STDP}$ is applied only to those presynaptic weights which carry a spike time greater than the postsynaptic spike time, and is defined for both liquid and readout neurons as:

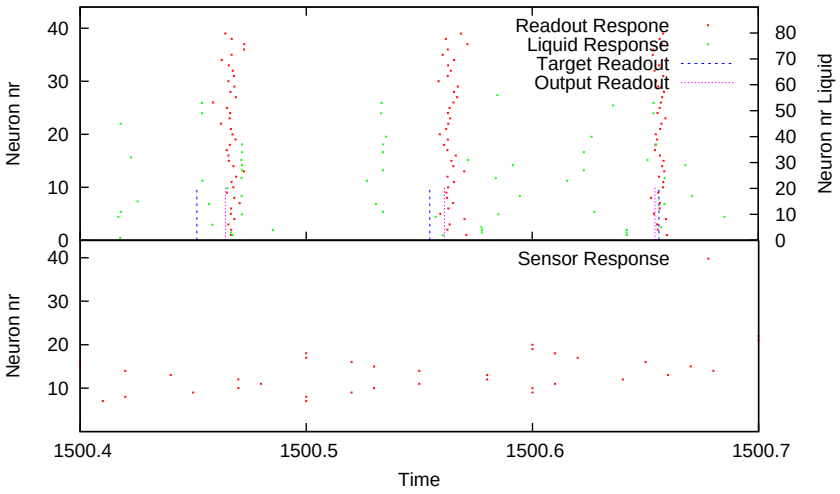


Figure 7.14: An overlapping plot of the liquid firing times and the readout firing times at the top. The vertical bars indicate the readout target and the readout output. At the bottom of the figure the sensor activity can be seen.

$$\Delta w_{ij}^{STDP} = \eta_{stdp}\, exp(-(1.0 - t_i^{norm})/C_{stdp})\, Y_{ij}\, E_p, \tag{7.32}$$

where C_{stdp} is a constant which determines how much weight change is preferred to diversify the firing times, Y_{ij} is the potential contribution of the connection from neuron j to i as defined by Equation 7.13, and t_i^{norm} is the normalized spike time used for computational convenience to acquire a value between 0 and 1. The normalized spike time is defined as:

$$t_i^{norm} = \frac{(t_i - t_{min})}{(t_{max} - t_{min})}, \tag{7.33}$$

where t_{max} is the largest firing time, t_{min} is the smallest firing time within the neuronal structure. This means that t_{max} for the readout is the maximum firing time of all the readout neurons, and t_{max} for a liquid neuron is the maximum firing time of all the liquid neurons.

In Figure 7.12 the results can be seen for $C_{STDP} = 0.5$ where only the readout neurons are adjusted. We can see some more variation in the firing

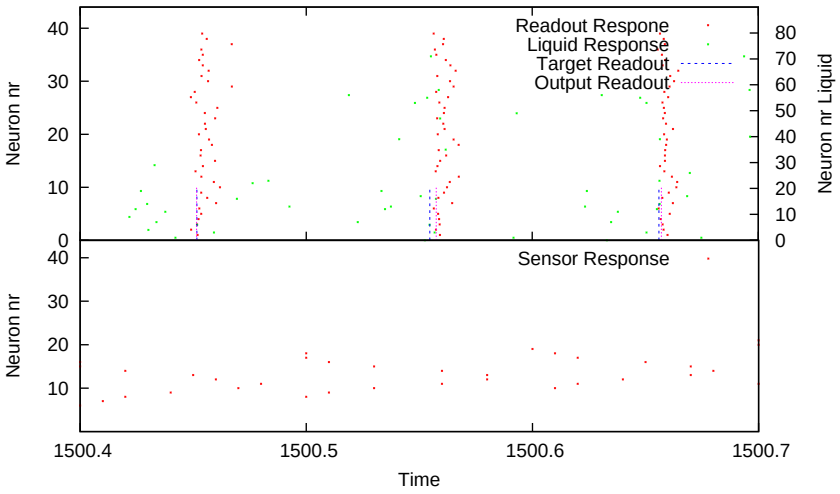


Figure 7.15: An overlapping plot of the liquid firing times and the readout firing times at the top. The vertical bars indicate the readout target and the readout output. At the bottom of the figure the sensor activity can be seen. The value for C_{STDP} at the liquid level is 0.1 and C_{STDP} is 1.0 for the readout neurons.

times of the readout neurons for a single input. Note that the vertical bars indicate the readout output and readout target. It can be seen that the error is very low. It can also be observed that many liquid spikes do not participate in the readout output since they occur when all the readout spikes have been emitted. In Figure 7.13 it can be seen that for a higher value $C_{STDP} = 1.0$, more variation in the readout spike times exits. The number of liquid firing times which contribute to the output of the readout varies greatly. In the interval [1500.4,1500.5] fewer liquid spikes can be labeled as effective than in the interval [1500.4,1500.5] which can be interpreted as either a positive or a negative characteristic as explained in Section 7.4.5. In Figure 7.14 the variety of readout responses is even greater for $C_{STDP} = 2.0$, however, it can also be seen that the error for the inputs is greater. We believe that this variety is a good characteristic and could be beneficial for the precision of the network but it takes longer to learn the input pattern. In Figure 7.15 both the liquid and readout neurons are subject to the changes based on

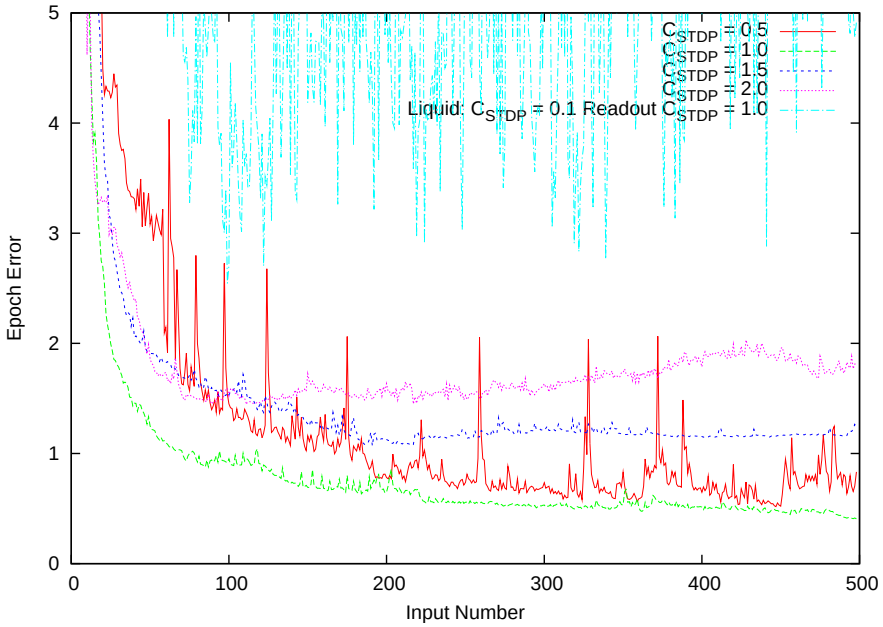


Figure 7.16: Epoch error for several values of the parameter C_{STDP}

Equation 7.32. It can be seen that fewer liquid neurons are active. This is due to the fact that when neurons are encouraged to fire at a later time the presynaptic potential has to be decreased. When the presynaptic potential falls below the threshold, the neuron no longer fires. Equations 7.16 and 7.14 are then applied such that the neuron will start to fire again. However, this disrupts the learning process and negatively impacts the error of the readout. It can be seen in Figure 7.16 that this has quite a negative impact on the learning ability of the network. Figure 7.16 also shows that an optimum exists. A low value of C_{STDP} of 0.5 performs poorly, while very high values do not perform well at all. For $C_{STDP} = 1.0$, a stable learning curve can be observed. Note that the amount of spikes in both the sensor and the liquid could be of great importance in determining the optimal values of C_{STDP}. On top of that, the number of readout spikes which are used to average over, could also be of great influence on the optimal value of C_{STDP}. However we will not pursue to investigate the influence of the number of spikes on the optimal value of C_{STDP}.

7.4.5 Effective Spikes

In Section 7.4.4 an experiment was described where late firing times were increased in order to more efficiently make use of many connections. The reason behind this was that the spiking nature of the neuronal model used adds a time dimension which can be used to encode information but does not seem to be used effectively. Note that this time dimension is not present in traditional sigmoidal neural models. This time dimension adds a complexity to the network with respect to coding information through the firing times of the liquid. Once a readout neuron spikes it will not use subsequent spikes to determine the output. A readout neuron only fires once as is described in Section 7.3.1. This means that some connections with a firing time which is greater that the firing time of the postsynaptic neuron, will not be effectively used in determining the output for a particular input. In Section 7.4.4 we tried to combat this phenomenon by leveraging the fact that we average the firing times of the readout. Now we want to investigate whether or not the lack of effectiveness of connections can also be an advantage.

Some connections might be used very infrequently. We could gain an advantage by adjusting those effective presynaptic weights which are specific to this input. A connection is specific to a certain input when the fraction of the inputs where the connection is actually effective, is quite low. When dominantly adjusting those connections which are specific to the current input, we reduce the crosstalk so that other inputs will be less influenced by adjusting connections which are specific to this input. This is a possibility due to two reasons. The first is that we use specialization at a neuronal level. Only a fraction of all the neurons is allowed to be active for any given input. The second reason is due to the timing of the spikes. After a postsynaptic neuron fires, any subsequent presynaptic spikes will not contribute to the postsynaptic output. This means that these connections are not effective for this particular input. The consequence of this is that another input exists, for which this connection might be more specific. The effectiveness score, which is defined as the normalized fraction of effective connections, is defined as:

$$\chi_{ij} = \frac{(s_{ij}/s_{ij}^{tot}) - \chi_{min}}{(\chi_{max} - \chi_{min})}, \tag{7.34}$$

where s_{ij} is the number of presynaptic spikes for which the connection between presynaptic neuron j to postsynaptic neuron i has been effective, s_{ij}^{tot} is the total number of inputs the presynaptic neuron j has produced a spike received by postsynaptic neuron i and χ_{min} and χ_{max} are the minimum and maximum effective score in order to create a value between 0 and 1 for the effective score χ. We define the effectiveness factor η_{ij}^{χ} which is specific to

the connection between neuron i and j, which determines how much this connection will be adjusted as a result of how specialistic this connection is for the current input, as:

$$\eta_{ij}^{\chi} = \eta_{\chi}\, exp(-(1.0 - \chi_{ij})/C_{\chi}) \tag{7.35}$$

where η_{χ} could be considered to be the learning rate for establishing the connection specific factor η_{ij}^{χ}. The reason for scaling the effectiveness factor is that the effectiveness factor η_{ij}^{χ} itself scales per connection how much the weight will be modified. This means that all weight modifications will be scaled by a factor between 0 and 1 which will slow down the overall learning speed. We wish to combat this by modifying those increasing the weight change for effective spikes and decreasing the weight change for weights with a low effective score. We set η_{χ} such that the integral of the effectiveness score will equal the constant C_{scale}. This is defined as:

$$C_{scale} = \eta_{\chi} \int_0^1 exp(-(1.0 - x)/C_{\chi})dx, \tag{7.36}$$

where the constant C_{scale} determines how much faster the weights with high effective scores will be modified. On top of that, even the factor η_{ij}^{χ} is a learning rate which is a function of the effectiveness factor χ_{ij}. Several previously defined adaptation rules will use this effectiveness factor as a learning rate. The adaptation rule defined by Equation 7.14 to adapt for the lifetime sparseness will then become:

$$\Delta w_{ij} = -\eta_T\, Y_{ij}\, w_{ij}\, E_T\, \eta_{ij}^{\chi}, \tag{7.37}$$

the adaptation rule defined by Equation 7.16 to adapt for the population sparseness will then become:

$$\Delta w_{ij} = \begin{cases} \eta_{pop}\, Y_{ij}\, w_{ij}\, E_{pop}\, \eta_{ij}^{\chi} & \text{if } S_{max} - S_L^i > C_{min}^{\ell} \text{ and } E_{pop} > 0 \\ & \text{or } S_L^i - S_{min} > C_{max}^{\ell} \text{ and } E_{pop} < 0, \\ 0 & \text{otherwise,} \end{cases} \tag{7.38}$$

the adaptation rule defined by Equation 7.22 to adapt the presynaptic weights in order to reduce the error of the readout will then become:

$$\Delta w_{ij} = \begin{cases} \eta_P\, E_i\, w_{ij}\, Y_{ij}\, \eta_{ij}^{\chi} & \vartheta : x_i(t) \;\geq\; \vartheta \text{ and } i \in P, \\ \eta_P\, (t_{max} - t_{min})\; w_{ij}\, Y_{ij}\, \eta_{ij}^{\chi} & \vartheta : x_i(t) \;<\; \vartheta \text{ and } \Gamma_o \text{ is incomplete,} \end{cases} \tag{7.39}$$

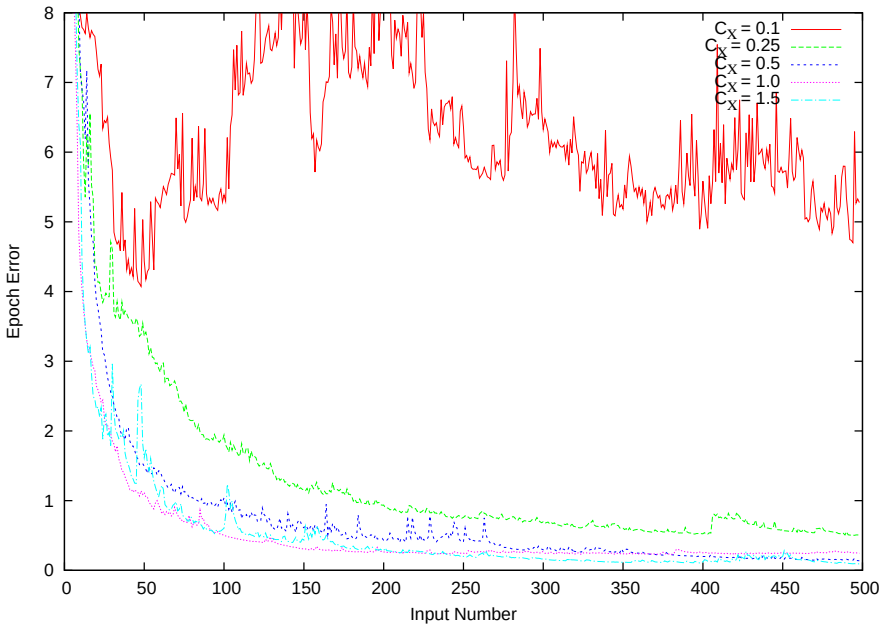


Figure 7.17: Epoch error for several values of the parameter C_χ

and finally, the adaptation rule defined by Equation 7.27 to adapt the presynaptic weights within the liquid in order to reduce the error of the readout will then become:

$$\Delta w_{ij} = \eta \, \delta_i \, Y_{ij} \, r \, \eta_{ij}^{\chi}. \tag{7.40}$$

In Figure 7.17 it can be seen that for higher values of the parameter C_χ the learning speed improves considerably. We believe that this can be explained by the fact that crosstalk is reduced between the input patterns when weights, which are specific to the current input, are dominantly adjusted over weights which are not specific to the current input.

7.4.6 Increasing the Variety of Neural Responses by Increasing Orthogonality

In Section 7.3.3 it was described how the firing times of the liquid neurons are adapted in order to reduce the readout error. Although we do not use the well-known back-propagation algorithm, it still has all the characteristics of a gradient descent algorithm. Gradient descent algorithms have a tendency to get stuck in local minima and therefore are not an optimal way to train a neural network. In order to combat this we introduce here a method, which

combined with the gradient descent method, will improve the learning speed. It has been stated in [19] that a great amount of variety must exist between the responses of the individual neurons. This can be done by inhomogeneous connectivity, random weight initialization and by varying the neuronal parameters. We think that a great variation in neuronal responses will reduce the network's tendency to get stuck in local minima when training on input data. We adopt the notion of a great variance in neuronal responses into our framework of learning rules. We believe we can increase the variety of neural responses within the network by introducing a learning rule which will shift the focus of the neurons to different parts of the input. By shifting the focus it is meant that every neuron is dominantly excited by a different part of the input, where the input of a liquid neuron can be either sensor or other liquid neurons. We aim to accomplish this by altering the weight vectors of the liquid neurons in such a way that every liquid neuron perceives the input being presented to the network differently. We believe this can be achieved by considering the weight vectors of the liquid as a base. When this base moves towards an orthogonal base each neuron will perceive the same input differently which is reflected in its postsynaptic potential and ultimately in its firing time. This result can be explained by the mutually exclusive nature of an orthogonal base.

Since not all neurons have the same number of presynaptic weights we define a weight vector $\vec{w_i}$ for neuron i with the same number of dimensions as the number of neurons combined in the sensor and liquid structure. When no connection exists between neuron i and j the jth entry of the weight vector $\vec{w_i}$ is 0, otherwise the jth entry will be specified with the weight between neuron i and neuron j. In order to increase the orthogonality of two weight vectors $\vec{w_i}$ and $\vec{w_j}$, we need to split one of the vectors, in this example $\vec{w_j}$ into the components into the direction $\vec{w_i}$ and into a direction perpendicular to $\vec{w_i}$ as is depicted in Figure 7.18. The projection $Proj_{\vec{w_i}}\vec{w_j}$ is the component of the vector w_j onto the vector $\vec{w_i}$ and is defined as:

$$
\begin{aligned}
Proj_{\vec{w_i}}\vec{w_j} &= \lambda\,\vec{w_i} \\
Proj'_{\vec{w_i}}\vec{w_j} &= \vec{w_j} \;-\; \lambda\,\vec{w_i} \perp \vec{w_i} \\
&\Leftrightarrow 0 \;=< \vec{w_j} - \lambda\,\vec{w_i}, \vec{w_i} > \\
&\qquad =< \vec{w_j}, \vec{w_i} > \;-\; \lambda \;< \vec{w_i}, \vec{w_i} > \\
&\Leftrightarrow \lambda \;=\; \frac{< \vec{w_i}, \vec{w_j} >}{< \vec{w_i}, \vec{w_i} >} \\
Proj_{\vec{w_i}}\vec{w_j} &= \frac{\vec{w_i} \;\cdot\; \vec{w_j}}{|\vec{w_i}|^2}\,\vec{w_i}
\end{aligned}
\tag{7.41}
$$

The component of the vector $\vec{w_j}$ perpendicular to the vector $\vec{w_i}$, as depicted in Figure 7.18, is then defined as:

$$Proj'_{\vec{w_i}}\vec{w_j} = \vec{w_j} - \frac{\vec{w_i} \cdot \vec{w_j}}{|\vec{w_i}|^2}\vec{w_i} \perp \vec{w_i}. \tag{7.42}$$

Note that we do not store negative values in our network, an inhibitory presynaptic neuron will be specified in the weight vector as a positive number. We can justify this by stating that the inhibitory characteristic is defined at the level of a neuron and all neurons connected to an inhibitory neuron will perceive its activity in the same manner. The fact that only positive numbers are stored in the vectors makes that the maximum angle between two weight vectors is 90 degrees. Of course we do not want to limit the search space with orthogonal weight vectors alone. However we believe that increasing orthogonality will increase the networks's performance. Therefore we want to slowly increase the orthogonality of the weight vectors of the individual neurons as long as the error of the readout neurons is not within the tolerable error. We apply the Gram-Schmidt process to the weight vectors of a random selection of k neurons in the liquid to find an orthogonal base. The Gram-Schmidt Orthogonalization algorithm is defined as:

$$\begin{aligned}
\vec{u_1} &= \vec{w_1} \\
\vec{u_2} &= \vec{w_2} - Proj_{\vec{u1}}\vec{w_2} \\
&\vdots \\
\vec{u_k} &= \vec{w_k} - \sum_{j=1}^{k-1} Proj_{\vec{u_j}}\vec{w_k},
\end{aligned} \tag{7.43}$$

where $\vec{u_k}$ is the kth vector in the orthogonal base derived from vectors $\vec{w_1}, \vec{w_2}, \ldots, \vec{w_k}$ which have been randomly selected from the liquid. Only those neurons which have emitted a spike are subject to this part of our learning framework. The fact that connections within the liquid and connections from the sensor to the liquid are based on the locations of the neurons, makes sure that neurons which are active are relatively close together within the liquid. This also means that the neurons which are selected for the orthogonalization process received the same or very similar input spike trains. It would be highly decremental to the performance of the network to alter every weight vector such that all the weight vectors form an orthogonal base. Neurons would not receive enough presynaptic current to emit a postsynaptic spike. The goal is to select k neurons whose weight vectors will slowly be moved towards orthogonality. A true orthogonal base will not be achieved. Increasing

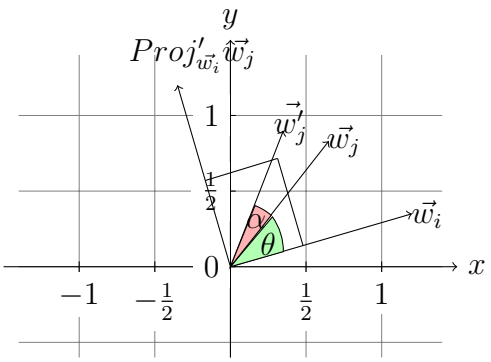


Figure 7.18: Increasing the orthogonality between vectors $\vec{w}_i$ and $\vec{w}_j$ results in $\vec{w'}_j$.

the orthogonality is done by calculating the orthogonal base, for a randomly selected collection of neurons which have emitted a spike for the particular input, and move each weight vector into the direction of its orthogonal counterpart. We increase the angle between the vectors $\vec{w_k}$ and its orthogonal counterpart $\vec{u_k}$, by moving vector w_k slightly into the direction of vector u_k. To accomplish this we define the weight change as:

$$\Delta w_k = -\eta_o \, (\vec{w_k} - \vec{u_k}) \, E_P \, y_p(t_k), \tag{7.44}$$

where Δw_k specifies the weight change for the entire weight vector, η_o is the orthogonalizing learning rate, E_P is the readout error and $y_p(t_k)$ is the potential contribution of the pth element in vector w_k for postsynaptic neuron k. By increasing the orthogonality of the weight vectors of all active neurons in the network, all postsynaptic neurons will develop unique presynaptic weight vectors since different presynaptic weights will be adjusted closer to zero as a result of increasing orthogonality. Foremost, this will increase the uniqueness of a neuron's response to an input. However, as a result of the temporal nature of the input spike trains combined with the increasing orthogonality of the weight vectors, we believe that this will enable different neurons to be dominantly excited by different parts of the same input spike train. The result is that the information encoded within the incoming spike train is absorbed into the population of neurons more efficiently.

In Figure 7.19 it can be seen that the adaptation rule defined by Equation 7.44 has a positive influence on the learning ability. For higher values of the learning rate η_o it can be seen that the precision of the network increases across the entire training phase of the network. Although at first the error is more erratic when applying Equation 7.44, the output of the network

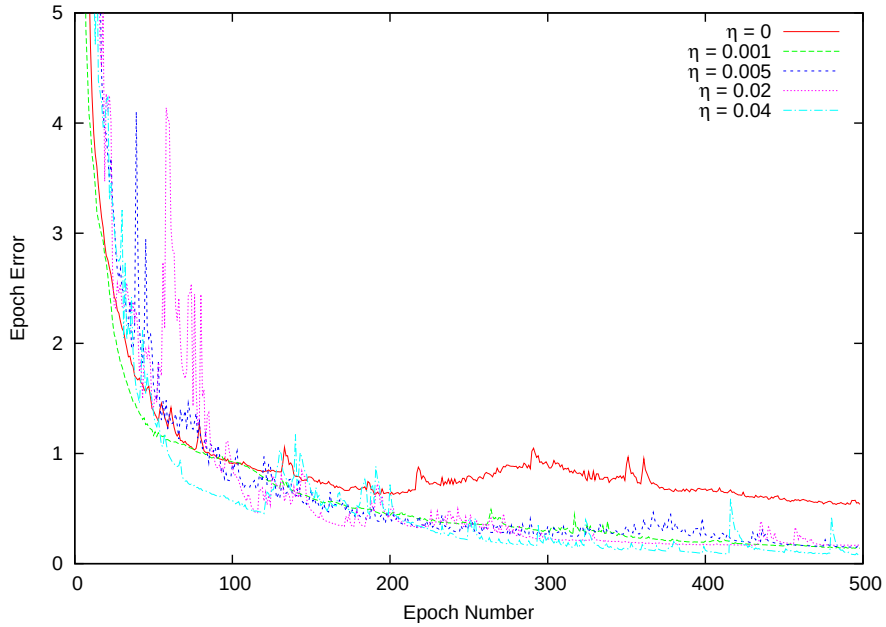


Figure 7.19: Epoch error for several values of the orthogonal learning rate η_o.

stabilizes and the error becomes a smooth line which moves towards zero. Note that higher learning rates of the orthogonalizing process cause more fluctuations in the error at first but do stabilize. It seems that the maximum learning rate for which the network behaves reliably is about 0.04 as can be seen in Figure 7.19. We conclude that the process of orthogonalizing the weight vectors is a good way to avoid getting stuck in local minima. The reason for this is that neurons specialize highly efficiently. Neurons absorb different subsets of all available input information, for a single input, in such a way that the union of all these subsets makes up all of the available input information. All input information is distributed across all of the active neurons.

7.4.7 Introducing Recurrent Connections

In this section we will show the results of our experiments when we add recurrent connections to the liquid. First we will describe the input data used in the experiments. Then we will demonstrate the liquid's ability to retain information about past inputs by training the readout to output the previous input. Then we will demonstrate the liquid's ability to detect a pattern by predicting the next input based on previous inputs. The network we trained in the experiments used a reservoir which consisted of 135 neurons, a sensor

of 60 neurons and a readout which consisted of 60 neurons.

7.4.8 XOR Data Set

The input data we use is derived from the XOR pattern. This pattern is especially interesting since it can only be learned by a three layer feed forward network. This data will thus be suited to initially pose a great enough challenge for a liquid state machine to investigate some basic properties. We choose to translate XOR input data into temporal data stream by the example of [12]. Two randomly generated input bits are being placed one after the other followed by the output bit. An example input string would be:

$$110011000110011101101110 10.$$

When a readout is being trained to find a pattern in the liquid states that are the result of the input stream, it will need to output the result based on both the current input and the previous input. This input data is especially interesting since it is only possible to predict every third bit. A correct result cannot be predicted for every first and second bit since these bits are generated randomly. This increases the complexity when training the network on this input string. We generated an input string which consists of 51 bits.

7.4.9 Creating a Time Dimension within the Liquid Structure

In order to pass information about the previous input when the current input is being presented to the network, we created a new kind of connection within the liquid. Traditionally in recurrent neural networks the outputs of the neurons in the hidden layer are copied to an additional input. This way the output of the hidden layer at time t is being used as an input at time $t + 1$ as is described in Section 2.3. Since the whole liquid is a hidden layer we found it to be inflexible to simply copy the firing times of the liquid and have an additional input with as many neurons as there are in the liquid. To gain some flexibility with respect to controlling how much information from past inputs is present within the liquid, we added a new set of connections which have a delay as long as the input time window T. We will call these connections simply *recurrent connections*. The recurrent connections are created in the same way as the non-recurrent connections, which is described in Section 7.1.1. The probability of a connection between neurons i and j is determined by a parameter λ_R and specified by:

$$C \cdot exp(-D(i,j)^2/\lambda_R^2),$$

where $D(i,j)^2$ is the Euclidean distance between neurons i and j and C scales the probability for creating a connection. The parameter λ_R controls the connectivity density as well as the average distance between the connections. We set λ_R as well as λ_L to 1.3, where λ_L defines the connection parameter for non recurrent connections within the liquid. We then defined two parameters P_C and P_R, where P_C specifies the percentage of liquid neurons which have incoming connections which are not recurrent, and P_R defines the percentage of neurons which have recurrent incoming connections. Initially we set both P_C and P_R to 1.0, varying these parameters could impact the ratio between information about the current input and information about past inputs.

Discriminatory property

In this section we will introduce a method how we can gain an insight into the discriminatory property of the liquid, i.e., we will describe a method to measure the differences of the resulting liquid states. We will focus especially on the effect the recurrent input connections have on the differences in the resulting liquid states for the same input value. We assume that measuring the distances between liquid states will give us an indication of the ability of the liquid to function as a fading memory. We also want to determine whether we actually can measure the ability of the liquid to function as a fading memory by comparing the results in later experiments.

The memory of the liquid is dynamic in the fact that the state of the column of neurons contains information about the last several recent inputs as described in Section 5.3. The rationale is that the liquid state at the beginning of each new input determines the response of the liquid to that input and in turn impacts the response of the next input. Therefore the response of an input is determined by the initial condition of the liquid as well as the value of the input itself. The liquid state is determined through a set of variables. First of all the voltage of each neuron at the beginning of an input impacts the response of the liquid to an input. Secondly the time of the last incoming spike for each neuron has an effect on the response for an input, due to the refractory effects. The last way for an input to have an effect on the future inputs, which is described in Section 2.3.2, is to input the liquid state as a result of the previous input. In the experiments we will especially focus on this effect of recurrent inputs.

We call the resulting liquid state $L_t(\xi)$ at time t as a function of input ξ, a *base liquid state* when input ξ is being presented to the liquid when the liquid is at rest. We call a resulting liquid state, $L_t(\xi)$ at time t, due to an input ξ when the liquid was not at rest an *accumulated liquid state*, since the perturbations of previous inputs are present in the current liquid state. The

foundation of the liquid as a dynamic memory buffer lies in the fact that an input ξ^t at time t has an influence on the response of the liquid for input ξ^{t+1} since the liquid state is in an accumulated state as a result input ξ^t when input ξ^{t+1} is being presented. In turn, input ξ^{t+1} causes the liquid to be in an accumulated state when input ξ^{t+2} is being presented. When the liquid state does not return to its resting state, in theory no two liquid states will be the same [21] and is called computing without stable states. We define $\Gamma^{L(\xi)}$ to be the set of liquid states which were the result of a single input value ξ. Note that the liquid states in the set $\Gamma^{L(\xi)}$ can be very different due to the fact that the liquid acts as a dynamic memory buffer. We define the set $\Gamma^{L(\xi_k)}$ as the set of liquid states resulting from input values $\xi_1, \xi_2, \ldots$, which can be called the history of liquid states. In this experiment we wish to gain insight into the dynamic memory capabilities of the liquid by measuring the difference between $L(\xi)$ and all the possible accumulated liquid states for input ξ, $\Gamma^{L(\xi)}$. We define the liquid state as a result of input ξ at time t_k as $L_{t_k}(\xi)$. It is important that the difference between the states $L_{t_k}(\xi)$ in $\Gamma^{L(\xi)}$ is large enough so the readouts have information about the previous state hidden in the difference between the accumulated liquid states. In [21] this is called the *separation property*. A readout needs to have sufficient information in a liquid state to map the state to a certain target value. However, when the difference between the states in $\Gamma^{L(\xi)}$ is too great the differences between all the accumulated liquid states for an input ξ will be too great for a readout to learn the correct output value. When every liquid state is completely different from all the previous liquid states, nothing sensible can be said about the associated target value. In [21] this is *called the separation property.*

The consequence of this fact is that every distinct input value ξ produces a set of different liquid state $\Gamma^{L(\xi)}$ as a result of the input ξ. We have chosen to define the distance between two liquid states $L(\xi_1)$ and $L(\xi_2)$ in two distinct ways. The first method to calculate the distance between two liquid states focuses on the first spike time of every neuron within the liquid and calculates the distance between two liquid states $L(\xi_1)$ and $L(\xi_2)$ as:

$$D_1(L(\xi_1), L(\xi_2)) = \sum_i (t_i(\xi_1) - t_i(\xi_2)), \tag{7.45}$$

where $t_i(\xi_1)$ is the time of the *first* spike by neuron i as part of the liquid state $L(\xi_1)$ as a function of input ξ_1 and K is the number of past inputs we wish to calculate the average distance over. The average distance D_1^{AV} between the set of liquid states $\Gamma^{L(\xi_k)}$ as a result of input value ξ_k is then defined as:

$$D_1^{AV} = \frac{\sum_{k=1}^{K} \sum_{\ell=1}^{k} D_1(L(\xi_k), L(\xi_\ell))}{\frac{1}{2}\, K\, (K\, +\, 1)^2}. \tag{7.46}$$

The second method to calculate the distance between two liquid states determines the distance based on the entire spike train of each neuron in the liquid, and is defined as:

$$D_2(L(\xi_1), L(\xi_2)) = \sum_i \sum_f (\int \varepsilon(t\, -\, t_i^f(\xi_1))\, dt\, -\, \int \varepsilon(t\, -\, t_i^f(\xi_2))\, dt\,) \tag{7.47}$$

where $\varepsilon(t)$ is the spike response function as defined in Equation 6.2, $t_i^f(\xi_1)$ is the time of the fth spike by neuron i as a function of input ξ_1, as part of the liquid state $L(\xi_1)$. The average distance D_2^{AV} between the set of resulting liquid states $L(\xi_i)$ as a result of input value ξ_i is then defined as:

$$D_2^{AV} = \frac{\sum_{k=1}^{K} \sum_{\ell=1}^{k} D_2(L(\xi_k), L(\xi_\ell))}{\frac{1}{2}\, K\, (K\, +\, 1)^2}. \tag{7.48}$$

The values D_1^{AV} and D_2^{AV} is significant especially when investigating a sparse neural code. For a good performance of the liquid state machine it is crucial that the readouts can recognize a liquid state as fast as possible after the input was presented to the liquid. The distance measure as defined in Equations 7.46 and 7.48 will give an indication of this property.

The final method we will look at, in order to gain insight into the extent in which past inputs influence the response of the liquid to the current input, is what we would like to call the deviation of the individual neurons in the liquid. We define the deviation Ξ of a neuron in the liquid as the difference between the smallest firing time $t_i^{min}(\xi)$ and the largest firing time $t_i^{max}(\xi)$ in $\Gamma^{L(\xi)}$, as a function of a single input value ξ for a neuron i. The deviation Ξ for a neuron i is then calculated as:

$$\Xi = t_i^{max}(\xi) - t_i^{min}(\xi), \tag{7.49}$$

where ξ is the distinct input value for which the minimum and maximum firing time for a neuron i are recorded. The greater the deviation, the greater the influence of past inputs onto the response of the liquid to the current input.

7.4.10 Information about Past Inputs

In this section we will show the network's ability to store information about past inputs when recurrent connections are present. In Figure 7.20 the epoch

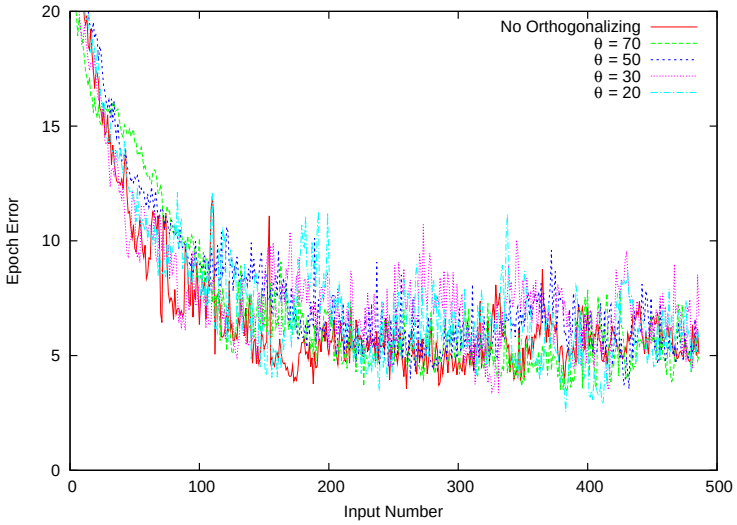


Figure 7.20: Epoch error for several values of the maximum angle between weight vectors.

error is displayed when the network is being trained to output the previous input when the current input is being propagated through the network. This is difficult since at any point in time the network contains information about all past inputs. Each time the entire input string has been processed by the network, the liquid is reset, which means that at the first input bit of the string the liquid state is at rest. This creates a different liquid state for each input bit, even though only two different values exist in the input data.

In Section 7.4.4 it was described that the timing element can be a limiting factor for training a network. We noticed that this is indeed the case for the results in Figure 7.16. Every neuron in the liquid fires once. This means that when a neuron fires any additional incoming spikes do not participate in the neurons response. When the neurons in the liquid receive predominantly spikes from non-recurrent connections, no information about past inputs is being processed by the postsynaptic neuron. In the other case, where neurons predominantly receive spikes from recurrent connections only information about the past inputs is being processed and in future inputs no information about the current input is present. This can be translated to a general problem where neurons need a mechanism to determine to which input spikes the postsynaptic firing time needs to based when more than a single input is present. That is why it is very important that every neuron receives a mix of spikes form non-recurrent and recurrent connections. This can be established by the orthogonalizing rule as defined by Equation 7.44.

The larger the maximum angle between the weight vectors in the liquid, the more specialized is the input the neurons receive. In Section 7.4.6 we stated that this specialization of neurons was favorable to the learning ability of the network. However, we establish now that this specialization can be harmful for training the network when two or more inputs are being input to the network. We believe an optimum can be found. Note that this issue arises out of the fact that a selection of neurons fire only once per input. This sparse neural code is our main research objective. We favor a sparse code because of computational efficiency. Furthermore related to the timing issues is that we found we needed to scale the firing times which are being propagated to other neurons across recurrent connections since the liquid firing times were always later than the sensor firing times in such a way the the recurrent firing times were unable to participate in the postsynaptic potential. Therefore we introduce a parameter S_{FT} which we set to 0.3, which scales a firing time relative to the current time window.

In Figure 7.21 the results can be seen of training the network to output the previous input when every neuron which has outgoing recurrent connections outputs two spikes. Earlier in this Section we introduced the parameter S_{FT} to scale recurrent firing times. We now want to see what the influence of several recurrent spikes have on the performance of the network. We set S_{FT} to 0.1 which scales the firing time as:

$$t_i^{start} = t_i S_{FT}. \tag{7.50}$$

This reduces the spike time to a spike. Any timing information is lost on this scale. We encode this spike time into subsequent spikes by calculating a spike time interval which is calculated as:

$$t_{interval} = \frac{(TW - t_i^{start})}{C_N} \frac{t_i}{TW}, \tag{7.51}$$

where C_N is a constant which determines how many subsequent spikes are generated after the start spike. In Figure 7.21 the results can be seen when neurons which have outgoing recurrent connections generate spike trains as defined by Equation 7.50 and 7.51. Again the timing of presynaptic spikes proved to be crucial and a limiting factor to the performance of the liquid. Note that we want to achieves sparse neural code and ideally wish to prevent multiple spikes for a single input. The result is only slightly better, which raises the question if multiple spikes per neuron for a single input can improve performance. We believe the solution is to be found in a mechanism for a neuron to choose its input spikes.

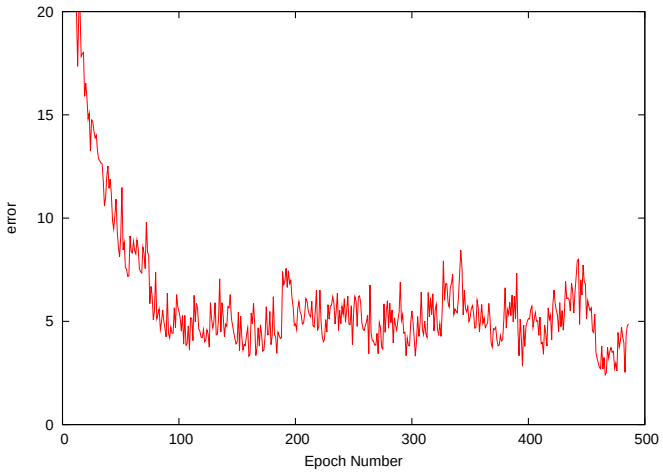


Figure 7.21: Epoch error when every spike with outgoing recurrent connections emits two spikes.

Orthogonality

In Section 7.4.6 it has been described how the efficiency of training the network can be improved by increasing the orthogonality of the weight vectors in the network. We found that when applying Equation 7.44 the networks performance can become unstable. This is due to the fact that when applying Equation 7.44 weights in the network are decreased in order to increase orthogonality. This causes the postsynaptic potential to drop below the spiking threshold. The postsynaptic potential will then be increased by Equations 7.16 and 7.14 which causes different neurons to emit spikes which in turn need to be trained all over again. We attempt to counteract this by establishing a maximum average angle α^{MAX} for the base of weight vectors which have been selected for the Gram-Schmidt process as described in Section 7.4.6. When for a base of k weight vectors $\vec{w_1}, \vec{w_2}, \ldots, \vec{w_k}$ the average angle α exceeds the predefined maximum angle α^{MAX}, the base of weight vectors will be reduced in orthogonality. This is done by moving all the weights which have participated in a postsynaptic spike, towards the average value of the all weights. When all weights have the same value, and thus all weight vectors are the same, the orthogonality is at a minimum. Several plots can be seen in Figure 7.20 for several values of the maximum angle between the weight vectors. It can be seen in Figure 7.20 that the previous input can be learned by the network quite well. However, it has to be noted that the performance of the network drops considerably compared to training a

non temporal problem and no significant improvement can be detected for different values of the maximum angle α^{MAX}.

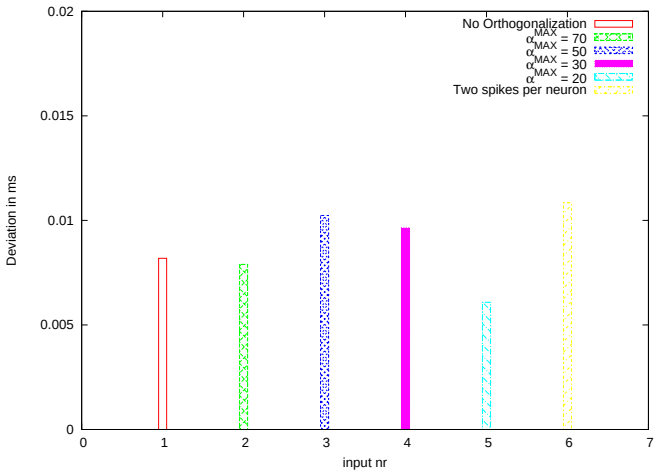
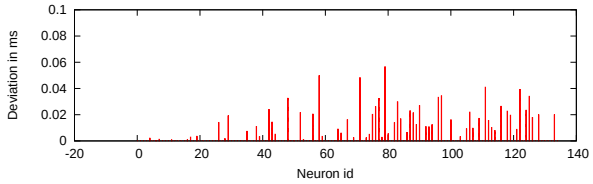


Figure 7.22: Deviation averaged across all neurons for several values of the maximum angle between weight vectors.

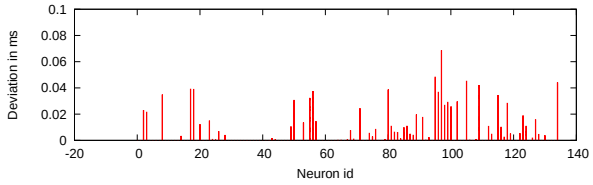
In Figure 7.23 the deviation per neuron is depicted, whereas in Figure 7.22 the deviation can be seen averaged across all liquid neurons. The deviation for a neuron is defined by Equation 7.49. The input data described in Section 7.4.8 has two distinct input values, i.e. 0 and 1. However, there are many times more different liquid states as a result of the input string consisting of only two values. This is due to the recurrent connections as described in Section 7.4.9. It can be seen that not all neurons have a significant deviation and some neurons do not have a deviation at all. This can be optimized by making sure all neurons in the liquid receive both recurrent and non-recurrent spikes. Neurons which do not show any deviation receive dominantly non recurrent spikes. It is important for the liquid neurons to display deviation since this will increase the learning ability of the liquid. In Figure 7.24 the distances are depicted between the liquid states as described in Section 7.4.9.

7.4.11 Information about Future Inputs

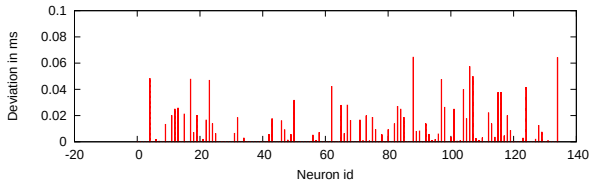
In this section the results are shown when the readout is trained to output the next input. As explained in Section 7.4.8 not every input can be predicted and this makes this problem very difficult. Adaptations in weights are made for every input and may disrupt the the internal representation of the network for these inputs which can be predicted. When training the



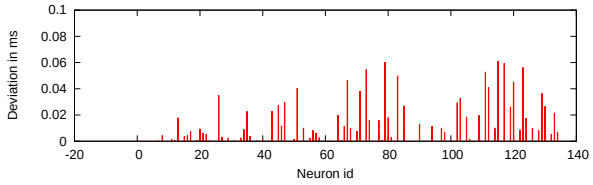
(a) No Orthogonalization



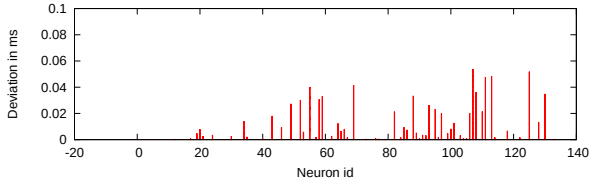
(b) $\alpha^{MAX} = 70$



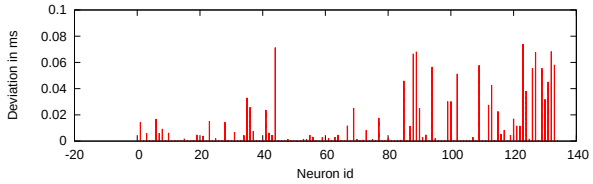
(c) $\alpha^{MAX} = 50$



(d) $\alpha^{MAX} = 30$



(e) $\alpha^{MAX} = 20$



(f) Two spikes per neuron

Figure 7.23: The deviation of all liquid neurons across all unique inputs for several values of the maximum angle α^{MAX}.

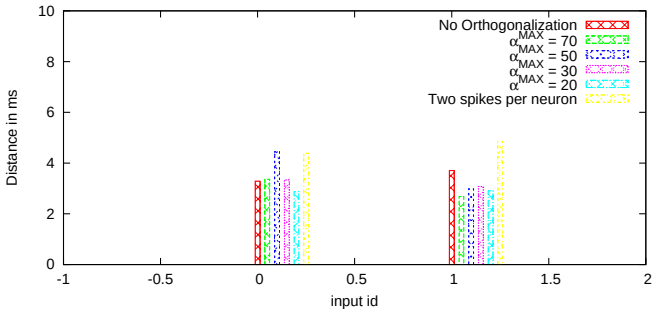


Figure 7.24: Distance per distinct input value.

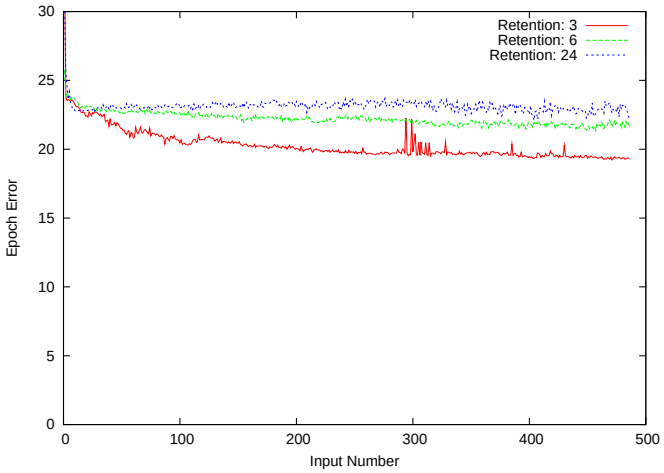


Figure 7.25: Epoch error for several values of retention.

network to predict the next input, a pattern can emerge which reveals itself in a cycle in the error pattern. It can be seen in Figure 7.26 that every second input has a low error since every third input can be predicted correctly. This sudden drop in error reveals a pattern in the input data. In Figure 7.28 the output of the readout is plotted against the output target. We found that the amount of history which is kept within the liquid is of great importance to the performance of the network. It has been described in Section 7.4.9 that the liquid response is, theoretically, not the same for any input value. This increases the search space tremendously and poses an extra challenge for the learning framework and the network to correctly output the intended pattern. In order to reduce the complexity of training the network we introduce a concept which we would like to call *retention* of data. The XOR pattern

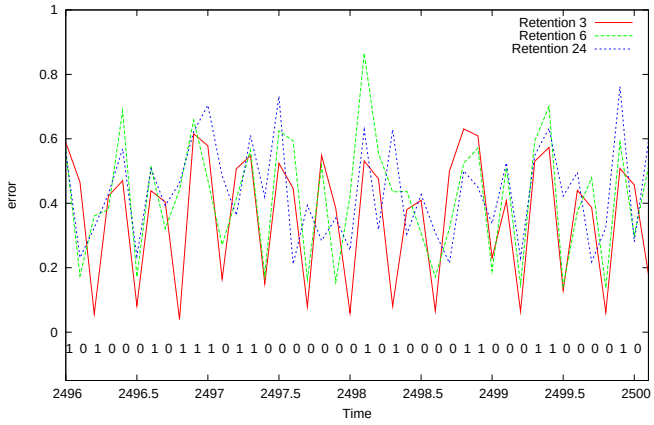


Figure 7.26: Error of the readout for several values of retention.

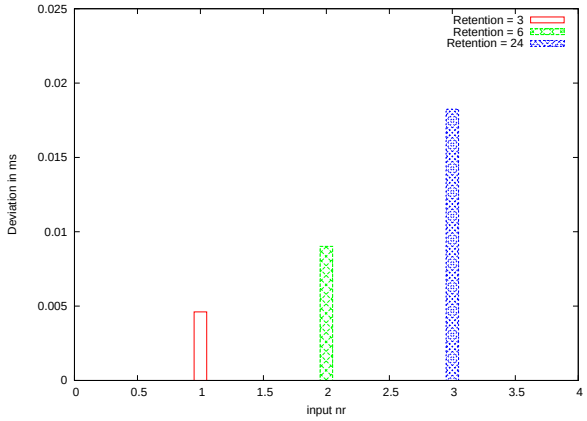
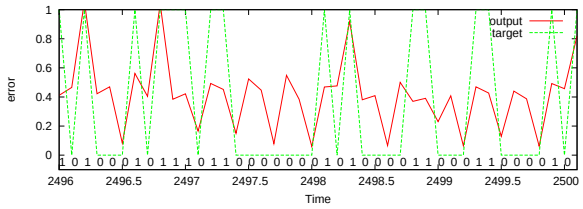
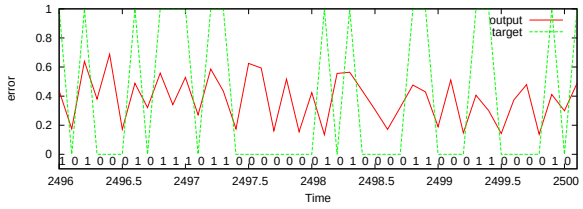


Figure 7.27: Deviation averaged per neuron for different values of retention

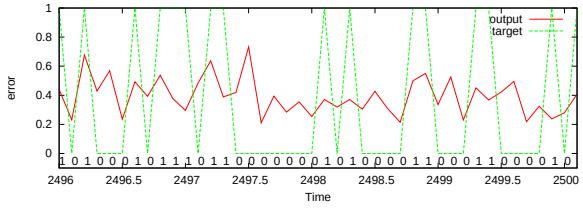
as described in Section 7.4.8 needs the previous and the current input to be able to predict the next input. Due to the recurrent connections every input will cause a different liquid response since both the current input and the input history determine the liquid response. This makes the state space much greater than when the liquid state is caused by the current input alone. The explosion of the state space compromises the ability of the network to be trained on an input pattern. When the input pattern is very short, as is the case with the input string described in Section 7.4.8, then this enormous state space is completely unnecessary and therefore inefficient. We therefore do not propagate recurrent spikes after a certain number of inputs, which we call the retention value. This resets the memory of the liquid. In Figure 7.25



(a) Retention 3



(b) Retention 6

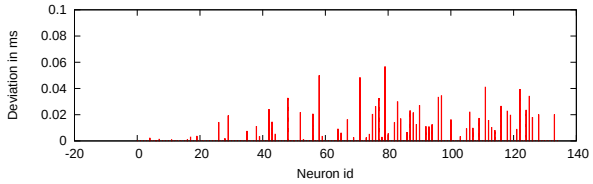


(c) Retention 24

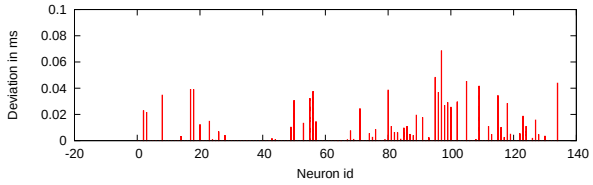
Figure 7.28: The output of the readout is shown together with the target output for several values of the retention.

the epoch error can be seen when the network is trained on the XOR input string for different values of retention. It can be seen that for a retention value of 3 the error is much lower than for retention values of 6 and 24. We expect the deviation of neurons to be much higher when the retention is high. This is clearly the case as can be seen in Figures 7.29 and 7.27. For higher values of retention it is much more difficult to predict the next input value as can be seen in Figures 7.26 and 7.28. It is described in Section 7.4.8 that only every third bit can be predicted correctly. In Figure 7.26 it can be seen that every third bit has a low error. Form this pattern in the error of the output can we derive that a pattern exists in the input.

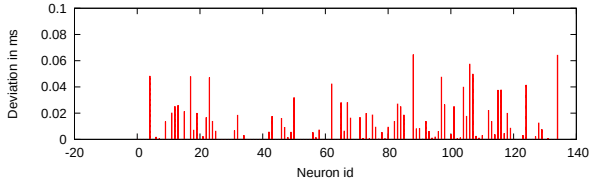
When training the network on a Mackey-Glass data set of 250 data points the results look promising. It can be seen in Figure 7.31 that the output of the network matches the target pattern quite well. In Figure 7.31(a) the output of the network is depicted in relation to the intended target output when



(a) Retention 3



(b) Retention 6



(c) Retention 24

Figure 7.29: The deviation of all liquid neurons across all unique inputs for several values of the retention.

training the network to predict the next input. In Figure 7.31(a) the results can be seen when the network is being trained to predict the input value ten inputs ahead. Surprisingly, in Figure 7.30 it can be seen that the overall error is lower when the network is trained to predict the input value ten inputs ahead that to predict the next input value. We think this is a property of this specific data set but implies that is not per se more difficult to predict inputs that lie further into the future.

7.5 Temporal Pattern Prediction

In this section we will demonstrate the applicability of our neural structure to more complex temporal data. We will train the network on an encoded string of text. Natural language has a temporal structure and is therefore well-suited as input data in our experiments. First we will describe the data and then we will present the results.

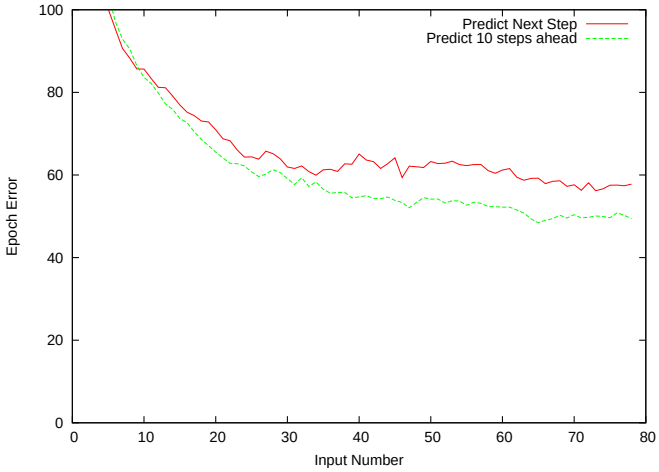
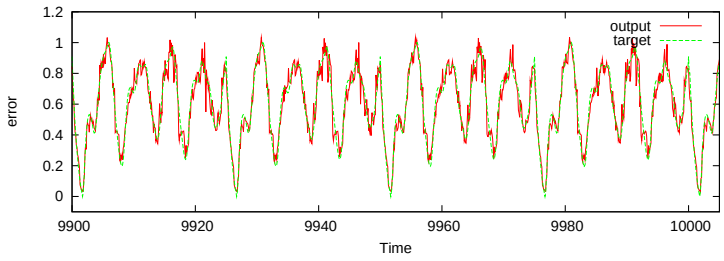
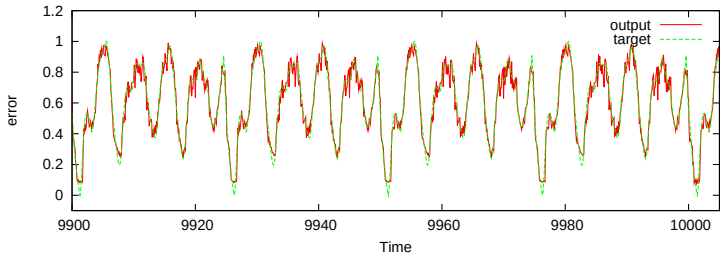


Figure 7.30: The Epoch error when predicting future inputs for the Mackey-Glass time series.



(a)



(b)

Figure 7.31: The output of the readout is shown together with the target output for the Mackey-Glass time series.

7.5.1 Data

We have chosen a string of 700 characters from the famous work of Charles Darwin, *Origin of Species*:

> *Owing to this struggle for life, any variation, however slight and from whatever cause proceeding, if it be in any degree profitable to an individual of any species, in its infinitely complex relations to other organic beings and to external nature, will tend to the preservation of that individual, and will generally be inherited by its offspring. The offspring, also, will thus have a better chance of surviving, for, of the many individuals of any species which are periodically born, but a small number can survive. I have called this principle, by which each slight variation, if useful, is preserved, by the term of Natural Selection, in order to mark its relation to man's power of selection.*

The string is encoded as depicted in Figure 7.32. The vowels in the string are replaced with numbers lower than 6, consonants will be replaced with a number between 6 and 26 and punctuation marks will be replaced with the numbers 27 through 29. This replacement scheme creates three groups of input values. The inputs can be characterized as a vowel, consonant or punctuation mark. Note that this will give us the ability to to train the network on different properties. We will train the network to predict the next group as well as the next input value.

7.5.2 Applicability

The final experiment we conducted is training the network on an encoded string of text. In Figure 7.34(a) the results can be seen when the network has been trained to predict the group the next input value belongs to. The three groups, vowels, consonants and punctuation marks have been described in Section 7.5.1. The results are not as good as the results on the Mackey-Glass data set. However, the data set is almost three times as large. The most noticeable element of Figure 7.34(a) is that the output does go into the right direction of the target but does not span the entire range of the output target. Note that only three different target values are present. Possibly more training is required. In Section 7.4.11 it has been described that

Input character	Encoded value
a	1
b	6
c	7
d	8
e	2
f	9
g	10
h	11
i	3
j	12
k	13
l	14
m	15
n	16
o	4
p	17
q	18
r	19
s	20
t	21
u	5
v	22
w	23
x	24
y	25
z	26
,	27
.	28
Space	29

Figure 7.32: To encode a string of text each letter is replaced with a number. Vowels, consonants and punctuation marks form a range of [0,5],[6,26] and [27,29] respectively.

not all neurons in the reservoir are influenced by previous inputs. This was shown in Figure 7.29 since not all neurons display a deviation for a distinct input value. We believe that this is the reason the results can be improved significantly.

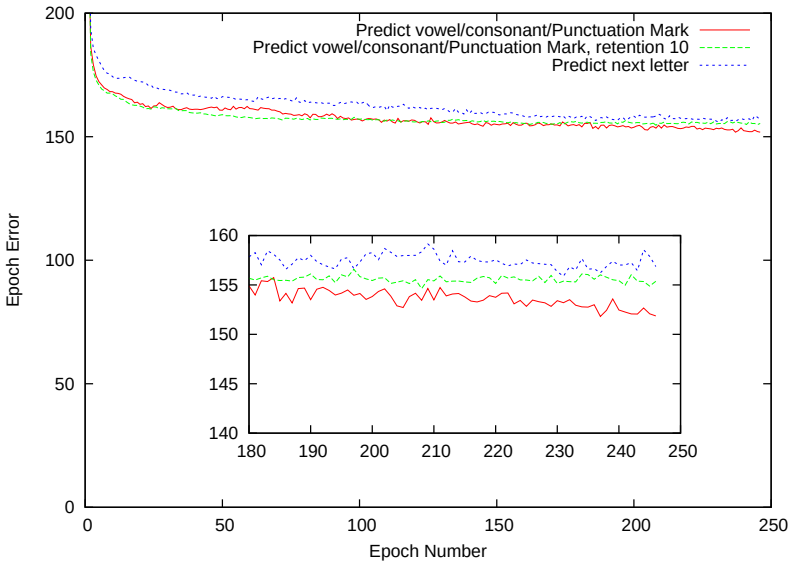
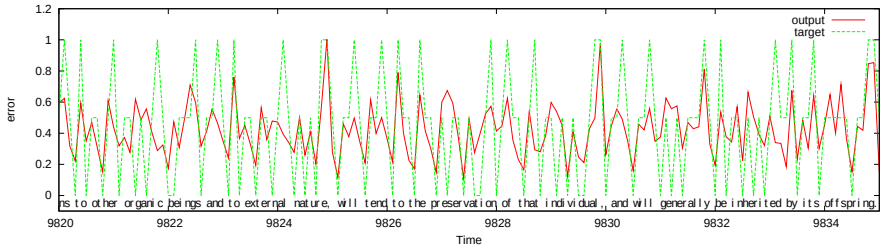
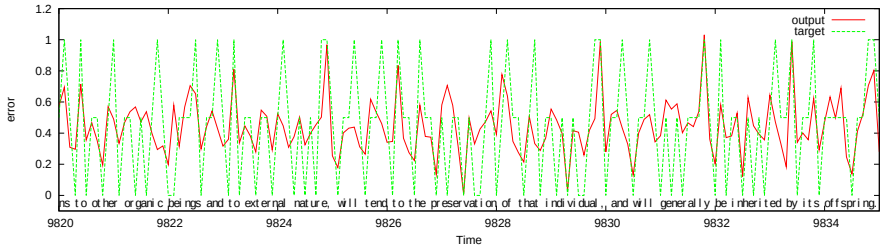


Figure 7.33: The Epoch error when predicting future inputs for an encoded string of text.

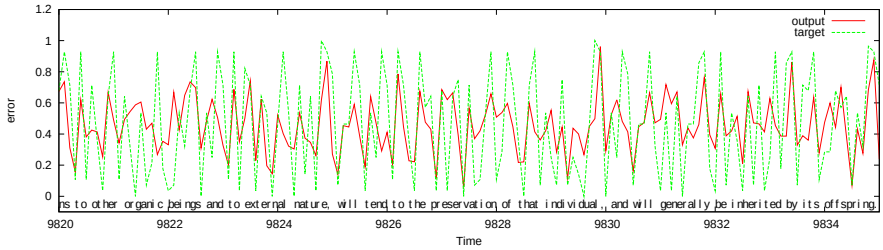
In Figure 7.34(b) the results are shown when the network has to predict the group of the next input value with a retention of 10. The retention was defined in Section 7.4.11 as the number of consecutive inputs after which all information about previous inputs in the reservoir is erased. It can be seen in Figure 7.33 that the error is slightly lower when the retention is being set to 10. In figure 7.34(c) the results an be seen when the network has been trained to predict the next input. Again, The output does go into the right direction of the target output but the range of the network output is much smaller. We believe that increasing the network capacity and a higher number of iterations will improve the results.



(a) Predict the group the next input belongs to.



(b) Predict the group the next input belongs to with a small retention.



(c) Predict the next input

Figure 7.34: The output of the readout is shown together with the target output for an encoded string of text.

Chapter 8

Conclusions and future research

Neural networks constitute a sub-area of artificial intelligence modeled after the human brain. Neural networks are an excellent way of tackling problems which are difficult to explicitly express by rules. Examples of these are dealing with uncertainty in data, image and text processing, pattern recognition and speech. A neural network comprises of a collection of neurons which can be viewed as very simple processing units. The behavior of a single neuron is simple, a collection of neurons can lead to very complex systems which can represent information. Neural networks have input neurons which translate an input value into some sort of neuronal code which serves as input for the subsequent layers of neurons the input neurons are connected to. The connections between neurons have a weight associated with it. These weights are crucial to the functioning of the network. The activity of the layers of neurons is propagated through the network all the way to the output neurons which encode a target value. Training the network is necessary to produce the correct output for a given input. Training involves adjusting the weights in such a way that the error of the output is reduced. It may take many iterations to correctly learn the training data set. Many methods exist for training a neural network based on different principles such as self-organizing maps and supervised learning. One of the most well-known techniques for training neural networks is back propagation.

In [21] a new computing paradigm was presented based on a neural structure of randomly connected spiking neurons. Independently echo state networks [19] were invented which are also based on a randomly connected structure, but consisting of sigmoidal neurons. Characteristically both neural paradigms rely on a highly dynamical reservoir from which readout neurons can tap into, in order to translate the excited medium to an output value. An example of the neural structure can be found in Figure 8.1, which depicts schematic overview. It can be seen that the sensor neurons, which are de-

picted in green connect to different parts of the reservoir of neurons which are depicted in blue. The dark blue neurons are used to illustrate that recurrent loops occur within the neural structure. The red neurons portrait a context layer and are the way to implicitly incorporate a time dimension within the liquid structure. Traditionally output values of neurons at time t are copied to the context layer and then used as input at time $t + 1$. Instead, we used longer delays and therefore the context layer is illustrative. The liquid state is decoded by the readout neurons which can be seen in yellow, and produce the output value. This type of neural network has come to be known as reservoir computing. Both paradigms have a way to store temporal information within the population of neurons. In [21] dynamical synapses store information about past inputs and in [19] recurrent connections from the readout structure provide temporal information. The latter method makes for recurrent connections in the classical sense. Both structures are well suited for time-series prediction. In both neural systems the weights of the reservoirs are not trained. Only the weights to the readout neurons are adapted to correct the output values. Several learning methods have been applied since then to adjust the readout weights. Both offline constructive methods, such as a system of differential equations or any linear regression method, as well as online adaptive methods, for example a gradient descent method, have been successfully applied. A variety of evolutionary algorithms have been applied for training these networks as well, for example CMA-ES or Evolino. The most powerful part of the network, namely the randomly connected reservoir is being left unchanged during training.

In order to improve on this concept we have introduced an online, input driven training method where weights of the randomly connected structure of neurons are adapted to meet several target constraints. The learning rules we present consist of several unrelated parts with each a constraint as a target. First of all we aim for a supersparse code for computational efficiency and, very importantly, scalable storage provided by a population of neurons. Contrary to [21] and [19] we do not create a population of neurons with intrinsic dynamics, rather we create a method which creates a highly scalable storage medium within a population of neurons. Per input signal, only a fraction of the neurons within the reservoir are allowed to be active. Secondly, a simple locally defined gradient descent method is applied both to the reservoir of neurons as well as the presynaptic weights of the readout neurons to reduce the output error. Thirdly, in order to maximize the efficiency of the network, we introduce a mechanism to determine which connections will be most efficient to adjust. This speeds up the learning process by reducing crosstalk between patterns. Finally, to combat the phenomenon of the network getting stuck in local minima, and in an attempt to create just the right amount of

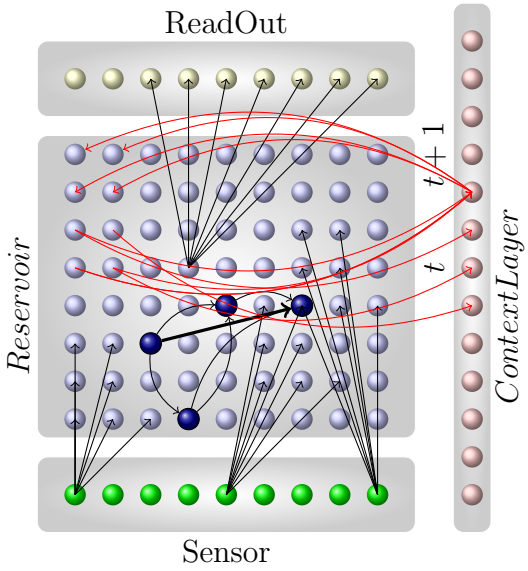


Figure 8.1: Schematic overview of a reservoir based neural network

specialization of neurons, we increase the orthogonality of the active weight vectors in a random way as long as the readout error is not within the tolerable margin. To provide the network with temporal information, we took the state of the column of neurons and presented this as an additional input.

We have shown how a supersparse code of a population of neurons can act as a scalable storage medium. The reason for its tremendous scalability is the result of a decrease in crosstalk between the input patterns and a decrease in computational load. Furthermore we successfully applied a set of learning rules to train a randomly connected structure of neurons. Not only are we able to control the neuronal activity, we have succeeded to adjust the firing times in order to reduce the output error. Although we have been able to get some positive results out of the recurrent structure with the goal of time-series prediction, we expected a better performance. We think the combination of spiking neurons and a sparse structure compromises the ability of the structure of neurons to act as a fading memory. We found a need for neurons in the network to be able to select which input information is necessary. Due to the spiking nature of the neuronal model we used, some input information remains unused. Further research is encouraged to improve the performance of a randomly connected network of neurons which is sparsely active, with respect to time-series prediction.

Bibliography

[1] Adrian, E., *The Impulses Produced by Sensory Nerve Endings*, J. Physiol. (Lond.) 61 (1926) 49–72

[2] Auer, P., Burgesteiner, H., Maass, W., *The p-Delta Learning Rule for Parallel Perceptrons*, `www.igi.tugraz.at/maass/p_delta_learning.pdf` (2002)

[3] Auer, P., Burgesteiner, H., Maass, W., *Reducing Communication for Distributed Learning in Neural Networks*, ICANN (2002) 123–128

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

[5] Bishop, C. M., *Pattern Recognition and Machine Learning*, Springer, 2006

[6] Bohte, S., *Spiking Neural Networks*, Ph.D. Thesis, Leiden University 2003

[7] Bohte, S., Kok, J., Poutré, H. la, *Error-backpropagation in Temporally Encoded Networks of Spiking Neurons*, Neurocomputing 48 (2002) 17–37

[8] Bohte, S., *The Evidence for Neural Information Processing with Precise Spike-times: A Survey*, Natural Computing 3 (2004) 195–206.

[9] Bohte, S., Mozer, M. C., *Reducing the Variability of Neural Responses: A Computational Theory of Spike-Timing-Dependent Plasticity*, Neural Computation 19(2) (2007) 371–403

[10] Bodén, M., *A Guide To Recurrent Neural Networks and Backpropagation*, (2001)

[11] Deneve, S., Latham, P., Pouget, A., *Reading Population Codes: A Neural Implementation of Ideal Observers*, Nature Neuroscience 2 (1999) 740-745

[12] Elman, J., *Finding Structure in Time*, Cognitive Science 14(2) (1990) 179–211

[13] Georgopoulos, A., Kalaska, J., Caminiti, R., Massey, J. T., *On the Relations Between the Direction of Two-dimensional Arm Movements and Cell Discharge in Primate Motor Cortex*, J. Neuroscience 2 (1982) 1527-1537

[14] Gerstner, W., *Time structure of the Activity in Neural Network Models*, Phys. Rev.E 51 (1995) 738–758

[15] Gerstner, W., Kistler, W., *Spiking Neuron Models*, Cambridge University Press, 2002

[16] Gupta, A., Wang, Y., Markram, H., *Organizing Principles for a Diversity of GABAergic Interneurons and Synapses in the Neocortex*, Science 287 (2000) 273–278

[17] Hertz, J., Krogh, A., Palmer, R. G., *Introduction to the Theory of Neural Computation*, Addison-Wesley, 1991

[18] Hochreiter, S., Schmidhüber, J., Maass, W., *Long Short-Term Memory*, Neural Computation 9(8) (1997) 1735–1780

[19] Jaeger, H., *The Echo State Approach to Analyzing and Training Recurrent Neural Networks*, (Tech. Rep. No. 148). Bremen: German National Research Center for Information Technology. (2001).

[20] Kandel, E. C., Schwartz, J. H., *Principles of Neural Science*, Elsevier, New York, 3rd edition, 1991

[21] Maass, W., Natschläger, T., *Real-Time Computing Without Stable States: A New Framework for Neural Computation Based on Perturbations*, Neural Computation 14(11) (2002) 2531–2560

[22] Maass, W., *Fast Sigmoidal Networks via Spiking Neurons*, Neural Computation 9(2) (1997) 279–304

[23] Maass, W., Markram, H., *Synapses as Dynamic Memory Buffers*, Neural Networks 15 (2002) 55–161

[24] Maass, W., *Noisy Spiking Neurons with Temporal Coding have more Computational Power than Sigmoidal Neurons*, NIPS 9 (1996) 211–217

[25] Markram, H., Wang, Y., and Tsodyks, M., *Differential Signaling via the Same Axon of Neocortical Pyramidal Neurons*, Proc. Natl. Acad. Sci. 95 (1998) 5323–5328

[26] Natschläger, T., Maas, W., *Finding the Key to a Synapse*, Advances in Neural Information Processing Systems 13 (2001) 138–144

[27] Olshausen, B., Field, D., *Sparse Coding of Sensory Inputs*, Current Opinion in Neurobiology 14 (2004) 481–487

[28] Pouget, A., Dayan, P., Zemel, R., *Information Processing with Population Codes*, Nat Rev Neuroscience 1(2) (2000) 125–32

[29] Rieke, F., Warland, D., Ruyter van Stevenink, R. de, Bialek, W., *Spikes-Exploring The Neural Code*, MIT Press, 1997

[30] Rolls, E. T., Tovee, M. J., *Sparseness of the Neuronal Representation of Stimuli in the Primate Temporal Visual Cortex*, J Neurophysiol 73 (1995) 713–726

[31] Rossum, M. C. W., Renart, A., Nelson, S. B., Wang, X.-J., Turrigiano, G. G., *Reading Out Population Codes With a Matched Filter*, `http://www.era.lib.ed.ac.uk/bitstream/1842/234/1/pc_nips.pdf` (2001) [Accessed 2009]

[32] Rossum, M. C. W., Bi, G.-Q., Turrigiano, G. G., *Stable Hebbian Learning from Spike Timing-Dependent Plasticity*, J Neurosci. 20 (2000) 8812–21

[33] Schrauwen, B., Campenhout, J. van., *Extending Spikeprop*, Proceedings of the International Joint Conference on Neural Networks 1 (2004) 471–476

[34] Schrauwen, B., *Pulstreincodering en Training met Meerdere Datasets op de CBM*, M.Sc. Thesis., Universiteit Gent, 2002 [Dutch]

[35] Sordo, M., *Introduction To Neural Networks in Healthcare*, Open Clinical, `http://www.openclinical.org/docs/int/neuralnetworks011.pdf` 2002 [Accessed 2009]

[36] Veelen, M. van, Nijhuis, J. A. G., Spaanenburg, L., *Neural Network Approaches to Capture Temporal Information*, Computing Proceedings of the CASYS AIP 517 (1999) 361–371

[37] Vinje, W. E., Gallant, J. L., *Natural Stimulation of the Nonclassical Receptive Field Increases Information Transmission Efficiency in V1*, J. Neuroscience 22 (2002) 2904–2915

[38] Vinje, W. E., Gallant, J. L., *Sparse Coding and Decorrelation in Primary Visual Cortex During Natural Vision*, Science 287 (2000) 1273–1276

[39] Vreeken, J., *On Real-world Temporal Pattern Recognition Using Liquid State Machines*, M.Sc. Thesis, University Utrecht, 2004

[40] Weliky, M., Fiser, J., Hunt, R. H., Wagner, D. N., *Coding of Natural Scenes in Primary Visual Cortex*, Neuron 37 (2003) 703-718

[41] Williams, R. J., Zipser, D., *A Learning Algorithm for Continually Running Fully Recurrent Neural Networks*, Neural Computation 1(2) (1989) 270–280

[42] Willmore, B., Tolhurst, D. J., *Characterizing the Sparseness of Neural-Codes*, Network 12 (2001) 255–270

[43] Zemel, R., Dayan, P., Pouget, A., *Probabilistic Interpretation of Population Codes*, Neural Computation 10 (1998) 403–430

[44] Zhang, K., Sejnowski, T., *Neuronal Tuning: To Sharpen or to Broaden?*, Neural Computation 11(1) (1999) 74–85