A Comparison of Simple Recurrent and Sequential Cascaded Networks for Formal Language Recognition

Henrik Jacobsson

Submitted by Henrik Jacobsson to the University of Skövde as a dissertation towards the degree of M.Sc. by examination and dissertation in the Department of Computer Science.

November 12, 1999

I hereby certify that all material in this dissertation which is not my own work has been identified and that no material is included for which a degree has already been conferred upon me.

____

Henrik Jacobsson

Abstract

Two classes of recurrent neural network models are compared in this report, simple recurrent networks (SRNs) and sequential cascaded networks (SCNs) which are first- and second-order networks respectively. The comparison is aimed at describing and analysing the behaviour of the networks such that the differences between them become clear. A theoretical analysis, using techniques from dynamic systems theory (DST), shows that the second-order network has more possibilities in terms of dynamical behaviours than the first-order network. It also revealed that the second order network could interpret its context with an input-dependent function in the output nodes. The experiments were based on training with backpropagation (BP) and an evolutionary algorithm (EA) on the A^nB^n-grammar which requires the ability to count. This analysis revealed some differences between the two training-regimes tested and also between the performance of the two types of networks. The EA was found to be far more reliable than BP in this domain. Another important finding from the experiments was that although the SCN had more possibilities than the SRN in how it could solve the problem, these were not exploited in the domain tested in this project.

Contents

List of Figures **v**

List of Tables **viii**

1 Introduction **1**

2 Background and project description **3**

2.1 Previous work on RNN comparison 3

2.2 Problem domain 4

2.3 Previous work on the A^nB^n-language 6

2.4 Training algorithms 8

2.5 Project statement 9

3 Network analysis **11**

3.1 Representation of the problem 11

3.2 Architectures 12

3.3 Identification of the nodes and weights 13

3.4 Hyperplane analysis 14

3.4.1 SRN 14

3.4.2 SCN 16
3.5 Taxonomy of networks 21
3.5.1 SRN 21
3.5.2 SCN 25
3.6 Summary 33

4 Experiments **34**
4.1 Quantitative measures 34
4.2 The evaluation of correctness 35
4.3 Details of the simulations 37
4.3.1 BP 37
4.3.2 EA 38

5 Results **41**
5.1 Reliability 41
5.1.1 BP 42
5.1.2 EA 44
5.1.3 EA vs. BP 45
5.2 Quality of successful networks 49
5.2.1 BP 49
5.2.2 EA 50
5.2.3 EA vs. BP 52
5.3 Efficiency 57
5.3.1 BP 58
5.3.2 EA 59

5.3.3 EA vs. BP 59
5.4 Consistency 64
5.4.1 BP 65
5.4.2 EA 66
5.4.3 EA vs. BP 67
5.5 Generalization 77
5.5.1 BP 77
5.5.2 EA 78
5.5.3 EA vs. BP 78
5.6 The distribution of signatures 82
5.6.1 BP 83
5.6.2 EA 86

6 Conclusions **93**
6.1 Future work 95

Acknowledgments **97**

Bibliography **98**

Appendices **103**

A Artificial neural networks **103**
A.1 Feed forward networks 104
A.2 Recurrent neural networks 107
A.2.1 First-order recurrent networks 108
A.2.2 Second-order recurrent networks 110

B Backpropagation **114**

B.1 BP training of SRN 114

B.2 BP training of SCN – The "backspace trick" 117

C Evolutionary algorithms **120**

D Dynamic systems theory **126**

D.1 Attractors 128

D.1.1 Fixed point attractors 128

D.1.2 Periodic attractors 129

D.1.3 Strange attractors 130

D.1.4 Basin of attraction 131

List of Figures

1 Some strings of the A^nB^n-language 5

2 A part of the error surface of a self-recurrent node 7

3 The SRN architecture for predicting the A^nB^n-language 13

4 The SCN architecture for predicting the A^nB^n-language 14

5 The dynamics of the threshold values of the function network 18

6 An example of an internal activation space of an SCN where two hyperplanes are visible 20

7 The internal behaviour of different SRNs 24

8 An example of the internal behaviour of an SCN with signature $(2, 0, 2, 0, 0)$ 28

9 An example of the internal behaviour of an SCN with signature $(2, 0, 1, 1, 1)$ 29

10 An example of the internal behaviour of an SCN with signature $(1, 1, 2, 0, 1)$ 29

11 An example of the internal behaviour of an SCN with signature $(1, 1, 1, 1, 0)$ 30

12 An example of the internal behaviour of an SCN with signature $(1, 1, 1, 1, 1)$ 30

13 An example of the internal behaviour of an SCN with signature $(1, 1, 0, 2, 1)$ 31

14 An example of the internal behaviour of an SCN with signature $(0, 2, 1, 1, 1)$ 31

15 An example of the internal behaviour of an SCN with signature $(0, 2, 0, 2, 0)$ 32

16 The generalization capacity of the successful SRNs trained by BP . . 79

17 The maximum generalization capacity of the successful SRNs trained by BP . . . 79
18 The generalization capacity of the successful SRNs trained by EA . . 80
19 The maximum generalization capacity of the successful SRNs trained by EA . . . 80
20 The generalization capacity of the successful SCNs trained by EA . . 81
21 The maximum generalization capacity of the successful SCNs trained by EA . . . 81
22 The generalization capacity of SRNs, trained by BP, with different signatures . . . 85
23 The distribution of different SRN-signatures in the EA-population . . 89
24 The generalization capacity of SRNs, trained by the EA, with different signatures . . . 90
25 The distribution of different SCN-signatures in the EA-population . . 91
26 The generalization capacity of SCNs, trained by the EA, with different signatures . . . 92
27 A formal description of a feed-forward network . . . 106
28 The logistic function . . . 107
29 A simple recurrent network (SRN) . . . 109
30 Backpropagation through time (BPTT) . . . 109
31 A second order non-recurrent network . . . 112
32 A sequential cascaded network (SCN) . . . 113
33 The principle of gradient descent searching . . . 115
34 The principle of the backspace trick . . . 117

35 The principles of an evolutionary algorithm 122

36 An example of an evaluation function 122

37 The Gaussian distribution 125

38 Different dynamic behaviours of $f_a = ax(1-x)$ 127

39 Bifurcation diagram of $f_a = ax(1-x)$ 132

List of Tables

1	Representations of A and B	12
2	The signatures of the SRN	23
3	The signatures of the SCN	27
4	The distribution of lengths in the training set for BP	37
5	Reliability of BP when the two-bit representation was used	46
6	Reliability of BP when the one-bit representation was used	47
7	Reliability of EA	48
8	Quality of BP when the two-bit representation was used	53
9	Quality of BP when the one-bit representation was used	54
10	Quality of the successful networks trained by EA	55
11	The relative quality of the EA	56
12	Efficiency of BP when the two-bit representation was used	61
13	Efficiency of BP when the one-bit representation was used	62
14	Efficiency of EA	63
15	Consistency of BP when the two-bit representation was used	68
16	Consistency of BP when the one-bit representation was used	69
17	Consistency of EA	70

18 Average preservation time of BP when the two-bit representation was used 71

19 Average preservation time of BP when the one-bit representation was used 72

20 Average preservation time of EA 73

21 The number of rediscoveries of solutions for each length, when the two-bit representation was used 74

22 The number of rediscoveries of solutions for each length, when the one-bit representation was used 75

23 The number of rediscoveries of solutions for each length for the EA . 76

24 The distribution of signatures of the SRNs trained by BP 84

25 The distribution of signatures of the SCN in the final networks from BP 85

26 The distribution of signatures of the SRN in the resulting populations of the EAs 88

27 The distribution of signatures of the SCN in the resulting populations of the EA 88

28 Different attractors 128

29 A part of the orbit of $f_4 = 4x(1-x)$ 131

Chapter 1

Introduction

First- and second-order recurrent networks are used in various domains and fields of cognitive and computer sciences. At a first glance, they are fundamentally different from each other and one could expect that the second-order networks would be able to do much more than the first-order networks. Which one of the network types that is used is often just a matter of habit or convenience. It has been shown that, if the general classes of first- and second-order networks are considered, they are, in fact, computationally equivalent [SS94, Sie99] and therefore could potentially compute the same things. Hence, those that prefer to use second-order networks could in principle use first-order instead and vice versa. However, the theoretical proof of equivalence does not show the practical differences as whether the network can *learn* to compute the same things or if they will solve the same problems in similar ways. The comparison in this dissertation is focused on two specific types of first- and second-order networks, a Simple Recurrent Network [Elm90] (SRN) and a second-order network called Sequential Cascaded Network [Pol86] (SCN) (for details of the network architectures see appendix A).

This dissertation aims at analysing and testing the SRN and the SCN in such way that the differences between them become apparent. The analysis is conducted by using tools from Dynamic Systems Theory (DST) (see appendix D). The experimental comparison is conducted on a formal language domain, the A^nB^n-grammar, which requires that the networks are able to "count" the number of As in order to be able to correctly predict the number of Bs . The networks are trained using two fundamentally different learning techniques: a gradient descent search algorithm called backpropagation (BP) (described in appendix B) and an Evolutionary Algorithm (EA) (described in appendix C).

The comparison is based on specific simple instances of the SRNs and SCNs which are chosen such that they share some common features. The feature they share is the how many context nodes they have and the fact that they are among the smallest possible networks of their type. The number of context nodes defines the dimensionality of the state space that is the only type of "short-term memory" the networks have access to.

The report is arranged as follows. In chapter 2 the project is defined in detail and the problem domain is identified. Chapter 3 describes the methods that are used to analyse the behaviours of the networks and a description of the theoretical differences between the networks. The details of the experiments are described in chapter 4 and the results of these experiments are described in chapter 5. In chapter 6 the results are discussed and conclusions are drawn from both the analysis in chapter 3 and the results of chapter 5. Directions for future work are given in the end of chapter 6.

Chapter 2

Background and project description

The high-level goal of this project is to map the differences between first- and second-order recurrent neural networks. The comparison is limited to simple instances of SRNs and SCNs which are described in appendix A.

This chapter describes some of the previous work on first- and second-order recurrent network comparison. The problem domain, which will be used in this dissertation, and related work on this domain is also presented. Finally, the project is defined in the last section.

2.1 Previous work on RNN comparison

Previous works which compare first- and second-order recurrent networks are either theoretical [SS94, GGCC94, Sie99] or based on experiments [MG93, HG95]. The theoretical work has shown that the two types of networks are in general computationally

equivalent [SS94, Sie99] and, hence, can compute the same functions. This was not verified by the results of the experiments in [MG93] which showed that their second order networks could solve problems for which no first-order networks were found. Also [HG95] which tested a number of recurrent architectures without analysing their behaviour in detail, did show that, for some problems, the second-order recurrent network outperformed the first-order. [MG93] employed *state machines* in their analysis of the behaviours of the networks which did not lead to much more than the question: "Why do second order networks outperform the first order variety?". This is in part explained in [GGCC94] where it was shown that, if the networks are limited to one layer of weights, the second-order network can compute more functions than the first-order network.

One question that remains unanswered, however, is how the training of the networks affects how and how well they solve a problem. One approach to solve this is taken in [TBW99] where the behaviour of SRNs is analysed using Dynamic Systems Theory (DST, see appendix D). Their results shed light on what types of dynamic solutions were employed by SRNs.

2.2 Problem domain

There are many formal language problems that can be used for training and testing RNNs. In this report a language generated by the A^nB^n-grammar will be used (see figure 1). The motivation for basing the quantitative analysis on this problem is that it is a well-defined language which is easy to analyse. Many previous papers have also focused on this problem using SRNs, e.g. [WE95, RWE99, BWTB99, TBW99]. The motivation for using the A^nB^n-grammar was, in [WE95], that it is

the simplest possible grammar requiring a counter or a push down automata. The language requires a representation of the syntactic structure in order to be recognized.

AB
AABB
AAABBB
AAAABBBB
AAAAABBBBB
⋮

Figure 1: Example of strings that are possible in the A^nB^n-language. The rule is simple: a number of As followed by an equal number of Bs.

The task for the networks will be to continually predict the next symbol in the sequence of symbols built up of the strings from the grammar. A set of strings from the grammar will be merged together, such that after the last B of a string, the A of the succeeding string should be predicted. In many other works, e.g. in [Pol91] and [Pol90], language acquisition is treated as a *classification task*. Then the network is to separate between strings belonging to the language from a set of counter-examples, i.e non-members. The role of the "teacher" is then quite explicit since the performance of the network depends heavily on which counter-examples are included in the training set.

However, all previous work on the A^nB^n-language, which has been included as background material in this dissertation, defined the task as a *prediction task*. In order to make correct predictions the network must, just as for the classification task, have some kind of representation of the grammar. One early approach to use prediction as the basis for temporal tasks for RNNs was made in [Elm90] where an SRN solved a temporal XOR problem and a few natural language problems. In

those specific problems, without resorting to any details, only a few symbols were predictable, but the network was still able to figure out the underlying structure of the grammar. The strings of the A^nB^n-language are also partly unpredictable since the first B of the strings is unpredictable when the length of the string is unknown beforehand.

2.3 Previous work on the A^nB^n-language

The first paper (to the authors knowledge) that employed recurrent networks (see appendix A) for the prediction of the A^nB^n-grammar is [WE95]. They used back-propagation through time (BPTT, see appendix B) to train a simple recurrent network and found that the network solved the task in an unexpected way. The network which they analysed manipulated its internal representation by oscillation, which was not the behaviour they expected. This is best understood if examples of this, and other behaviours are studied, see figure 7 on page 24 for a few examples of monotonic and oscillating behaviours found when recurrent networks solved the A^nB^n-grammar.

In [RWE99] the dynamics of the SRNs were analysed with tools from dynamic systems theory (see appendix D). They found that the network actually could predict the A^nB^n-grammar by manipulating its internal representation monotonically. However, they found that the oscillating solutions were far better on generalizing to longer strings. They also concluded that dynamical systems theory was a more reliable approach for analysing RNNs than discrete methods, although the theoretical framework of discrete automata is well-developed and describes a hierarchy of languages. The use of dynamical systems theory allows to take into account the continuous states that RNNs employ [Kol94], this is not possible using discrete analysis,

e.g. finite state machines.

In [BWTB99] the problem was investigated further from the view of the training algorithm. The error-landscape which BPTT (a training algorithm, partly described in appendix A) searches through was investigated and it was found that it was of a chaotic nature with many high peaks, a difficult environment for training with gradient search methods such as BP and BPTT. Figure 2 illustrates a piece of the error-landscape, showing the error-gradient as a function of a bias and self-weight of a context node. These peaks represent very high derivatives which causes BPTT to take large leaps in the search space.

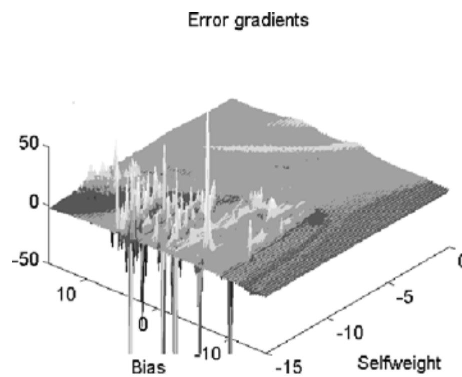

Figure 2: This diagram shows the error gradients as a function of the bias and self-weight of a context node in the SRN when the A^nB^n-grammar is considered. The high peaks that are found are problematic for BPTT. The diagram is based on the results in [BWTB99].

An evolutionary algorithm was used to train the networks in [TBW99]. It was found that the evolutionary method was able to find solutions which were not found with BP or BPTT. In order to analyse which types of solutions were found, they introduced a classification method which tagged the networks with “signatures”. Each

signature is represented by a type of dynamical behaviour in the network. The signatures are described in section 3.5. After the most successful types of networks were identified using their signatures, they also tested to train the BPTT under biased conditions to guide the training towards these signatures. The results were promising in some cases. The signatures, introduced in [TBW99], are important since they allow automatic classification of the different types of solutions which a network can employ. It reveals that there are not only two classes of behaviours, monotonic and oscillating, but at least one more which they called "exotic".

2.4 Training algorithms

The networks, described in the last section of this chapter, are trained to solve the problem. This training can be conducted using different training algorithms with different properties. The BP-algorithm (explained in appendix B) is quite dependent on error gradients which correctly directs the search towards a solution but when the BPTT-algorithm is used on, e.g., the A^nB^n-language, the gradients contains many features that misguides the search [BWTB99]. An evolutionary algorithm (EA), as described in appendix C, is not dependent on the gradients. This is emphasized by Goldberg [Gol89, p. 2].

> These algorithms are computationally simple yet powerful in their search for improvement. Furthermore, they are not fundamentally limited by restrictive assumptions about the search space (assumption concerning continuity, existence of derivatives, unimodality, and other matters).

And it is precisely this lack of "restrictive assumptions" that we seek to get a fair comparison of the two networks. This higher "freedom" from assumptions can be argued to lead more comparable results than for the gradient search methods since the assumption of error gradients underlying BP may affect the results differently for the both network types. The fitness function of the EA does not need to be derivable, as the error function must be when using BP (see appendix B for details). The EA only needs a measure of how well every individual candidate solution solves the problem at hand. SSE is an approximative measure of how well the A^nB^n-grammar is predicted by the network, while the fitness function may be more exact based solely on the prediction ability, see section 4.3.2 for a description of the fitness function chosen.

An EA might also find more "interesting" solutions since it does not, as easily as gradient methods, get stuck on local maxima [Gol89]. This is exemplified in the work of Meeden in [Mee96] where an EA found solutions, to a robot control task, which a gradient method did not ever find.

2.5 Project statement

Since the comparison previously has been conducted from either a theoretical or a pragmatical point of view, this project aims at combining these two types of analysis. The theoretical analysis will be based on DST and will, as in [GGCC94], be aimed at specific restricted instances of SRNs and SCNs. The result of this analysis will then be used to analyse the experimental part of the project which is to let the SCN and SRN be trained on a well-known and, within the domain of recurrent networks, commonly used formal language domain. There are several aspects that influence the analysis of the architectures, such as the method of training. Therefore the networks

have been trained using both BP and EA and separate analysis has been done for these both training-regimes. BP is described in appendix B and the principle behind the EA in appendix C.

Chapter 3

Network analysis

This chapter explains the network architectures that are used as a basis for the experiments and analyses their theoretical possibilities in terms of dynamic behaviours. A classification scheme of their different behaviours is also defined.

3.1 Representation of the problem

Two different representations of the symbols in the A^nB^n-language will be used, a *one-bit* representation and a *two-bit* representation, which are are shown in table 1. The shorter representation is more efficient in the implementation of the networks while the longer provides redundant information, which may be useful for the gradient descent search.

Symbol	1-bit	2-bit
A	0	01
B	1	10

Table 1: The symbols of the A^nB^n-language are handled with two different representations, the *one-bit-* and the *two-bit-representation*.

3.2 Architectures

The architectures are chosen to be the simplest possible instances of first- and second-order networks (see appendixA). Both networks were chosen such that they have one input and one output layer and two context nodes[1]. These limitations affect the whole theoretical discussion but do also allow us to find the differences between simple instances of first- and second-order networks. If general first- and second-order networks would be considered, without any restrictions, the result would only be same as in [Sie99], i.e. that they are computationally equivalent. Moreover, implementation of the systems would be impossible without any restrictions.

In previous work on the A^nB^n-language an SRN as shown in figure 3 was used. The choice of the SRN architecture here was therefore quite straightforward, namely the same. The use of two context units allows the activation to be plotted into a graph which simplifies the analysis. A variant with only one input and one output node is used when the one-bit representation of the language is used.

The architecture of the SCN should be as comparable to the SRN as possible. In [HG95], which compared a wide set of different recurrent networks, the networks were compared by either holding the number of weights or context units constant. It was

[1] This is not the "standard" term used by Elman who addresses the context nodes as hidden nodes and the nodes which contain the activation of the hidden nodes in the last time step as context nodes.

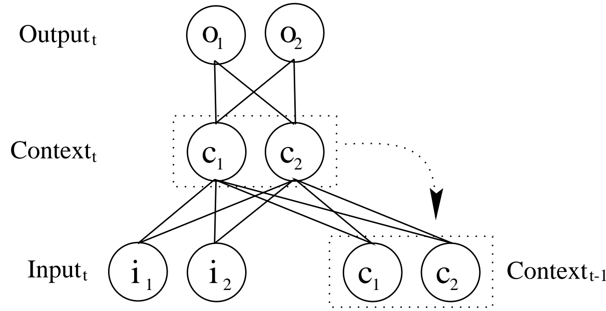

Figure 3: The SRN architecture for predicting the A^nB^n-language. A slightly different variant of this architecture was also used in for the one-bit representation, it had only one input node and one output node instead of two. The biases of the nodes are not shown in this figure.

concluded that which one of these was held constant did not significantly change the relative comparison of the networks' performance. Since the analysis of the networks is based on their internal representation it was decided to keep the number of context units constant in this project. Therefore the number of context units was set to two, just as for the SRN. The architecture is shown in figure 4. As for the SRN, this architecture was used in a slight variant with only one input- and one output-node when the one-bit representation of the language was used.

3.3 Identification of the nodes and weights

To simplify the discussions that follows, an identification system of the weights will be introduced. The weights of SRN architecture will get their own individual identifiers. The identifiers are built up by the frame $W_{from\ to}$, e.g. the weight from the bias to the first output node will be called W_{bo_1} and the weight from node c_1 to itself $W_{c_1c_1}$. When the single bit representation of the symbols is used the output node is simply

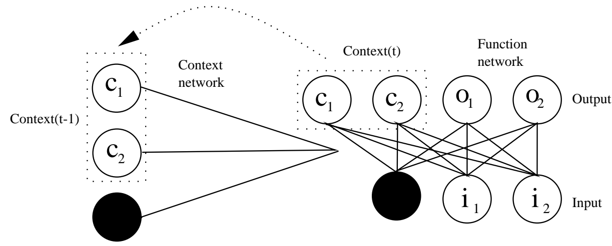

Figure 4: The SCN architecture chosen for predicting the A^nB^n-language. A slightly different variant of this architecture were also used in the experiments, it had only one input node and one output node instead of two. The biases are explicitly shown as black nodes in this figure to clarify that there are both first and second order biases in the architecture.

referred to as o and the input as i.

The same system for identifying weights is used for the second order network. The dynamical weights will be named exactly as in the SRN. The second order weights are named on the form $W_{context-nodeW_{from\ to}}$ e.g. the second order weight connecting the first context node with the dynamic weight between the first input node and the second output node will be called $W_{c_1W_{i_1o_2}}$.

3.4 Hyperplane analysis

3.4.1 SRN

The activation of the hidden-, or context-nodes in the SRN can be plotted in a graph in order to analyse how the dynamics of the internal representation relates to the learned task. Similar analysis of the internal behaviour has been done in the previous work on using SRNs on the A^nB^n and is a frequently used method for analysing ANNs.

A *decision hyperplane* is a "border" in the state-space of the hidden nodes which separates one output class from another. Each output node implements its own hyperplane that separates the state-space of the preceding layer into two regions which are interpreted as two different classes by the node. More on hyperplane analysis can be found in [HB90], [MP69] and [SJ90]. Analysis methods of the internal representation in general are described in [Bul97].

If we consider the one-bit representation of the A^nB^n-language, the activation of the single output node is determined by[2]

$$o(t) = \Gamma(c_1(t)W_{c_1o} + c_2(t)W_{c_2o} + W_{bo}) \tag{1}$$

The output is rounded off to nearest integer value and, hence, the threshold of the interpretation of the activation in o is 0.5. An activation of 0.5 is equivalent with a *net*-value of 0 on the basis of the definition of Γ, see equation 22. The *net*-value of o, net_o is determined by

$$net_o(t) = c_1(t)W_{c_1o} + c_2(t)W_{c_2o} + W_{bo} \tag{2}$$

$net_o = 0$ represents a "border" which separate the output classes of a node, e.g. it separates As from Bs in the A^nB^n-language.

$$\begin{aligned} net_o(t) &= 0 \\ c_1(t)W_{c_1o} + c_2(t)W_{c_2o} + W_{bo}t &= 0 \\ c_1(t) &= -\frac{c_2(t)W_{c_2o} + W_{bo}}{W_{c_1o}} \end{aligned} \tag{3}$$

[2]The nodes and weights are identified as described in section 3.3.

which can be plotted as a line in the two-dimensional state space. The solution of $net_o = 0$ is represented by a *hyperplane* which separates the activation space into two regions. In this case, when we have two hidden nodes, the hyperplane is a straight line in a two-dimensional state-space. Figure 7 on page 24 shows some examples of hyperplanes, plotted in the activation space of the context nodes in the SRN network chosen for the $A^n B^n$-grammar.

3.4.2 SCN

A hyperplane for the function network

If we consider the network for the one-bit representation, the function sub-network of the SCN has one input unit and one output unit. Since the subnetwork, studied in isolation, is nothing else than a feed-forward network it can be analysed as such. The output unit of the function network defines a decision hyperplane in the activation space of the single input unit just as the output nodes of the SRN defined a hyperplane in the activation space of the context unit's activation.

The activation of the output node in the function network is determined by

$$o(t) = \Gamma(W_{io}(t) + W_{bo}(t)) \tag{4}$$

That means that the net-value of the output unit, when not considering the context network or the time, is defined as

$$net_o(t) = iW_{io}(t) + W_{bo}(t) \tag{5}$$

If we solve the equation $net_o = 0$ as before to get the decision hyperplane, we get

$$\begin{aligned} net_o(t) &= 0 \\ i(t)W_{io}(t) + W_{bo}(t) &= 0 \\ i(t) &= -\frac{W_{bo}(t)}{W_{io}(t)} \end{aligned} \tag{6}$$

i.e. since the activation space of the input unit is one-dimensional, the decision hyperplane is a single point, determined by the last line of equation 6. Since the weights of the function network are dynamical (see equation 24) and changes every time step, the decision hyper plane changes also. Figure 5 shows the dynamical hyperplane of the function network as a function of time.

When the two-bit representation is used and we consider the first of the two output units, o_1, the equation which describes the hyperplane in the function network is

$$i_1(t) = -\frac{i_2(t)W_{i_2o_1}(t) + W_{bo_1}(t)}{W_{i_1o_1}(t)} \tag{7}$$

and likewise for o_2.

A hyperplane for the whole SCN

It is also possible to not only calculate the hyperplane of the function network in isolation, but also include the context network, which is continuously updating the weights of the function network. This is done for the SCN by solving the same equation as for the SRN and function network, $net_o = 0$. The result is a bit different, however, as we shall see. The hyperplane is not determined by the immediate weights leading to the output unit, since these are dynamic, but instead the second order

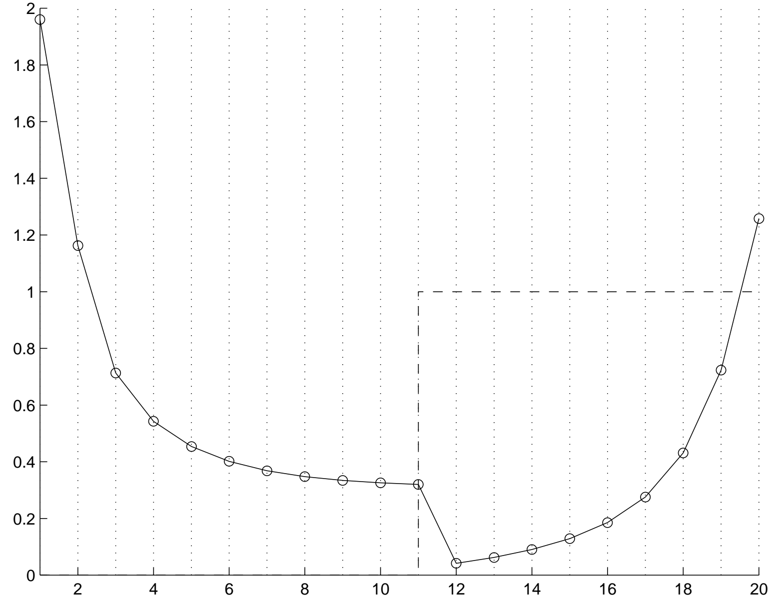

Figure 5: The dynamics of the threshold values of the output node in the function network. The dotted line is the input to the network ($i = 0$ for A and $i = 1$ for B) and the curve represents the dynamic hyperplane of the function network, according to equation 6, which determines which symbol will be predicted.

weights in combination with the activation of the input unit.

If the second order weights are written out in equation 4, we get a specific case of the general equation 25 on page 112

$$
\begin{aligned}
o(t) = \Gamma(c_1(t-1)(i(t)W_{c_1W_{io}} + W_{c_1W_{bo}}) + \\
c_2(t-1)(i(t)W_{c_2W_{io}} + W_{c_2W_{bo}}) + \\
i(t)W_{bW_{io}} + W_{bW_{bo}})
\end{aligned}
\tag{8}
$$

i.e. the net-value of the output node is

$$\begin{aligned} net_o(t) = {} & c_1(t-1)(i(t)W_{c_1W_{io}} + W_{c_1W_{bo}}) + \\ & c_2(t-1)(i(t)W_{c_2W_{io}} + W_{c_2W_{bo}}) + \\ & i(t)W_{bW_{io}} + W_{bW_{bo}} \end{aligned} \tag{9}$$

If we solve the equation $net_o(t) = 0$ as we did to generate the hyperplane for the SRN we get

$$\begin{aligned} net_o(t) &= 0 \\ 0 &= c_1(t-1)(i(t)W_{c_1W_{io}} + W_{c_1W_{bo}}) + \\ & \quad c_2(t-1)(i(t)W_{c_2W_{io}} + W_{c_2W_{bo}}) + \\ & \quad i(t)W_{bW_{io}} + W_{bW_{bo}} \\ c_1(t-1) &= -\frac{c_2(t-1)(i(t)W_{c_2W_{io}} + W_{c_2W_{bo}}) + i(t)W_{bW_{io}} + W_{bW_{bo}}}{(i(t)W_{c_1W_{io}} + W_{c_1W_{bo}})} \end{aligned} \tag{10}$$

That means that the hyperplane is determined as a linear function for the SCN as well as it was for the SRN. In this case the hyperplane defines $c_1(t-1)$ as a function of $i(t)$ and $c_2(t-1)$ (given constant weights in the context network). Since the input to the network in the domain of the A^nB^n-language only have two possible values we can define *two* hyperplanes, one when A is given as input to the network and one when B is. An example of these hyperplanes is shown in figure 6.

This principle works also when we consider the two-bit representation of the problem, the only thing that differs is that the equation is a bit more complex since we will have four input-output connections instead of one. The equation describing the

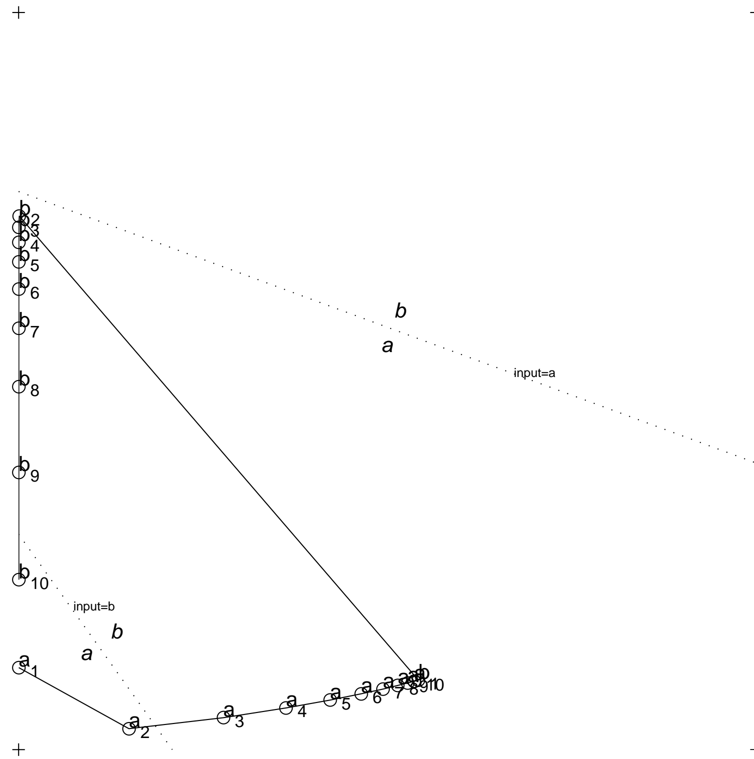

Figure 6: An example of an internal activation space where two hyperplanes are visible (the dotted lines). One hyperplane which is active when the input is an A and one when it is a B. The hyperplane for A-inputs is often outside the activation space since the network always predicts As after As. In this example, however, both hyperplanes are visible.

hyperplanes in the activation space of the context units when using the two-bit representation is

$$c_1(t-1) = -\frac{\begin{pmatrix} c_2(t-1)(i_1(t)W_{c_2 W_{i_1 o_1}} + i_2(t)W_{c_2 W_{i_2 o_1}} + W_{c_2 W_{bo_1}})+ \\ i_1(t)W_{bW_{i_1 o_1}} + i_2(t)W_{bW_{i_2 o_1}} + W_{bW_{bo_1}} \end{pmatrix}}{i_1(t)W_{c_1 W_{i_1 o_1}} + i_2(t)W_{c_1 W_{i_2 o_1}} + W_{c_1 W_{bo_1}}} \tag{11}$$

and likewise for o_2. That means that also for the SCN designed for the two-bit

representation there is one decision hyperplane for each possible input in every output node.

3.5 Taxonomy of networks

The behaviour of the trained networks varies between oscillating and monotonic solutions to the counting problem. A classification scheme of this behaviours is suggested in [TBW99] based on the *self-weights* of the recurrent nodes, i.e. the weights from the context nodes back to themselves. A negative self-weight in an SRN causes the activation of the node to oscillate. Instead of using the self-weight as the basis of the classification into signatures we will define the signatures using *self-derivatives*. This, more general definition, allows the classification to be transfered to the SCN in the next section.

Definition 1 A *self-derivative* of a time-dependent parameter x, is the derivative of x at time t given its previous value in time $t-1$, i.e. $\frac{\partial x(t)}{\partial x(t-1)}$.

3.5.1 SRN

If we consider the SRN handling the one-bit representation, the activation, at time t, of c_1 is determined by

$$c_1(t) = \Gamma(c_1(t-1)W_{c_1c_1} + c_2(t-1)W_{c_2c_1} + iW_{ic_1} + W_{bc_1}) \qquad (12)$$

The self-derivative of c_1, i.e. of the current activation in c_1 given its previous activation is then

$$\frac{\partial c_1(t)}{\partial c_1(t-1)} = W_{c_1c_1}\Gamma'(c_1(t-1)W_{c_1c_1} + c_2(t-1)W_{c_2c_1} + i(t)W_{ic_1} + W_{bc_1}) \qquad (13)$$

Since $0 < \Gamma < 1$ and $\Gamma'(x) = \Gamma(x)(1 - \Gamma(x))$ according to equation 23 on page 105, we have that $0 < \Gamma'(x) \leq 0.25$, i.e. $\Gamma'(x)$ is always positive and therefore the sign of $\frac{\partial c_1(t)}{\partial c_1(t-1)}$ is determined by $W_{c_1c_1}$ only. The same can also be shown for c_2. In other words, the sign of the self-derivative of a context node is determined by the sign of its self-weight, hence the definition subsumes the one used in [TBW99].

If the architecture for the two-bit representation is used, no change in the determination of the sign occurs since the number of context nodes is the same and, hence, the number of self-weights (and self-derivatives) is the same. From the definition 6 in appendix D we know that whether the state oscillates in one dimension or not is determined by the sign of the self-derivative of the transformation function of that dimension. Here we only consider the oscillation of one node at a time and the transformation function is based on the weights of the network.

The *signature* of the SRN networks was defined as (p, q) in [TBW99], where p and q are the number of positive and negative self-weights, respectively, i.e. determined by the signs of $W_{c_1c_1}$ and $W_{c_2c_2}$. There are three possible instances[3] of this signature which are described in table 2. Some examples of the typical behaviour of the different signatures are shown in figure 7.

[3]If making the reasonable assumption that there will not occur any zero-weights.

Signature	Term	Behaviour
$(2, 0)$	Monotonic	Monotonic
$(1, 1)$	Exotic	Either monotonic or oscillating
$(0, 2)$	Oscillating	Oscillating

Table 2: The three possible signatures of an SRN. The terms are taken from [TBW99].

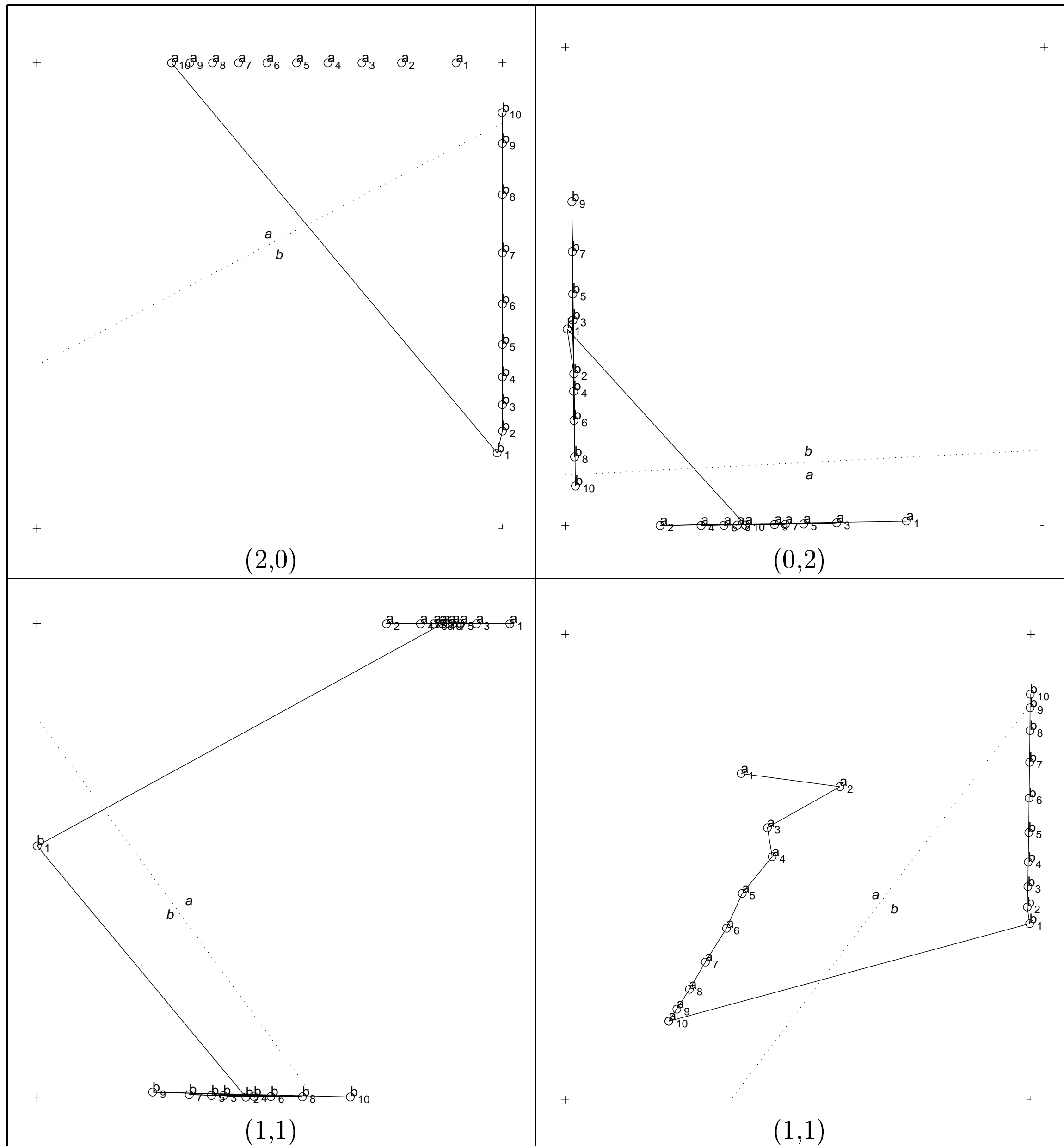

Figure 7: Typical examples of the internal behaviour in SRNs with different signatures. The horizontal and vertical axis represent the activation of the two context nodes c_1 and c_2 as in figure 3 which can have activations in the range [0,1]. The dotted line is the decision hyperplane determined by equation 3. Note that when the last B, shown as b_{10} in the figure, is presented the network predicts an A.

3.5.2 SCN

The classification of the SRNs was based on self-derivatives and so should the classification of the SCN be, since we want to be able to compare the two networks. The activation of c_1 given the architecture of the SCN used for the one-bit representation is determined by[4]

$$\begin{aligned} c_1(t) = \Gamma(c_1(t-1)(i(t)W_{c_1W_{ic_1}} + W_{c_1W_{bc_1}}) + \\ c_2(t-1)(i(t)W_{c_2W_{ic_1}} + W_{c_2W_{bc_1}}) + \\ i(t)W_{bW_{ic_1}} + W_{bW_{bc_1}}) \end{aligned} \tag{14}$$

and the self-derivative of the activation of c_1 by

$$\begin{aligned} \frac{\partial c_1(t)}{\partial c_1(t-1)} =& (i(t)W_{c_1W_{ic_1}} + W_{c_1W_{bc_1}}) \cdot \\ & \Gamma'(\ c_1(t-1)(i(t)W_{c_1W_{ic_1}} + W_{c_1W_{bc_1}}) + \\ & \quad c_2(t-1)(i(t)W_{c_2W_{ic_1}} + W_{c_2W_{bc_1}}) + \\ & \quad (i(t)W_{bW_{ic_1}} + W_{bW_{bc_1}})) \end{aligned} \tag{15}$$

and likewise for the other context node c_2. This means that the sign of the self-derivative of c_1, $\frac{\partial c_1(t)}{\partial c_1(t-1)}$, is determined by $(i(t)W_{c_1W_{ic_1}} + W_{c_1W_{bc_1}})$. Since $i(t)$ can have two values, 0 and 1 when the input is A and B respectively (see table 1), and the second order weights are fixed, the derivative may have two different signs, given the

[4]See equation 25 for a general description of the computation in the SCN.

input. If instead the two-bit representation is used, the activation of c_1 is given by

$$\begin{aligned} c_1(t) = \Gamma(&c_1(t-1)(i_1(t)W_{c_1 W_{i_1 c_1}} + i_2(t)W_{c_1 W_{i_2 c_1}} + W_{c_1 W_{bc_1}}) + \\ &c_2(t-1)(i_1(t)W_{c_2 W_{i_1 c_1}} + i_2(t)W_{c_2 W_{i_2 c_1}} + W_{c_2 W_{bc_1}}) + \\ &i_1(t)W_{bW_{i_1 c_1}} + i_2(t)W_{bW_{i_2 c_1}} + W_{bW_{bc_1}}) \end{aligned} \tag{16}$$

and the self-derivative of c_1 is then

$$\begin{aligned} \frac{\partial c_1(t)}{\partial c_1(t-1)} =&(i_1(t)W_{c_1 W_{i_1 c_1}} + i_2(t)W_{c_1 W_{i_2 c_1}} + W_{c_1 W_{bc_1}}) \cdot \\ &\Gamma'(c_1(t-1)(i_1(t)W_{c_1 W_{i_1 c_1}} + i_2(t)W_{c_1 W_{i_2 c_1}} + W_{c_1 W_{bc_1}}) + \\ &\quad c_2(t-1)(i_1(t)W_{c_2 W_{i_1 c_1}} + i_2(t)W_{c_2 W_{i_2 c_1}} + W_{c_2 W_{bc_1}}) + \\ &\quad i_1(t)W_{bW_{i_1 c_1}} + i_2(t)W_{bW_{i_2 c_1}} + W_{bW_{bc_1}}) \end{aligned} \tag{17}$$

and likewise for c_2. That means that the sign of the self-derivative of c_1, $\frac{\partial c_1(t)}{\partial c_1(t-1)}$, is determined by the sign of $(i_1(t)W_{c_1 W_{i_1 c_1}} + i_2(t)W_{c_1 W_{i_2 c_1}} + W_{c_1 W_{bc_1}})$, i.e. also in this case the sign may differ given the input. Even when the two-bit representation is used, there are only two different signs of the self-derivative, since there are only two possible inputs for the A^nB^n-language, A and B.

Therefore the signature of the SCN must be defined such that the different dynamical behaviours of the network, for both input A and B, are reflected. In addition to this, we need to distinguish between networks which change the sign of the self-derivatives in the transition between the symbols and those which have the same self-derivative for both A and B. The result is a signature of the form (p_a, q_a, p_b, q_b, c) where p_a, q_a, p_b and q_b are determined as for the SRN given different inputs, i.e. p_a and q_a are the number of positive and negative self-derivatives respectively given A

as input and correspondingly for p_b and q_b for B. The additional *conditional* element c is 1 if a change of any sign of the self-derivatives occurs, otherwise $c = 0$. There are ten possible instances of this signature which are shown in table 3.

(p_a, q_a, p_b, q_b, c)	Input A	Input B	Dynamics
$(2, 0, 2, 0, 0)$	Monotonic	Monotonic	Not changed
$(2, 0, 1, 1, 1)$	Monotonic	Exotic	Changed
$(2, 0, 0, 2, 1)$	Monotonic	Oscillating	Changed
$(1, 1, 2, 0, 1)$	Exotic	Monotonic	Changed
$(1, 1, 1, 1, 0)$	Exotic	Exotic	Not changed
$(1, 1, 1, 1, 1)$	Exotic	Exotic	Changed
$(1, 1, 0, 2, 1)$	Exotic	Oscillating	Changed
$(0, 2, 2, 0, 1)$	Oscillating	Monotonic	Changed
$(0, 2, 1, 1, 1)$	Oscillating	Exotic	Changed
$(0, 2, 0, 2, 0)$	Oscillating	Oscillating	Not changed

Table 3: The ten possible signatures for the SCN. The terms "monotonic", "oscillating" and "exotic" are described in the corresponding table for the SRN, table 2. The last column emphasises the meaning of the c in the signature, i.e. whether any self-derivative's sign is changed in the transition between the two possible inputs.

Note that c is redundant for all signatures except for $(1, 1, 1, 1, 0)$ and $(1, 1, 1, 1, 1)$ where c is necessary to separate the type of networks where the signs of the self-derivatives are "swapped" between the two context nodes on the transition between A- and B-inputs. Some typical behaviours of networks with these signatures are shown in figures 8 to 15. Not all signatures were present among those that solved the A^nB^n-language, therefore not all signatures are represented in those figures.

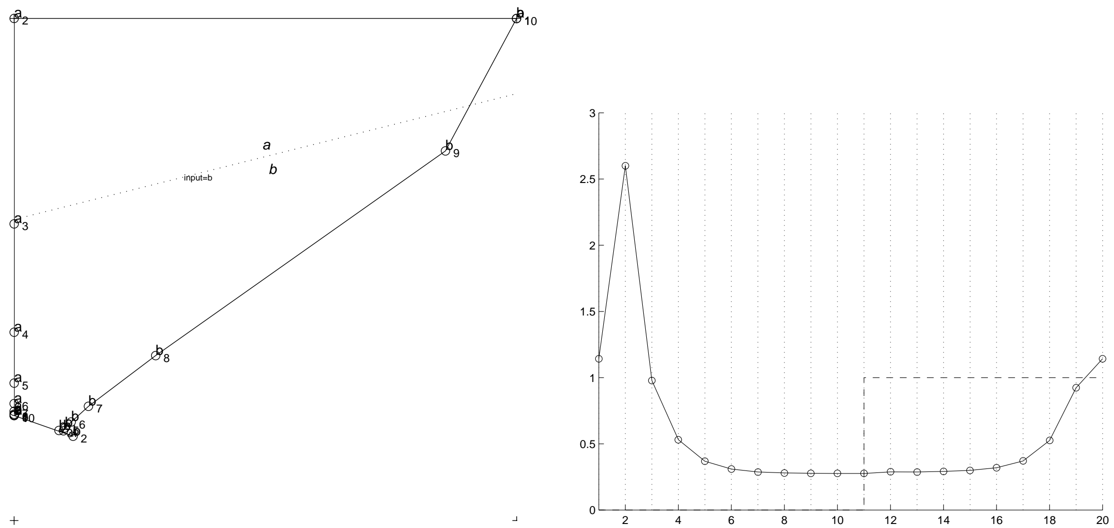

Figure 8: An example of the internal behaviour of an SCN with signature $(2, 0, 2, 0, 0)$. The left diagram shows the activation of the two context nodes plotted over time. The dotted line is the hyperplane, separating prediction of As from prediction of Bs when B is the input. The other hyperplane, for A as input is outside the graph in this case. The right diagram shows the dynamic hyperplane of the function network plotted over time, for a more detailed description of this diagram see figure 5. Note, in both diagrams, how the network when receiving the last B, predicts an A.

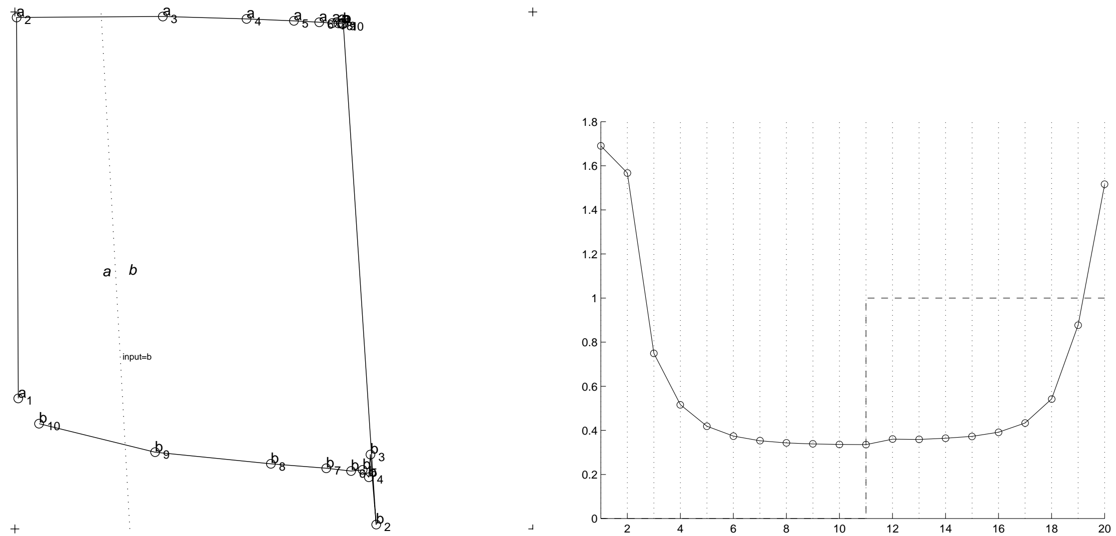

Figure 9: An example of the internal behaviour of an SCN with signature $(2, 0, 1, 1, 1)$. See figure 8 for a more detailed description.

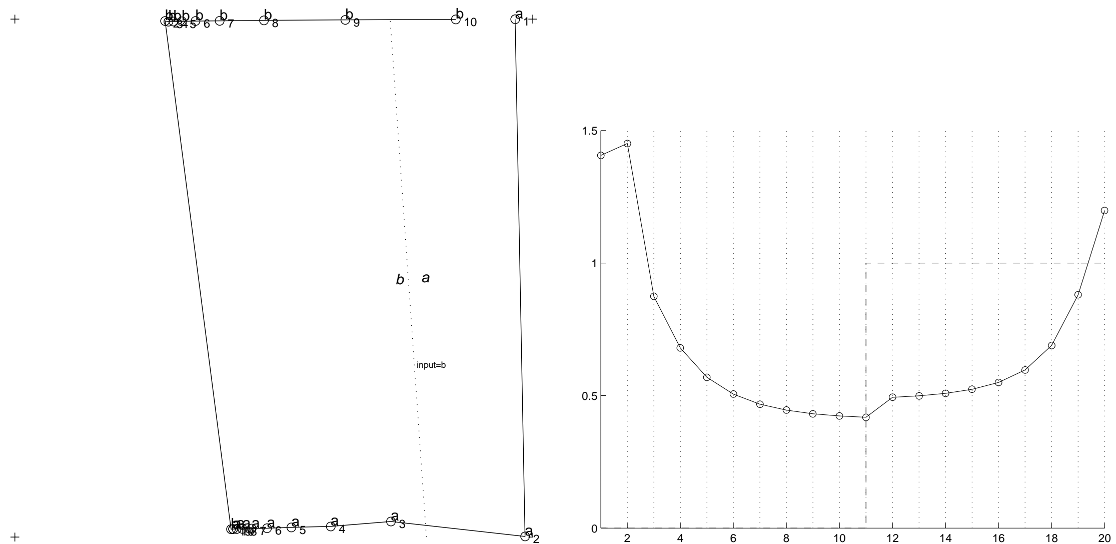

Figure 10: An example of the internal behaviour of an SCN with signature $(1, 1, 2, 0, 1)$. See figure 8 for a more detailed description.

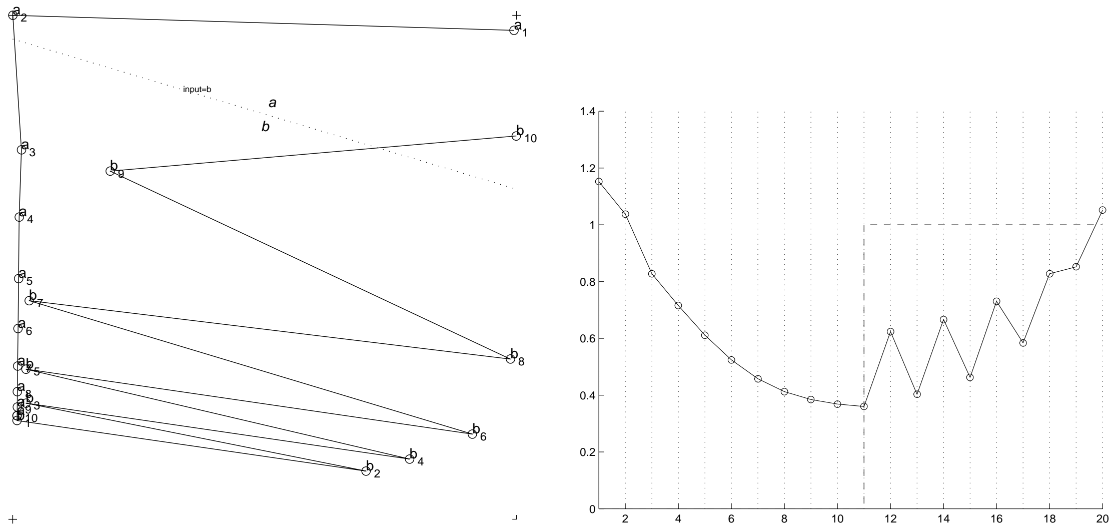

Figure 11: An example of the internal behaviour of an SCN with signature $(1, 1, 1, 1, 0)$. See figure 8 for a more detailed description.

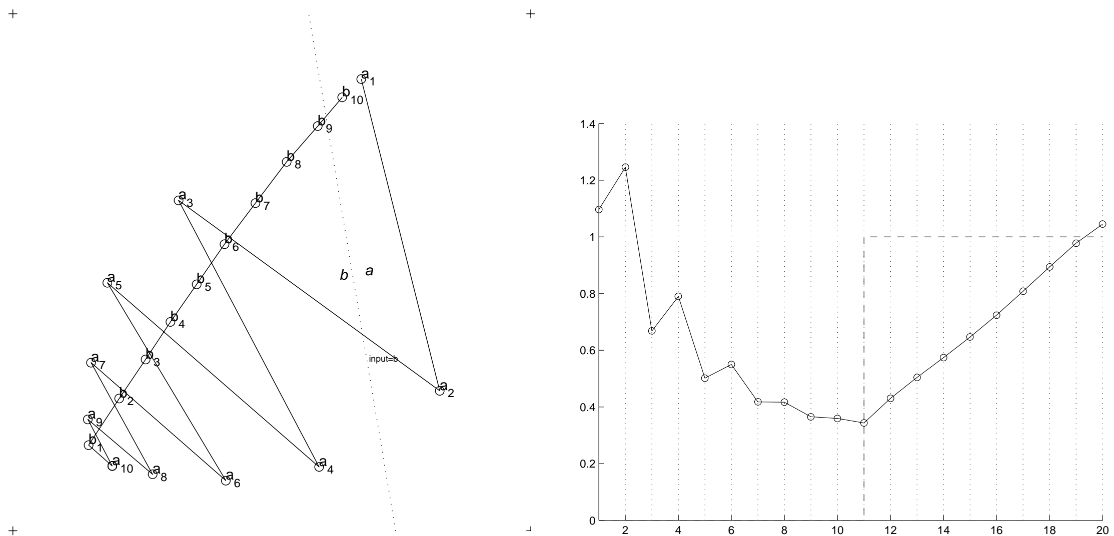

Figure 12: An example of the internal behaviour of an SCN with signature $(1, 1, 1, 1, 1)$. See figure 8 for a more detailed description.

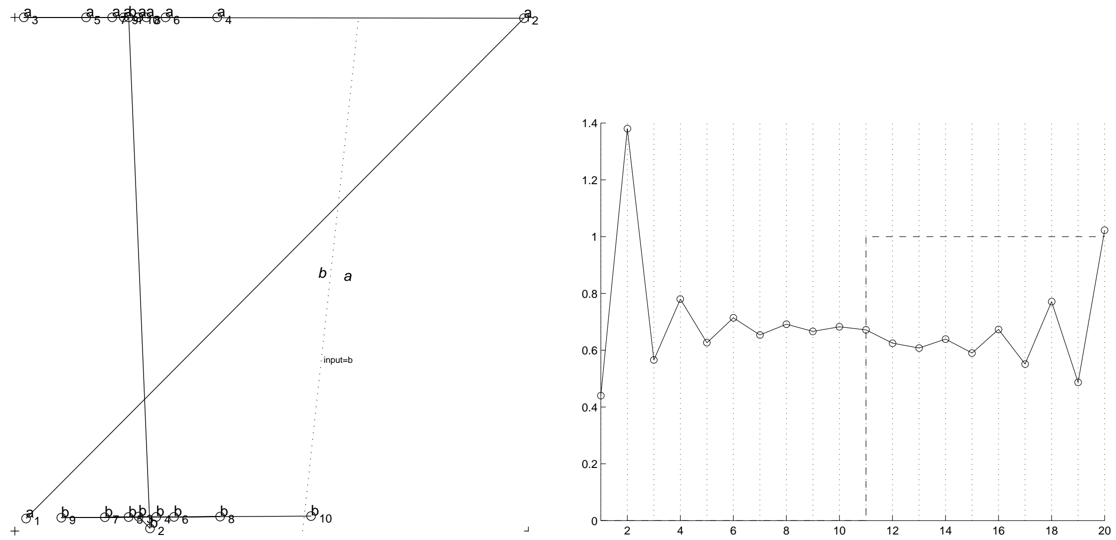

Figure 13: An example of the internal behaviour of an SCN with signature $(1, 1, 0, 2, 1)$. See figure 8 for a more detailed description.

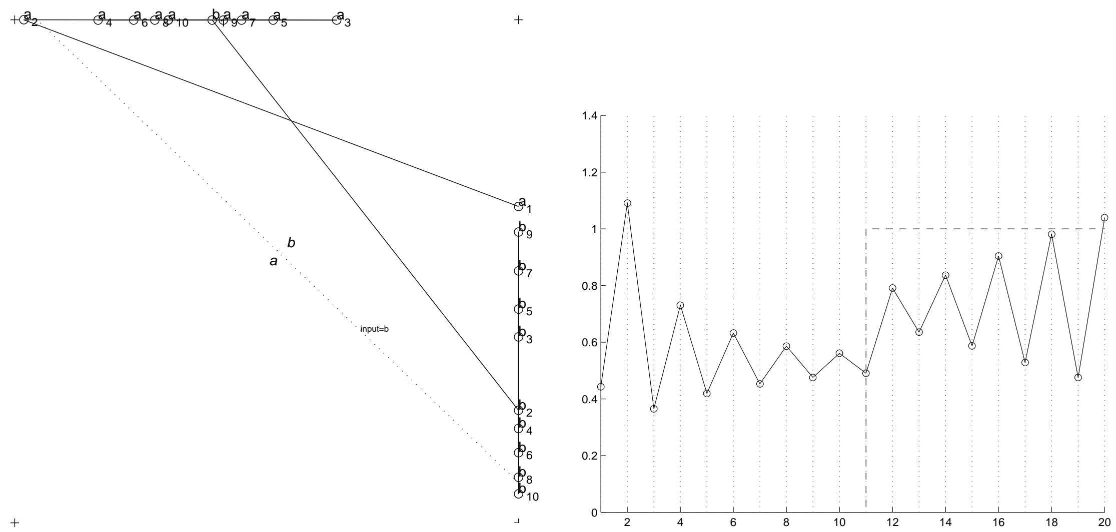

Figure 14: An example of the internal behaviour of an SCN with signature $(0, 2, 1, 1, 1)$. See figure 8 for a more detailed description.

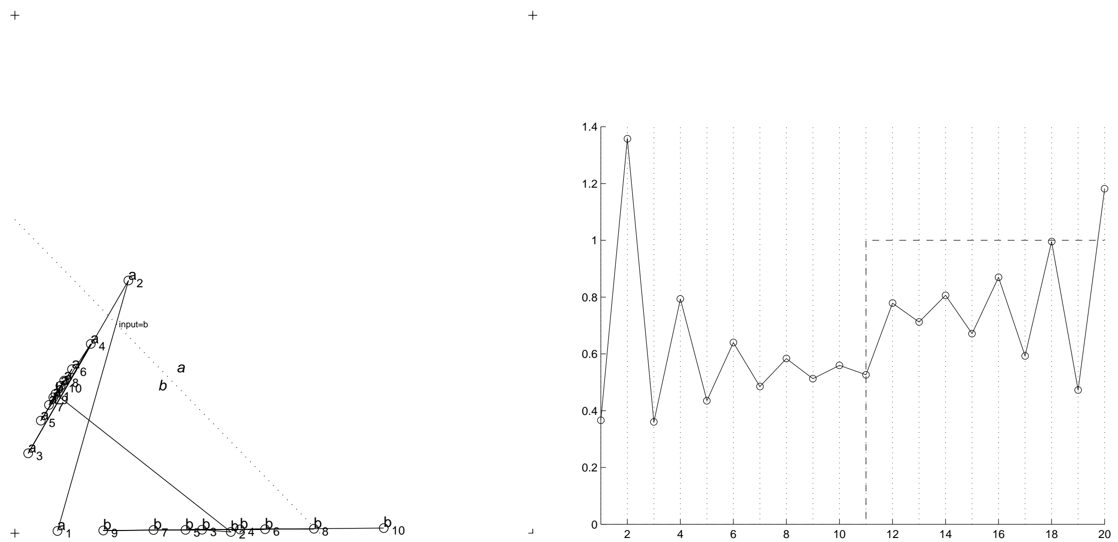

Figure 15: An example of the internal behaviour of an SCN with signature $(0, 2, 0, 2, 0)$. See figure 8 for a more detailed description.

3.6 Summary

Unlike the SRN, the output node of the SCN, does not get different inputs from the previous layer of nodes depending on the context. Instead the context determines the weights leading to the output node, such that the same input may be interpreted differently by the output node at different times. This is fundamentally different from how the SRN uses the state in the context units.

That the function network of the SCN changes when the context changes is not surprising, it is part of the very definition of an SCN. More interesting results are found when the decision hyperplanes of the output unit are calculated on the activation space of the context nodes. Then it is found that there is one hyperplane for each possible input.

The SCN can, at least in principle, also adopt different dynamical behaviours for different inputs, e.g. if both parameters are oscillating for A and monotonous for B. This is not possible for the SRN, if a context node is oscillating it must always be oscillating, no matter what the input is. That the SCN can alter its dynamical behaviour suggests that it can solve more complex problems than the SRN. As the number of signatures for each type of network suggests, the SCN has more possibilities for solving a problem.

Chapter 4

Experiments

4.1 Quantitative measures

The quantitative analysis of this project is aiming at mapping the measurable differences between the architectures and training methods. The measurements are made in five different dimensions and determined by simulations of the architectures.

- *Efficiency.* How many strings must be evaluated before a network solves the A^nB^n with $1 \leq n \leq N$. The efficiency can be evaluated for different values of N.

- *Reliability.* How many of the initial networks are trained into successful solutions by the training algorithm? This can be shown for different values of N.

- *Quality.* The training algorithms use simple measures to determine the successfulness of the networks as a termination criteria. If these simple measures classify networks as successful, how "good" are these networks really, if they are

more extensively tested? The quality is measured on the successful networks to determine, with a higher significance, the true "successfulness" of the network.

- *Consistency.* When a solution was found, how long was it sustained by the training algorithm? This can be shown for different values of N and is measured by counting the number of epochs when N is not decreased again. The consistency for the EA is expected to be near 100% since no individuals are replaced until a better individual is generated. Some individuals may be lost, however, since the evaluation is indeterministic.

- *Ability to generalize*, i.e. if the network is trained for some subclass of the problem, can it generalize to other problem instances? This corresponds to correctly classifying longer strings than the network has been trained on in the A^nB^n-problem.

The efficiency, reliability and consistency were also measured in [TBW99]. The generalization ability is an important measure and all previous work on SRNs and the A^nB^n-grammar has implicitly or explicitly measured this. The quality measure was added to be able to see whether the evaluation method (see section 4.2) was successful in separating successful from unsuccessful networks.

4.2 The evaluation of correctness

After a complete string has been predicted by the network the activation of the context nodes should preferably return to a default initial state. But this initial state may differ slightly depending on the contents of the previous string. Therefore the

network should be tested many times per string length to account for more than one initial state.

This is solved by letting the networks be evaluated on a *randomly* ordered *evaluation set*, Φ_k^N, of A^nB^n strings with $1 \leq n \leq N$ and k instances of every individual length. In all experiments N was set to 10 since this was the target length we wanted the networks to solve.

Definition 2 In order for a network to be classified as *successful for length L on evaluation set* Φ_k^N it has to correctly predict every predictable symbol, i.e. the first A and all but the first B, in all strings in Φ_k^N with $n \leq L$.

Note that, since the evaluation set is indeterministic, the determination of successfulness of the network is also indeterministic, and a network can be classified as both successful and unsuccessful with different instantiations of the evaluation set.

Depending on the value of k in Φ_k^N the "difficulty" of the testing procedure varies. With a low k, e.g. Φ_1^N, the specificity of the evaluation is also low and a network might pass the test "by accident" although it seldom predicts all strings correctly. With higher values of k the quality of the passed networks is more probable to be higher, but the evaluation takes more computational power to perform and more time is needed for the training algorithm to find these better networks. So the choice of k is a matter of choosing between quality and efficiency.

4.3 Details of the simulations

4.3.1 BP

The simulations of BP always started by generating a network with random weights uniformly distributed in the interval $[-0.1, 0.1]$. The network was trained on a set of random ordered strings, from the A^nB^n-language, skewed to a higher distribution of shorter strings, with one exception, $n = 2$ was more frequent than $n = 1$. The distribution is shown in table 4.

n	1	2	3	4	5	6	7	8	9	10
%	19.27	29.33	14.33	10.91	8.46	6.43	4.79	3.37	2.12	1.00

Table 4: The distribution of lengths in the training set for BP. The first row shows the length of the string in terms of n in the A^nB^n-grammar and the second shows the distribution in percent. In previous work on the A^nB^n-grammar, the distribution of lengths has been biased towards shorter strings. Therefore this is implemented also in this project. The highest distribution was chosen for $N = 2$.

In every epoch the network was evaluated using the testing procedure described in section 4.2. Two evaluation sets of different sizes were tested on different simulations, Φ_3^{10} and Φ_1^{10}, i.e. three strings per length and one string per length. The use of Φ_3^{10} is motivated by the fact that the EA uses the same evaluation set (see section 4.3.2), but since this evaluation set seemed to tough for the BP (see table 5 and 6 on pages 46 and 47 respectively) the evaluation set Φ_1^{10} was also tested.

Both the SRNs and SCNs were tested on both the one-bit and the two-bit representation. The one-bit representation gives smaller networks which therefore are more efficient to train, but the two-bit representation may provide BP with useful redundant information. Whether this information is useful or not will be clear from

these experiments.

The SRNs were trained with BP only, not BPTT (see section A.2.1). This simplification was made since there is no accessible training method for SCNs similar to BPTT for the SRNs, and to develop such a training algorithm is beyond the scope of this project. The BP results will not be the best achievable since BP is not very suitable for sequential tasks, but hopefully the comparison becomes more reliable when both network types are trained with "equivalent" algorithms.

The SCN was trained using the "backspace trick" described in appendix B. For both network types the training algorithm calculated and summed up all the error gradients during the whole string and updated the weights after the string was completed.

The training of both networks was carried out with different values of the learning rate, η. Every training session lasted 500 000 epochs.

4.3.2 EA

As mentioned in section 2.4, an EA may produce more comparable and less biased results than BP. The principle of the EA is described in appendix C. The fitness function that will be used is based on how well the language is predicted. First we define a correctness function c

$$c(x,y) = \begin{cases} 0 & x \neq y \\ 1 & x = y \end{cases} \tag{18}$$

where x and y are symbols from a string. The fitness, f, of an individual network in the population is then determined by the function

$$f = \sum_{s \in \Phi_3^{10}} 2 \cdot \frac{\sum_{i=\frac{|s|}{2}}^{|s|} c(F(s_i), s_{i+1})}{|s|} \tag{19}$$

where $|s|$ is the length of string s, Φ_3^{10} the evaluation set as defined in section 4.2, s_i is the ith symbol in s, F is the function which the network represents[5] and $s_{|s|+1}$ should be interpreted as the first symbol of the succeeding string, i.e. an A[6]. The effect of this function is that the network gets up to one "point" for every correctly predicted string, but also that it get fractions of a "point" if only a part of the predictable part of the string is correctly predicted. The decision to use the evaluation set Φ_3^{10} is based on results from initial experiments that showed that bigger evaluation sets and/or longer strings considerably increased the simulation time needed. Any smaller evaluation sets did not give very interesting results either. The fitness function is in fact just an extension of the evaluation method described in section 4.2. The reason that the evaluation method is not used as a fitness function directly is that the EA is helped by differentiating between networks that predict parts of a string correctly from those that miss the whole string. Initial tests indicated that such a differentiation was helpful for the EA.

Why was not then the SSE chosen as a basis for the fitness function? This would have improved the comparison between BP and EA since both then would use the

[5]In other words, $F(s_i)$ is the prediction made by the network interpreted binary. F varies with the context of the network, but that has been left out of the equation to improve readability.

[6]If s is the last string, $s_{|s|+1}$ should still be an A although no string comes after.

same function. The explanation is simply that initial experiments showed that the EA performed much better with this fitness function than if only the SSE would be used. This is perhaps best explained by the fact that the fitness function now is based on the very same method which determines the successfulness of the network. Since the fitness function is not derivable, this can not be implemented for BP, which may be one of the reasons that the BP performed very bad (see chapter 5).

The networks of the initial population were initialized with uniformly distributed random weights in the interval [-1,1][7]. The population size was set to 100 of which 20 of the best, according to the fitness function in equation 19, were selected each generation as the "elite" of the population. The elite group was simply copied, unchanged, to the succeeding population. In each generation, a set of 80 new individual replaced the 80 worst individuals by randomly copying members of the "elite" and then mutate each weight by adding a Gaussian distributed value to them (see appendix C for a description of the mutation). Three different values of the mutation-rate, σ, were tested. The EA was tested for 20 000 generations in all experiments. Only one-bit representation were used in all EA experiments. The two bit representation would probably only slow down the EA and not provide it with any useful redundant information. Also, the analysis of hyperplanes becomes easier when only one output node per network needs to be considered.

[7]Not [-0.1,0.1] as for the BP since these intervals were "optimized" from the results of initial experiments.

Chapter 5

Results

This chapter will present the results according to the performance measures defined in section 4.1. All results are based on 100 experiments, for each setting of all parameters. In most cases the results are based on fewer networks, however, since there was not 100% correct networks at all times. If there were only 10% correct networks, for example, the result would be the average of the results of these 10 networks.

5.1 Reliability

The reliability is defined as the percentage of the experiments that lead to successful solutions, i.e. the highest possible reliability is 100%. The successfulness of a network is defined, for every string length. Therefore the reliability is measured individually for every string length in the training set, i.e. for $1 \leq N \leq 10$, based on definition 2 on page 36. To measure the reliability of shorter string-lengths than the target length allows that all experiment to be compared, even those not solving the target length. The reliability was measured for every network architecture in combination

with both training algorithms in order to allow conclusions about both the reliability of the architectures and the training algorithms.

Tables 5 and 6 show the reliability of the backpropagation algorithm for different settings of the learning rate (η) different evaluation sets. Table 5 shows the results for the two-bit representation of the A^nB^n-language and table 6 the one-bit representation. See section 3.1 for a description of the different representations. Table 7 shows the reliability of the EA with different settings of the mutation parameter (σ). The results will be discussed in detail in the following subsections.

5.1.1 BP

SRN vs. SCN

From table 5 and 6 it can be concluded that the reliability of BP is considerably higher for SRNs than for SCNs. Actually, no solution for $N = 10$ was ever found for the SCN by BP. The best solution for the SCN is for $N = 7$, but only one of a hundred simulations found this.

The influence of η

There seems to be no clear correlation between the learning rate (η) and the reliability. The learning rate $\eta = 0.1$ gave the best results in most cases, both for the SRN and SCN. However, when the two-bit representation and the evaluation set Φ_3^{10} were used, the best results for the SRN were obtained with $\eta = 0.05$.

One-bit vs. two-bit representation

To use two representations of the A^nB^n-language was motivated in section 3.1 by that the redundant information in the two-bit representation could provide the training algorithm with useful information and that the one-bit representation reduced computational overhead. The reliability was measured for both types of representations in order to analyse if the representation influenced the performance and the results do indeed indicate such differences.

The performance may not only be affected by the existence of redundant information in the representation. The two-bit representation also requires that the network correctly predicts both bits of the symbol representation while the one-bit representation only requires one correctly predicted bit at a time.

From the differences between the results in tables 5 and 6 it can be concluded that the choice of representation in combination with the architecture and evaluation set affects the performance. For the evaluation set Φ_3^{10} the reliability of both architectures decreases if we switch to the one-bit representation. If we consider Φ_1^{10}, the situation is a bit more complex.

Φ_3^{10} vs. Φ_1^{10}

The evaluation set Φ_1^{10} has a higher probability of evaluating possibly unsuccessful networks as successful than Φ_3^{10}, since the network is tested on three times as many strings in Φ_3^{10}. Since the evaluation sets are the basis of the reliability measurement, it can be expected that the reliability is higher for Φ_1^{10} than Φ_3^{10}. The results in table 5 and 6 confirm this expectation with a few exceptions. The main exceptions are found for the lowest η tested, i.e. $\eta = 0.05$. In table 6 it can be found that for this

value of η, a higher reliability is measured for the SRN when $2 \leq N \leq 9$, Φ_3^{10} and the two-bit representation is used. The SCN also has a slightly higher reliability for Φ_3^{10} when the one-bit representation was used. Other than this, there are a number of non-significant exceptions.

Although the use of Φ_1^{10} generally results in significantly higher reliability, the quality of the resulting networks may be lower, than those evaluated with Φ_3^{10}, due to the higher probability of networks being misclassified as successful. This is covered in section 5.2.

5.1.2 EA

SRN vs. SCN

The reliability of the EA is shown in table 7. When $\sigma = 1.0$ and $\sigma = 0.5$ The SRN has a higher reliability than the SCN, but when $\sigma = 0.1$ the opposite situation occurs and the reliability of the SRN is significantly lower.

It appears as if the reliability of the SCN is less influenced by the choice of σ while the reliability of the SRN is quite dependent on not having a too low mutation parameter.

The influence of σ

It is clear from table 7 that higher reliability is achieved for higher values of the mutation parameter, σ. This indicates that there are local optima which, when σ is too low, are "inescapable" for the EA.

5.1.3 EA vs. BP

It is obvious from the results in table 5, 6 and 7 that the EA is a much more reliable method for training networks in the A^nB^n domain. The reliability of the EA for finding successful networks solving length 10 is higher than all BP-simulations for all values of σ tested, except for the SRN when $\sigma = 0.1$. The reliability of the BP for finding SRNs is typically very low and the highest reliability was 32% (see table 6), while the corresponding values of the EA are around 95% except when the mutation parameter was too low. BP never found any successful SCN for length 10 while the EA never had a reliability lower than 70%.

	Evaluated on Φ_3^{10}					
	$\eta = 0.3$		$\eta = 0.1$		$\eta = 0.05$	
N	SRN	SCN	SRN	SCN	SRN	SCN
1	94	78	100	86	92	78
2	90	54	79	64	77	42
3	45	37	68	9	74	3
4	4	1	4	6	11	3
5	3	0	4	2	11	0
6	1	0	4	0	11	0
7	1	0	4	0	11	0
8	0	0	4	0	11	0
9	0	0	3	0	10	0
10	0	0	2	0	6	0
	Evaluated on Φ_1^{10}					
	$\eta = 0.3$		$\eta = 0.1$		$\eta = 0.05$	
N	SRN	SCN	SRN	SCN	SRN	SCN
1	95	79	99	82	94	81
2	91	48	83	64	70	51
3	65	33	76	23	68	10
4	6	4	9	4	9	9
5	6	2	9	2	9	4
6	4	0	9	2	9	1
7	3	0	9	0	9	0
8	2	0	9	0	9	0
9	0	0	9	0	9	0
10	0	0	9	0	9	0

Table 5: Reliability of BP when two-bit representation of the A^nB^nlanguage was used, shown in percent of the 100 simulations which found successful solutions for length N. The top half of the table shows the results for when evaluation set Φ_3^{10} was used and the lower the corresponding results for Φ_1^{10}. The three main columns show the results for different learning rates.

	Evaluated on Φ_3^{10}					
	$\eta = 0.3$		$\eta = 0.1$		$\eta = 0.05$	
N	SRN	SCN	SRN	SCN	SRN	SCN
1	100	90	97	87	98	85
2	98	48	91	33	92	14
3	80	28	69	3	63	1
4	25	0	48	2	40	0
5	24	0	44	0	26	0
6	19	0	36	0	7	0
7	18	0	36	0	3	0
8	10	0	14	0	0	0
9	6	0	3	0	0	0
10	0	0	0	0	0	0
	Evaluated on Φ_1^{10}					
	$\eta = 0.3$		$\eta = 0.1$		$\eta = 0.05$	
N	SRN	SCN	SRN	SCN	SRN	SCN
1	99	90	97	90	100	77
2	98	43	91	41	93	8
3	96	32	87	8	81	0
4	41	0	41	2	50	0
5	40	0	41	2	50	0
6	40	0	41	2	50	0
7	40	0	41	1	45	0
8	38	0	41	0	36	0
9	32	0	36	0	25	0
10	20	0	32	0	13	0

Table 6: Reliability of BP when the the one-bit representation of the $A^n B^n$language was used, showed in percent of the simulations which found successful solutions for length N. See the previous table for an explanation of the columns.

	$\sigma = 1.0$		$\sigma = 0.5$		$\sigma = 0.1$	
N	SRN	SCN	SRN	SCN	SRN	SCN
1	100	100	100	100	99	99
2	99	95	98	81	38	72
3	96	90	98	79	35	70
4	96	88	96	79	31	70
5	96	87	95	79	31	70
6	96	86	95	79	31	70
7	96	86	95	79	31	70
8	96	86	95	79	31	70
9	96	86	95	78	31	70
10	95	86	95	78	31	70

Table 7: Reliability of EA for different settings of the mutation parameter σ. Shown in percent of how many of the simulations that lead to successful solutions for each length N.

5.2 Quality of successful networks

The quality was measured on all networks that were classified as successful by the evaluation procedure of the training algorithm. The quality of a network was tested by letting the network predict strings from the evaluation set Φ^{10}_{1000}, i.e. 1000 instances of each string of the training set. The *quality* of a network is defined for each length N individually as the percentage of successfully predicted strings of length N in Φ^{10}_{1000}. The *total quality* of a network is defined as the average quality on all string lengths.

The results in table 8, 9 and 10 all show the average quality of all the *successful* networks for each test. Since the number of successful networks varies widely between the different experiments, the significance of the averaged result is varying, i.e. the results are more statistically significant when the reliability was high. This is important to keep in mind when the results are compared. Take for example the BP experiment with the two-bit representation, $\eta = 0.1$ and evaluation set Φ^{10}_{3} in table 8. There, only two successful SRNs for $N = 10$ were found (see table 5), which means that the resulting average quality is based only on these two networks.

Some results could not be obtained for BP, since it was not always successful in finding any solution. A "——" in a table indicates that no data was available because of this.

5.2.1 BP

Since no successful SCNs were found with BP, no quality measures of the SCNs can be obtained.

The influence of η

In table 8 we can see that, when the two-bit representation was used, the quality seems to decrease when η is decreased for both evaluation sets. Except for the shorter strings with $N = 1$ or $N = 2$, where the quality was higher for the low η. The latter may indicate that a lower value of η biased BP towards finding networks that primarily solved shorter strings. These observations are however based on vague grounds, since very few successful networks were found in these experiments (see table 5).

In table 8 we can see that the correlation between η and quality is less distinct when the one-bit representation was used. It is clear, though, that $\eta = 0.3$ gave the networks with best quality.

One-bit vs. two-bit representation

Since no networks were successful with the one-bit representation and Φ_3^{10}, only networks evaluated with Φ_1^{10} can be compared. This vague comparison only reveals that the two-bit representation gave networks with a slightly higher quality for the SRNs, which may indicate that the redundant information of the two-bit representation was useful.

5.2.2 EA

SRN vs. SCN

Table 10 shows that the quality of both types of networks trained by the EA was typically very high (between 94 and 100%) and no significant difference can be seen between SRNs and SCNs from the total quality of the networks except that the SRNs had slightly higher qualities when $\sigma = 1.0$ and $\sigma = 0.5$.

The influence of σ

When σ is decreased the quality variance in table 11 of the SRNs decreases while the same attribute of the SCNs increases. Using a lower mutation parameter also gave SCNs with higher quality. Why the mutation parameter has this effect on the networks is difficult to speculate in. But the effect is clear and quite significant. More experiments and further details must be obtained before any explanation can be given.

The influence of N

As mentioned earlier, the quality of the networks is varying for different values of N. But not as could be expected. The short strings should be easier to correctly predict since fewer symbols must be predicted and less “counting” is required by the networks. Therefore the most reasonable expectation is that the quality of shorter strings should be higher than for the longer strings. But the results in table 10, indicate that this is not always the case. This is clarified in table 11 where the relative quality is shown. For $N = 1$ the results agree with the expectation since all the relative qualities are positive. Also for $N = 9$ and $N = 10$, the relative quality matches our expectation since all but one set of networks show a negative relative quality. For all other lengths, the relative quality varies without any clear pattern, except for $N = 6$ which appears to be an “easy” length to predict, for all networks.

Why the qualities of the networks did not increase linearly, or almost linearly, with the string lengths, but instead varying, seemingly randomly, cannot easily be explained from the results in table 10 or 11. Therefore more experiments, aimed on explaining this feature, should be conducted.

5.2.3 EA vs. BP

The results in table 8, 9 and 10 clearly show that the EA outperforms BP in terms of quality. The focus of this comparison should lie on the results of the BP simulation with evaluation set Φ_3^{10}, since this is the same set that the EA uses. These results have the highest quality of all BP-trained networks, but still, the EA outperforms them as well.

	Evaluated on Φ_3^{10}		
	$\eta = 0.3$	$\eta = 0.1$	$\eta = 0.05$
N	SRN	SRN	SRN
1	—	85.15	90.73
2	—	85.00	88.83
3	—	89.60	80.72
4	—	92.70	77.32
5	—	93.65	78.43
6	—	90.40	77.38
7	—	92.15	76.25
8	—	92.70	77.13
9	—	72.05	76.52
10	—	92.60	73.95
$\overline{x}$	—	88.60	79.73
	Evaluated on Φ_1^{10}		
	$\eta = 0.3$	$\eta = 0.1$	$\eta = 0.05$
N	SRN	SRN	SRN
1	—	64.54	73.42
2	—	77.03	81.21
3	—	91.24	69.73
4	—	89.94	67.73
5	—	87.73	67.36
6	—	88.42	69.01
7	—	86.69	68.10
8	—	82.94	67.79
9	—	80.07	65.49
10	—	67.66	52.07
$\overline{x}$	—	81.63	68.19

Table 8: Quality of successful networks found by BP when the two-bit representation of the A^nB^n-language was used. Shown in percent of how many of all strings of each string length that were correctly predicted. The last row shows the total quality of all string lengths up to $N = 10$. There are no results for the SCNs since no successful SCNs were found by BP.

	Evaluated on Φ_3^{10}		
No successful networks found.			
	Evaluated on Φ_1^{10}		
	$\eta = 0.3$	$\eta = 0.1$	$\eta = 0.05$
N	SRN	SRN	SRN
1	79.38	80.56	73.82
2	76.23	77.63	74.01
3	73.75	53.94	59.37
4	67.45	51.75	54.08
5	65.58	49.05	51.03
6	69.62	50.68	54.86
7	67.37	50.82	52.57
8	68.22	43.93	50.67
9	58.71	46.99	47.18
10	63.21	32.86	38.67
$\overline{x}$	68.95	53.82	55.62

Table 9: Quality of BP when the one-bit representation of the A^nB^n-language was used. Since no successful solutions were found for Φ_3^{10}, this part of the table has been removed.

	$\sigma = 1.0$		$\sigma = 0.5$		$\sigma = 0.1$	
N	SRN	SCN	SRN	SCN	SRN	SCN
1	99.17	96.28	99.38	98.60	97.98	99.44
2	96.45	94.86	99.23	97.84	97.58	95.30
3	96.34	93.93	99.02	98.15	98.59	99.24
4	98.58	95.63	97.70	97.86	94.90	98.68
5	99.27	95.26	97.38	97.76	96.67	98.84
6	99.57	95.14	98.56	98.41	98.54	98.70
7	99.06	95.12	98.74	97.83	97.10	99.46
8	99.63	95.18	98.39	97.23	97.12	96.33
9	94.02	94.73	98.18	97.20	96.97	96.73
10	96.68	94.37	95.45	95.15	97.90	96.84
$\overline{x}$	97.88	95.05	98.20	97.60	97.34	97.96

Table 10: Quality of EA solutions shown in percent of how many of all strings of each string length that were correctly predicted. The last row shows the total quality of all string lengths up to $N = 10$. The columns shows the results for the different values of σ and different architectures

	$\sigma = 1.0$		$\sigma = 0.5$		$\sigma = 0.1$	
N	SRN	SCN	SRN	SCN	SRN	SCN
1	1.29	1.23	1.18	1.00	0.64	1.49
2	-1.42	-0.19	1.03	0.24	0.25	-2.65
3	-1.54	-1.12	0.81	0.55	1.25	1.28
4	0.70	0.58	-0.50	0.26	-2.44	0.72
5	1.39	0.21	-0.83	0.15	-0.67	0.88
6	1.70	0.09	0.36	0.80	1.21	0.74
7	1.19	0.07	0.54	0.23	-0.23	1.50
8	1.75	0.13	0.19	-0.37	-0.21	-1.63
9	-3.86	-0.32	-0.02	-0.41	-0.36	-1.23
10	-1.19	-0.68	-2.75	-2.45	0.57	-1.12
$\overline{x^2}$	3.22	0.38	1.20	0.84	1.05	2.04

Table 11: The relative quality of the EA. The values are calculated from the values in table 10 by subtracting the total quality from the qualities of all individual string length. The relative quality is negative for a length if the networks are worse at predicting that specific string length and vice versa. The last row is the variance, i.e. $(\sum_{i=1}^{n}(x_i - \overline{x})^2)/n$, of the quality. The variance should be interpreted as a measure of how much the quality varies for different string lengths.

5.3 Efficiency

In order to make the efficiency of the EA and BP comparable we cannot simply compare the number of generations with the number of epochs, since one generation involves the evaluation of the whole population of networks. Instead the total number of evaluations is measured. Every individual network in the EA-population is evaluated once per generations which means that the number of evaluations per generation is the same as the size of the population. In BP the network is evaluated only once per epoch. Table 12, 13 and 14 contain the measured efficiency, for BP and EA, in terms of number of evaluations needed before a successful solution of length N was found. Since the number of successful networks decreased for higher values of N, the significance of the results is also decreased with higher values of N. For example, the resulting average efficiency of length $N = 10$ is normally based on a smaller set of networks than for $N = 9$ which is based on a smaller subset than for $N = 8$, and so on. This sometimes make the results look a bit peculiar, when the number of evaluations may be smaller for longer strings.

The efficiency, or rather the computational intensiveness, is also depending on the size of the evaluation set Φ_k^N. Since two different evaluation sets were used in the BP-experiments we must remember that the smaller set is less computationally intensive. The efficiency of the algorithm, when simply measured in number of evaluations, may however be higher for more intensive evaluation sets since it may somehow help the search.

The choice of representation of the A^nB^n-language also affects the computational power required to simulate the networks. If the one-bit representation is used, the SRN will have 11 and the SCN 18 weights and biases, with the two-bit representation

the SRN will have 16 and the SCN 36[8]. This does not only mean that the simulation of the networks themselves takes longer time with the two-bit representation but also that the training algorithm has more parameters to adjust. Again, the efficiency of the algorithm, if measured in number of evaluations, may be higher with the redundant information although it may cause the actual computation time to be longer.

5.3.1 BP

SRN vs. SCN

The results in section 5.1 made clear that no successful SCNs for length $N = 10$ were found by BP, i.e. more epochs than tested (500 000) were needed to find an SCN solution. Hence, the training of the SRNs were obviously more efficient.

The influence of η

If we consider the successful SRNs, the number of required evaluations is clearly increased if the learning rate is decreased (see table 12 and 13).

One-bit vs. two-bit representation

If we only compare the successful SRNs (for $N = 10$) we can see that more than twice as many evaluations were needed for the one-bit representation than for the two-bit representation. This indicates that the redundant information in the two-bit representation actually improved the optimization speed.

The efficiency of the SCN was, however, considerably improved, for the shortest strings, when the one-bit representation was used. This indicates that the redundant

[8]Only counting the second order weights and biases since the dynamical weights are not part of the parameters which the training algorithm adjusts.

information did not help BP in the training of SCNs. Perhaps the double number of adjustable weights (36 for two-bit representation instead of 18 for one-bit representation) affected the efficiency negatively.

5.3.2 EA

The EA was tested for $2 \cdot 10^4$ generations in each experiment. Since the number of evaluations was measured in table 14 and there were 100 evaluations per generation, the maximum number of evaluations is $2 \cdot 10^6$.

SRN vs. SCN

As can be seen in table 14, in most cases the EA was more efficient in finding successful weight sets for SCNs than for SRNs. This is surprising since the SCN had slightly more weights than the SRN (18 instead of 11).

The influence of σ

From table 14 we can conclude that most efficient mutation parameter setting tested was $\sigma = 0.5$ for both network architectures. The efficiency was considerably worse for $\sigma = 0.1$.

5.3.3 EA vs. BP

To compare the efficiency of BP and EA with each other is not straightforward since the algorithms are very different. Here, the number of evaluations has been used to compare the efficiency, but one must remember the rest of the computation in implementation the algorithms is not included in this comparison.

However, the rough comparison shows that BP, when successful, were more efficient than the EA. Also, based on practical experiences from the experiments, if the actual CPU-time of the simulations had been measured, BP would outperform the EA in terms of efficiency.

	Evaluated on Φ_3^{10}					
	$\eta = 0.3$		$\eta = 0.1$		$\eta = 0.05$	
N	SRN	SCN	SRN	SCN	SRN	SCN
1	9.92	60.51	21.62	38.85	39.50	81.14
2	36.72	147.60	58.47	117.42	111.73	272.44
3	25.06	164.71	54.07	130.30	112.94	419.66
4	40.21	424.11	44.45	122.17	117.38	419.66
5	56.61	> 500	53.29	165.75	136.78	> 500
6	46.35	> 500	57.97	> 500	147.67	> 500
7	46.35	> 500	73.80	> 500	167.17	> 500
8	> 500	> 500	78.54	> 500	196.87	> 500
9	> 500	> 500	68.40	> 500	212.30	> 500
10	> 500	> 500	63.50	> 500	242.06	> 500
	Evaluated on Φ_1^{10}					
	$\eta = 0.3$		$\eta = 0.1$		$\eta = 0.05$	
N	SRN	SCN	SRN	SCN	SRN	SCN
1	10.04	60.58	11.26	42.41	24.74	57.25
2	23.13	163.46	47.89	148.87	107.18	279.68
3	21.83	139.37	44.40	250.22	107.88	331.52
4	115.97	295.83	61.24	121.03	91.31	341.28
5	117.40	367.48	63.57	117.96	103.63	415.12
6	153.75	> 500	64.24	118.89	105.85	442.73
7	184.36	> 500	66.55	> 500	117.65	> 500
8	164.36	> 500	70.00	> 500	128.54	> 500
9	> 500	> 500	77.25	> 500	133.03	> 500
10	> 500	> 500	79.14	> 500	147.04	> 500

Table 12: The efficiency of BP when the two-bit representation was used. Showed in thousands of number of evaluations needed before a successful solution for length N were found, i.e. the lower value, the higher efficiency. The results are shown for both evaluation sets Φ_3^{10} andΦ_1^{10}, for all ηs tested and the both architectures, SRN and SCN.

	Evaluated on Φ_3^{10}					
	$\eta = 0.3$		$\eta = 0.1$		$\eta = 0.05$	
N	SRN	SCN	SRN	SCN	SRN	SCN
1	9.24	5.06	15.88	22.01	28.02	57.92
2	49.63	28.19	74.47	108.98	140.01	289.54
3	50.06	96.05	78.00	224.54	136.80	341.22
4	44.05	> 500	129.41	121.78	273.66	> 500
5	67.85	> 500	211.59	> 500	325.78	> 500
6	75.79	> 500	207.98	> 500	438.38	> 500
7	87.11	> 500	231.11	> 500	456.94	> 500
8	102.91	> 500	340.45	> 500	> 500	> 500
9	119.09	> 500	320.24	> 500	> 500	> 500
10	> 500	> 500	> 500	> 500	> 500	> 500
	Evaluated on Φ_1^{10}					
	$\eta = 0.3$		$\eta = 0.1$		$\eta = 0.05$	
N	SRN	SCN	SRN	SCN	SRN	SCN
1	6.76	8.50	16.42	23.20	29.07	45.33
2	44.85	33.03	71.88	103.76	131.96	293.16
3	46.75	87.58	79.69	173.61	137.16	> 500
4	43.51	> 500	106.72	239.89	259.18	> 500
5	45.66	> 500	115.55	240.34	278.78	> 500
6	48.43	> 500	119.95	247.58	289.97	> 500
7	53.21	> 500	138.96	178.86	309.23	> 500
8	61.72	> 500	161.83	> 500	343.65	> 500
9	68.08	> 500	173.98	> 500	369.41	> 500
10	82.51	> 500	205.38	> 500	379.86	> 500

Table 13: The efficiency of BP when the one-bit representation was used. See the previous table for a more detailed explanation of the contents of the table.

	$\sigma = 1.0$		$\sigma = 0.5$		$\sigma = 0.1$	
N	SRN	SCN	SRN	SCN	SRN	SCN
1	0.72	0.30	1.00	0.42	6.28	1.00
2	19.69	30.59	17.93	42.42	135.90	218.94
3	53.10	40.54	26.40	51.36	149.09	216.78
4	71.75	39.83	68.56	52.64	247.48	231.85
5	93.03	47.53	69.26	53.14	253.36	236.87
6	98.71	46.14	70.83	54.24	262.87	238.68
7	106.75	54.63	74.73	55.37	263.28	239.97
8	114.89	75.86	78.58	62.69	263.72	242.15
9	134.17	111.85	85.51	60.46	264.43	243.67
10	151.29	191.98	96.83	72.85	266.12	245.63

Table 14: Efficiency of EA for different settings of the mutation rate σ. The table shows number of evaluations needed, in thousands, before a successful network, solving length N, was found. The number of evaluations per generations were 100, since this was the population size used. Therefore the worst case number of evaluations is $2000 * 10^3$, i.e. 20 000 generations, which was the maximum number of generations, multiplied with 100, the population size. The results are based on all 100 experiments, not only the successful ones.

5.4 Consistency

The consistency measures how well, or bad, the training algorithm is at remembering previously successful solutions. A good training algorithm should not "throw" away good solutions once they are found, but instead use the previous solutions as a basis for further training.

Definition 3 The *consistency* of one training of one network for a length N is the percentage of the following epochs or generations, after the first solution for length N was found, when the algorithm has a solution still successful for at least lengths up to N.

That means that, if a training algorithm has 100% consistency for length N, that algorithm never "forgets" a network solving length N, once it is found. Table 15, 16 and 17 shows the resulting average consistency for the different experiments. In addition to the consistency, the *average preservation time* is measured to gain a deeper and more detailed understanding of the consistency of the algorithms.

Definition 4 The *average preservation time* for N, of an algorithm training a network, is the average number of epochs or generations which the training algorithm still holds a network which is successful for all string lengths at least up to N, after a solution for N was found.

The least possible preservation time is then of course one epoch or generation, which would mean that the algorithm instantaneously forgets the solution after it is found. Table 18, 19 and 20 show the resulting average preservation time, based on the average of all results, for the different experiments.

A third parameter is also measured to analyse how much the training algorithm oscillates. This is simply measured by counting how many times a solution for length N is rediscovered. Table 21, 22 and 23 contains the result based on the number of rediscoveries of solutions.

5.4.1 BP

SRN vs. SCN

From table 15 it can be concluded that the consistency of the SCN is often higher than the SRN, when the two-bit representation is used, except for $N = 1$. Of course this comparison is vague since the reliability of the SCNs is so low, but for most successful lengths of the SCNs, this appears to be the case.

If we look at table 18 and 19 we see that the average preservation time is in general higher for the SCN than the SRN for small values of N. This is the case for both types of representation used, although the consistency is worse for the SCN. This indicates that the successfulness of the SCNs during training does not oscillate as much as for the SRN. The results in table 21 and 22 which verifies this.

The influence of η

From table 15, where the results for the two-bit representation is shown, we can see that, except for $N = 1$, the consistency of the SRN training is increased for lower values of η. This behaviour is counter-intuitive since a higher learning rate should increase the risk for a solution to be forgotten by BP.

The consistency appears to be influenced by η. But the influence of η is widely different between the two architectures, representations and evaluation sets.

One-bit vs. two-bit representation

When the evaluation set Φ_3^{10} is used, the consistency and average preservation time are both considerably higher for the two-bit representation. For Φ_1^{10}, the situation is generally the same, but less distinct.

These results indicates that, in most cases, the two-bit representation helped BP to not "forget" the solutions which were already found.

5.4.2 EA

SRN vs. SCN

If only $N = 10$ is considered in table 17, it can be concluded that the consistency for the training of the SRN is about the same as for the SCN except when $\sigma = 1.0$, then the consistency was lower for the SCN.

No significant difference can be found in terms of the average preservation time in table 20. The number of rediscoveries in table 23 varies between the two architectures but the difference depends heavily on the value of σ.

The influence of σ

From the last row in table 17, it can be concluded that, when considering only successful networks for $N = 10$, higher consistency was gained for lower values of the mutation parameter. The number of rediscoveries were generally lower with for lower values of σ.

The influence of N

For some reason, the consistency was considerably lower in some of the experiments when $N = 2$. This phenomenon would perhaps not be seen if the results in table 17 only had been based on the simulations which lead to successful networks for $N = 10$

The average preservation time appears to be higher for the shortest and longest strings, while lower for the middle lengths.

5.4.3 EA vs. BP

The comparison of EA and BP is based on the results from the result in tables 15-23 and because the large amount of information it is hard to get a good overview of the results. However, it is clear from table 15, 16 and 17 that the consistency of the EA was considerably higher for all values of N.

From table 18, 19 and 20 it can also be concluded that the EA tended to not forget a solution once it was found. BP often forgot solutions instantaneously for higher values of N while the EA remembered solutions for at least several thousands generations.

	Evaluated on Φ_3^{10}					
	$\eta = 0.3$		$\eta = 0.1$		$\eta = 0.05$	
N	SRN	SCN	SRN	SCN	SRN	SCN
1	60.2352	15.2209	53.3551	57.7583	36.3950	68.0886
2	0.6296	2.0937	0.9335	7.3360	2.7083	12.9664
3	0.0551	0.2171	0.2401	0.0087	0.5503	1.0439
4	0.0076	0.0013	1.4180	0.0047	1.0595	1.0196
5	0.0005	—	1.0671	0.0012	0.6074	—
6	0.0002	—	0.0221	—	0.0238	—
7	0.0002	—	0.0091	—	0.0114	—
8	—	—	0.0016	—	0.0024	—
9	—	—	0.0009	—	0.0012	—
10	—	—	0.0005	—	0.0011	—
	Evaluated on Φ_1^{10}					
	$\eta = 0.3$		$\eta = 0.1$		$\eta = 0.05$	
N	SRN	SCN	SRN	SCN	SRN	SCN
1	86.0099	7.4898	75.1200	57.2144	59.4759	61.0760
2	1.2872	0.5644	5.0761	7.8370	8.4145	13.0958
3	0.3517	0.4719	1.9553	2.5955	2.8466	0.1493
4	0.1735	0.0302	6.0638	0.0159	6.6441	0.1301
5	0.0177	0.0036	3.3563	0.0026	3.7104	0.0432
6	0.0069	—	0.3595	0.0003	0.3560	0.0017
7	0.0019	—	0.1544	—	0.1641	—
8	0.0003	—	0.0395	—	0.0292	—
9	—	—	0.0151	—	0.0130	—
10	—	—	0.0054	—	0.0046	—

Table 15: Consistency of BP when the two-bit representation was used. The values are shown in percent and are calculated according to definition 3. The values reflect how many percent of the epochs contained successful solutions after the first solution was found.

	Evaluated on Φ_3^{10}					
	$\eta = 0.3$		$\eta = 0.1$		$\eta = 0.05$	
N	SRN	SCN	SRN	SCN	SRN	SCN
1	54.5862	12.3665	30.8116	10.4669	17.2882	16.4581
2	2.9675	0.7123	7.1976	0.4352	4.1357	3.9024
3	0.4109	0.5716	1.5062	0.0015	0.9622	0.0057
4	0.1702	—	0.5506	0.0005	0.0286	—
5	0.0686	—	0.2818	—	0.0077	—
6	0.0103	—	0.0228	—	0.0105	—
7	0.0043	—	0.0082	—	0.0059	—
8	0.0010	—	0.0016	—	—	—
9	0.0005	—	0.0008	—	—	—
10	—	—	—	—	—	—
	Evaluated on Φ_1^{10}					
	$\eta = 0.3$		$\eta = 0.1$		$\eta = 0.05$	
N	SRN	SCN	SRN	SCN	SRN	SCN
1	79.8768	15.9333	57.1130	16.8671	40.5985	14.8033
2	12.0232	1.9570	23.1813	1.6059	26.7099	8.8450
3	2.1297	0.0327	8.4312	0.0684	12.1099	—
4	0.4599	—	5.6742	0.1923	2.6631	—
5	0.1081	—	2.5641	0.0634	0.5635	—
6	0.0306	—	0.2925	0.0029	0.1139	—
7	0.0100	—	0.1357	0.0003	0.0450	—
8	0.0029	—	0.0123	—	0.0123	—
9	0.0011	—	0.0058	—	0.0087	—
10	0.0004	—	0.0011	—	0.0045	—

Table 16: Consistency, according to definition 3, of BP when the one-bit representation was used.

	$\sigma = 1.0$		$\sigma = 0.5$		$\sigma = 0.1$	
N	SRN	SCN	SRN	SCN	SRN	SCN
1	99.99	99.99	99.98	99.99	98.81	99.99
2	95.98	93.16	99.94	97.27	94.62	99.78
3	99.55	96.07	99.78	99.93	99.72	99.76
4	99.16	98.16	98.70	99.89	99.87	99.74
5	99.07	98.22	99.65	99.79	99.51	99.91
6	99.10	98.55	99.62	99.64	99.93	99.93
7	99.13	98.55	99.63	99.49	99.91	99.93
8	98.84	98.53	99.52	98.96	99.91	99.91
9	97.71	97.40	99.57	99.57	99.92	99.92
10	97.28	94.14	99.63	99.55	99.81	99.90

Table 17: Consistency, according to definition 3, of EA for different settings of the mutation parameter σ.

	Evaluated on Φ_3^{10}					
	$\eta = 0.3$		$\eta = 0.1$		$\eta = 0.05$	
N	SRN	SCN	SRN	SCN	SRN	SCN
1	3.98	1759.92	8.77	1927.68	13.24	1109.84
2	1.70	4.90	1.11	8.94	1.10	7.08
3	1.05	2.29	1.08	1.85	1.08	1.40
4	1.01	1.00	1.57	1.28	1.17	1.39
5	1.00	——	1.62	1.00	1.14	——
6	1.00	——	1.03	——	1.03	——
7	1.00	——	1.02	——	1.03	——
8	——	——	1.00	——	1.01	——
9	——	——	1.00	——	1.00	——
10	——	——	1.00	——	1.00	——
	Evaluated on Φ_1^{10}					
	$\eta = 0.3$		$\eta = 0.1$		$\eta = 0.05$	
N	SRN	SCN	SRN	SCN	SRN	SCN
1	35.15490	529.314	32.32	2626.32	22.87	3322.96
2	1.61	2.82	1.41	10.97	1.45	5.87
3	1.20	2.73	1.22	2.72	1.20	1.60
4	1.06	1.05	1.51	1.67	1.34	1.57
5	1.02	1.25	1.60	1.00	1.36	1.07
6	1.02	——	1.16	1.00	1.09	1.00
7	1.16	——	1.12	——	1.09	——
8	1.00	——	1.06	——	1.07	——
9	——	——	1.03	——	1.06	——
10	——	——	1.01	——	1.04	——

Table 18: Average preservation time, according to definition 4, of BP when the two-bit representation was used. The values are the average number of epochs for which a solution was remembered by the algorithm before it was forgotten again.

	Evaluated on Φ_3^{10}					
	$\eta = 0.3$		$\eta = 0.1$		$\eta = 0.05$	
N	SRN	SCN	SRN	SCN	SRN	SCN
1	3.41	65.17	4.99	238.37	6.73	195.91
2	1.38	4.89	1.16	2.90	1.07	4.32
3	1.12	1.69	1.07	2.25	1.03	1.00
4	1.06	—	1.05	1.25	1.01	—
5	1.04	—	1.03	—	1.00	—
6	1.03	—	1.02	—	1.00	—
7	1.01	—	1.01	—	1.00	—
8	1.00	—	1.00	—	—	—
9	1.00	—	1.00	—	—	—
10	—	—	—	—	—	—
	Evaluated on Φ_1^{10}					
	$\eta = 0.3$		$\eta = 0.1$		$\eta = 0.05$	
N	SRN	SCN	SRN	SCN	SRN	SCN
1	20.18	284.01	7.62	598.75	17.84	301.79
2	1.79	7.02	1.70	3.29	1.69	4.86
3	1.23	1.33	1.26	1.23	1.28	—
4	1.14	—	1.19	1.26	1.10	—
5	1.08	—	1.15	1.04	1.04	—
6	1.05	—	1.05	1.00	1.01	—
7	1.03	—	1.04	1.00	1.01	—
8	1.01	—	1.02	—	1.00	—
9	1.00	—	1.01	—	1.01	—
10	1.00	—	1.00	—	1.00	—

Table 19: Average preservation time, according to definition 4, of BP when the one-bit representation was used.

	$\sigma = 1.0$		$\sigma = 0.5$		$\sigma = 0.1$	
N	SRN	SCN	SRN	SCN	SRN	SCN
1	16 496	17 427	17 095	18 481	16 147	19 435
2	8 513	8 794	10 385	12 105	13 890	13 378
3	4 446	6 803	4 596	10 077	8 240	12 098
4	5 525	6 900	5 984	10 727	4 004	9 283
5	5 532	6 810	7 884	9 521	4 804	8 159
6	6 292	7 127	8 644	9 227	10 368	10 816
7	6 459	8 730	9 020	10 272	11 328	10 977
8	7 233	9 470	9 482	11 053	11 701	11 953
9	7 138	9 539	10 259	11 116	12 340	12 846
10	10 645	11 857	12 453	12 781	13 018	13 114

Table 20: Average preservation time, according to definition 4, of EA for different values of σ. The values are the average number of generations that a solution was remembered by the algorithm before it was forgotten again.

	Evaluated on Φ_3^{10}					
	$\eta = 0.3$		$\eta = 0.1$		$\eta = 0.05$	
N	SRN	SCN	SRN	SCN	SRN	SCN
1	92 8330	1 0995	84 2243	3 3157	59 8996	2 6562
2	1 8901	3794	2 2330	1 7918	7 2828	1 4261
3	2346	2024	6650	144	1 7664	8070
4	345	10	4 0903	115	3 4769	7993
5	23	—	2 9287	40	1 9478	—
6	10	—	948	—	806	—
7	10	—	377	—	369	—
8	—	—	67	—	73	—
9	—	—	40	—	34	—
10	—	—	20	—	27	—
	Evaluated on Φ_1^{10}					
	$\eta = 03$		$\eta = 01$		$\eta = 005$	
N	SRN	SCN	SRN	SCN	SRN	SCN
1	11 9623	2 8454	18 8851	2 0761	26 6209	1 5623
2	2 0736	5008	8 9144	2 0180	14 2738	1 6732
3	7929	1841	5 1044	1507	7 8160	1552
4	4825	398	17 4316	335	20 2293	1318
5	517	45	9 452	95	10 7784	185
6	198	—	1 3227	10	1 2841	10
7	47	—	5819	—	5767	—
8	10	—	1559	—	1016	—
9	—	—	603	—	450	—
10	—	—	220	—	156	—

Table 21: The number of rediscoveries of solutions for each length when the two-bit representation was used.

	Evaluated on Φ_3^{10}					
	$\eta = 0.3$		$\eta = 0.1$		$\eta = 0.05$	
N	SRN	SCN	SRN	SCN	SRN	SCN
1	85 5104	1 7766	72 6627	1 4618	44 2755	1 5443
2	9 3908	7086	24 1941	6017	13 3400	3 2229
3	1 5691	2259	5 8154	20	3 3938	90
4	6871	—	1 9407	15	659	—
5	2690	—	8752	—	126	—
6	418	—	645	—	36	—
7	176	—	216	—	27	—
8	39	—	16	—	—	—
9	20	—	10	—	—	—
10	—	—	—	—	—	—
	Evaluated on Φ_1^{10}					
	$\eta = 0.3$		$\eta = 0.1$		$\eta = 0.05$	
N	SRN	SCN	SRN	SCN	SRN	SCN
1	27 9863	1 2170	56 6612	3 6493	57 4462	9882
2	23 9894	9036	45 1643	1 2919	46 6202	3 3751
3	7 1842	984	25 9078	1444	30 8170	—
4	1 8152	—	18 4123	4210	5 9124	—
5	4406	—	8 3310	1645	1 2349	—
6	1307	—	1 459	70	2460	—
7	430	—	4615	10	917	—
8	127	—	404	—	185	—
9	49	—	177	—	108	—
10	18	—	26	—	45	—

Table 22: The number of rediscoveries of solutions for each length when the one-bit representation was used.

	$\sigma = 1.0$		$\sigma = 0.5$		$\sigma = 0.1$	
N	SRN	SCN	SRN	SCN	SRN	SCN
1	164	136	148	124	170	107
2	1 383	1 790	593	632	1 336	223
3	3 024	2 783	1 756	645	548	294
4	4 477	6 163	2 195	997	851	438
5	5 757	10 555	2 642	1 472	1 019	580
6	6 057	7 237	2 788	2 008	832	677
7	6 317	7 567	2 901	2 526	854	602
8	5 940	7 898	2 809	6 502	887	780
9	3 513	9 097	2 239	2 297	848	605
10	3 420	9 909	1 519	1 370	1 358	505

Table 23: The number of rediscoveries of solutions for each length for the EA.

5.5 Generalization

The generalization capacity of the networks was tested using the same method as used for evaluation of the correctness of the networks described in section 4.2. In the test of generalization capacity the networks were tested on Φ^{50}_{1000}, i.e. a set of randomly ordered A^nB^n-strings with $1 \leq n \leq 50$ and 1000 instances of each string length. The proportion of correctly classified strings was measured for each n individually.

Φ^{50}_{1000} gave worse quality for $1 \leq N \leq 10$ than Φ^{10}_{1000} that was used to measure the quality in section 5.2. This is explained by that the initial context activation, before the first A is received, affects the orbit of the context nodes for the rest of the string. When the networks are tested on only the training set, the initial state, which is determined by the last state of the previous string, is returned to a value which the network can handle better.

Due to the massive amount of data from the experiments, the results have in some cases been merged. That means that, e.g. the effect of the different representations cannot be read in the diagrams.

Figure 16 and 17 shows the average and maximum generalization capacity of all successfully trained SRNs by BP. Figure 18 and 19 shows the same results for the SRNs trained with EA and figure 20 and 21 the same results for the SCN training with EA.

5.5.1 BP

The only parameter that has been expressed in figure 16 is the learning rate η. But there is no clear effect of η on the generalization abilities of the SRNs.

5.5.2 EA

SRN vs. SCN

From figure 18, 19, 20 and 21 The SRNs appears to be slightly better at generalizing than the SCNs. When looking on the maximum generalization ability in figure 19 and 21 we see that the SRN correctly predicted 100% at maximum for N up to 21 and the SCN only up to 14.

The influence of σ

Lower values of σ gave better generalization abilities for the SRNs. It is difficult to see whether σ had any influence on the SCNs.

5.5.3 EA vs. BP

Figures 16 to 19 show that the SRNs trained by the EA were far better on generalizing than those trained by BP.

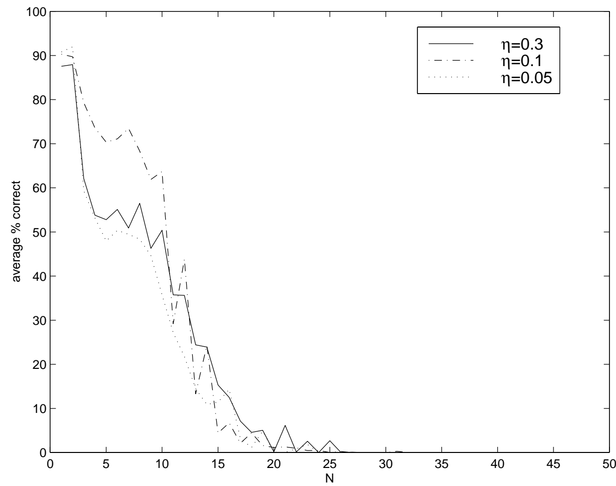

Figure 16: The average generalization capacity of the successful SRNs trained by BP. The networks were tested on 1000 strings of each length in random order. The vertical axis is the percentage of the strings that were successfully predicted for the N.

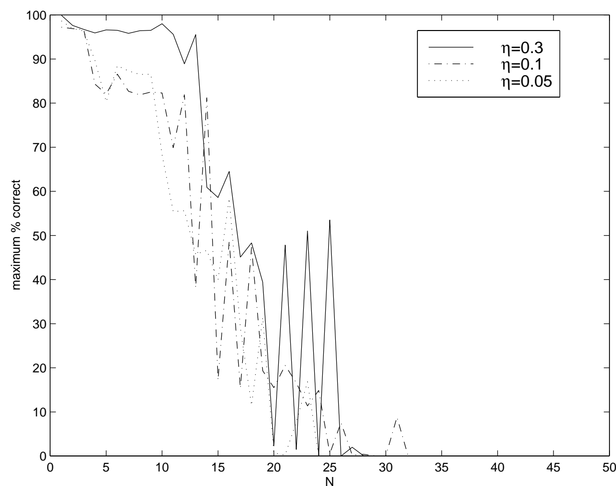

Figure 17: The maximum generalization capacity of the successful SRNs trained by BP.

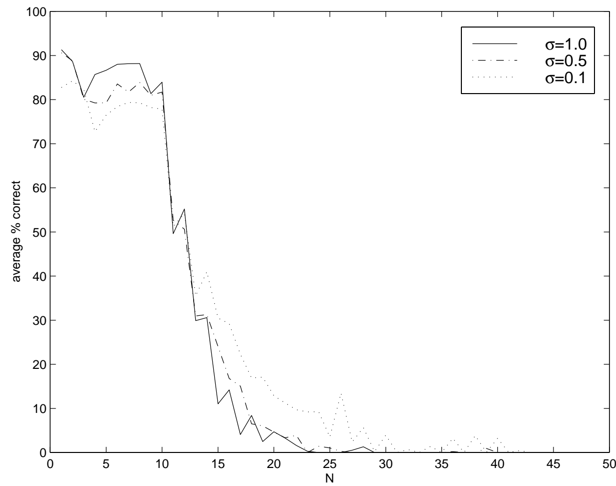

Figure 18: The average generalization capacity of the successful SRNs trained by EA. The networks were tested on 1000 strings of each length in random order. The vertical axis is the percentage of the strings that were successfully predicted for the N.

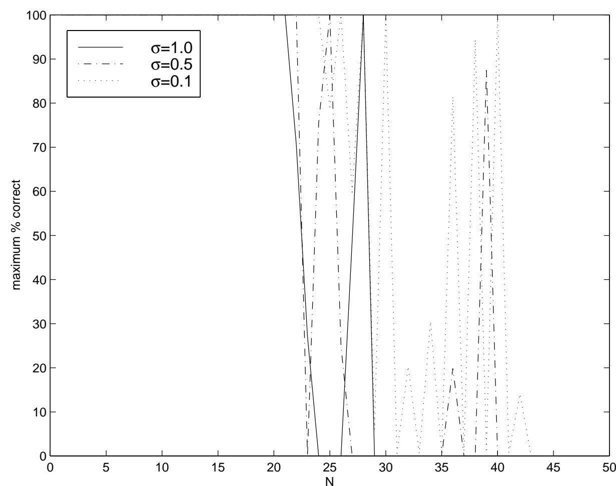

Figure 19: The maximum generalization capacity of the successful SRNs trained by EA.

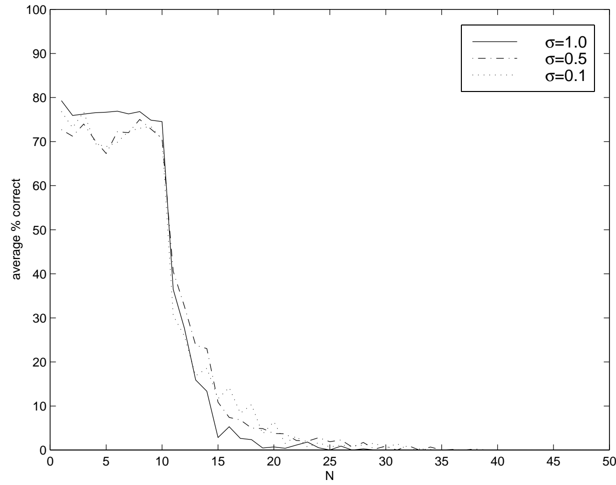

Figure 20: The average generalization capacity of the successful SCNs trained by EA.

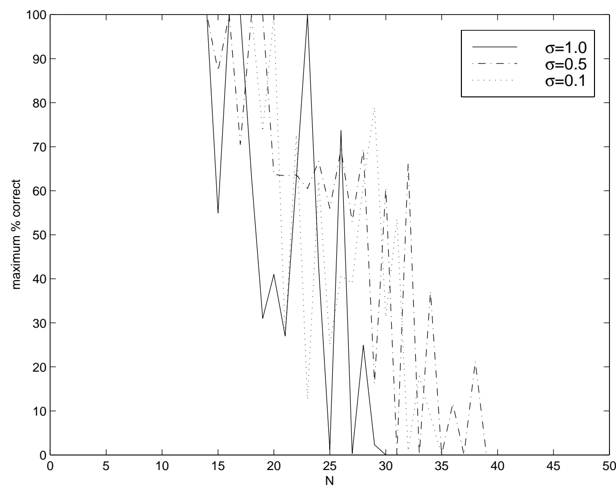

Figure 21: The maximum generalization capacity of the successful SCNs trained by EA.

5.6 The distribution of signatures

In section 3.5 a classification scheme of the networks was described using the self-derivatives of the activation in the context nodes. Table 2 describes the SRN signatures briefly and table 3 the signatures of the SCNs.

To gain a deeper understanding of what kind of solutions are found by BP and EA, the distribution of the different signatures of networks was measured. It was measured both for the successful networks (if any) and all the networks which was present at the end of the training in order to see both which signatures that the training algorithm could make successful and which it preferred to "exploit".

The distributions for different settings of the η-value, different representation and different evaluation set have all been merged into one set of results for BP. This was done because few of the BP experiments were successful.

In order to know the "normal" distribution of signatures, the signature distribution of randomly initialized networks was also measured. 10 000 random networks were generated for each network type, with weights randomly distributed in the interval [-1,1].

Table 24 and 25 show the distributions of the networks trained with BP, for the SRN and SCN respectively. Table 24 includes both the distribution of the successful and final networks while table 25 only shows the distribution of the final networks, since no successful SCNs were found with BP.

Table 26 and 27 show the signature distribution of the final and successful networks of the EA. Figure 23 and 25 shows how the distribution of signatures changed during the training with EA.

In order to measure the quality of the different signatures figure 22, 24 and 26

shows the generalization capacity of each individual signature. See section 5.5 for an explanation of how the generalization capacity are calculated.

5.6.1 BP

SRN

It is clear from table 24 that BP was biased towards exploiting monotonic solutions with signature $(2, 0)$. It is also clear, however, that it was completely unsuccessful in finding *successful* monotonic solutions. The solutions were biased towards oscillating solutions with signature $(0, 2)$, but there were a few exotic solutions as well. This agrees with the results in [TBW99] and [RWE99] where BPTT also seemed biased towards the oscillating solutions. BPTT was however successful in finding some monotonic solutions while the BP tested here is not. Ironically, the distribution of the final networks reveals that BP was biased towards exploring monotonic networks while at the same time, not being able to find solutions for this type of network. It appears as if once an "oscillating path" in the search space was followed, the probability of finding a solution increased significantly for BP. But also that these "oscillating paths" were harder to find than the "monotonic paths". If the distribution of the successful networks from the EA is compared to those from BP we find some similarities, the EA seemed biased towards finding the same type solutions.

In figure 22 it can be seen that the networks with signature $(0, 2)$, i.e. the oscillating, proved to perform with a more monotonic decreasing reliability for higher values of N than signature $(1, 1)$. This can probably be explained by the fact that signature $(1, 1)$ was less common and that the results therefore are based on fewer networks which causes the average data to be less "smoothened".

SCN

The final distribution in table 25 reveals that, compared to the EA, BP focuses the search on signatures which were less successful in solving the $A^n B^n$-language. Take for example signature $(2, 0, 0, 2, 1)$, this signature is never present in any successful networks trained by the EA while a substantial part of the final networks from BP are of this type. Also, the distributions of signatures $(1, 1, 1, 1, 0)$, $(0, 2, 1, 1, 1)$ and $(0, 2, 0, 2, 0)$ shows that BP seems to focus the search on the wrong types of networks. For these three signatures the distribution is significantly reduced by BP when compared to the initial distribution from which it started. The EA effects the distribution in an opposite direction and is clearly more successful than BP, perhaps because it is able to take advantage of these signatures.

Signature	Initial	Final	Successful	Succ. EA
$(2, 0)$	25.15	48.75	0.00	10.41
$(1, 1)$	49.61	25.50	6.60	15.38
$(0, 2)$	25.24	25.75	93.40	74.21

Table 24: The distribution of signatures of the SRN of all final networks and the successful networks trained by BP. By final network we refer to those after the last epoch. For comparison, the last column shows the distribution of the successful networks of all EA-simulations

Signature	Initial	Final	Succ. EA
$(2,0,2,0,0)$	14.19	3.42	8.55
$(2,0,1,1,1)$	9.15	18.67	3.42
$(2,0,0,2,1)$	1.68	28.56	0.00
$(1,1,2,0,1)$	9.15	0.55	5.56
$(1,1,1,1,0)$	27.84	3.33	29.06
$(1,1,1,1,1)$	3.19	9.70	1.71
$(1,1,0,2,1)$	9.78	26.99	9.40
$(0,2,2,0,1)$	1.63	0.00	0.00
$(0,2,1,1,1)$	9.56	0.28	14.10
$(0,2,0,2,0)$	13.83	8.50	28.21

Table 25: The distribution of signatures of the SCN in the final networks from BP. By final network we refer to those after the last epoch. The last column shows the distribution of the successful networks of all EA-simulations.

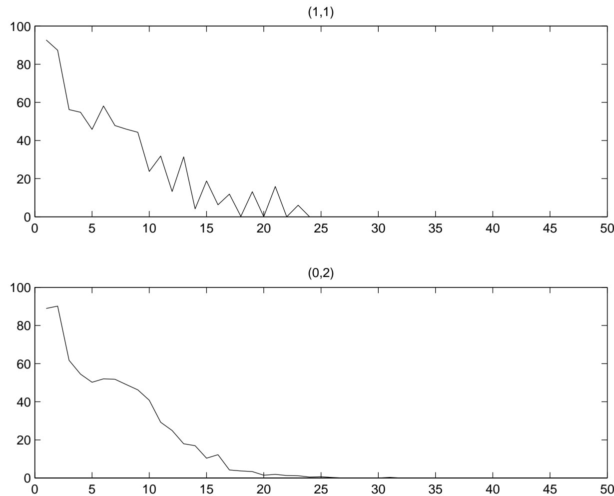

Figure 22: The generalization capacity of SRNs, trained by BP, with different signatures. These results are based on the average of the successful networks from all BP experiments. The signature (2,0) has been left out since no successful networks with this signature were found by BP.

5.6.2 EA

SRN

From table 26 it is clear that the EA, is biased towards finding SRNs with signature $(0,2)$, i.e. oscillating. This results matches the results of an evolutionary hill-climbing algorithm used in [TBW99], their algorithm also found more networks with signature $(0,2)$.

If table 26 is analysed in more detail we find that σ affects the distribution of the final networks more than the distribution of successful networks. What can be concluded from the influence of σ is probably that a signature, more common for lower values of σ, are situated in the search-space, such that the search algorithm required small, stepwise adjustments in order to not "miss" the signature.

From figure 23 we can conclude that the EA affects the distribution only very early in the evolving process. After the initial distribution had been changed, it remained virtually fixed during the rest of the training.

Figure 24 reveals that the signature $(2,0)$ did not generalize very well to longer strings than those in the training set. Already for $N = 11$, the percentage of correctly predicted strings was reduced significantly and from $N \geq 15$ the the networks with this signature are unable to correctly predict any strings. The networks with signatures $(1,1)$ and $(0,2)$ did however generalize better which indicates that being able to oscillate is important for generalization. This was also found in [RWE99].

SCN

In table 27 we can see the signature distribution of the final and successful SCN trained with EA. The influence of σ is not as easy to analyse as it were for the SRN where

the influence of the final distributions of signature were monotonically increasing or decreasing with σ. For some of the signatures, $\sigma = 0.5$, appears as the most promoting of the three values of σ, these signatures are $(1,1,1,1,0)$, $(0,2,2,0,1)$ and $(0,2,1,1,1)$. For two other signatures, $\sigma = 0.5$ oppresses their presence in the final distribution, these are $(2,0,2,0,0)$, $(1,1,2,0,1)$. The other signatures are positively or negatively affected by the value of σ.

If signature $(1,1,0,2,1)$ is examined for $\sigma = 0.1$ we find it was 0.00% of the final distribution but 8.57% of the successful distributions. This is quite a contradiction at a first glance, because how could there be any successful networks of this signature, when there were none among the final networks?. This is however not impossible, the results indicates that the successful networks with signature $(1,1,0,2,1)$ were all found near the beginning of the training where after the signature shifted into another before the end of all 20 000 generations.

As figure 25 indicates, the distribution of signatures stabilises early in the training. This was also found for the SRN.

In figure 26 we find that the signatures of the SCNs can be divided into two groups in terms of generalization. One group performed well on the training set, i.e. with $1 \leq N \leq 10$, while at the same time performing very badly on longer strings. The other group of signatures had a worse performance than the first group on the training set but were at the same time better at generalising to longer strings. All successful networks' signatures that were monotonic for either A or B as input are in the first group while those with an oscillating behaviour ended up in the other group.

Signature	Initial	$\sigma = 1.0$		$\sigma = 0.5$		$\sigma = 0.1$	
$(2, 0)$	25.15	8.55	(8.42)	10.00	(10.53)	10.98	(16.13)
$(1, 1)$	49.61	15.20	(12.63)	20.88	(18.95)	39.67	(12.90)
$(0, 2)$	25.24	76.25	(78.95)	69.12	(70.53)	49.35	(70.97)

Table 26: The distribution of signatures of the SRN in the resulting populations of the EA.The initial distribution was calculated by randomly generating 10000 networks. The numbers within parentheses are calculated by including the signatures of the successful networks.

Signature	Initial	$\sigma = 1.0$		$\sigma = 0.5$		$\sigma = 0.1$	
$(2, 0, 2, 0, 0)$	14.19	9.80	(9.30)	3.88	(5.13)	9.45	(11.43)
$(2, 0, 1, 1, 1)$	9.15	3.86	(6.98)	2.95	(2.56)	0.84	(0.00)
$(2, 0, 0, 2, 1)$	1.68	0.04	(0.00)	0.00	(0.00)	0.00	(0.00)
$(1, 1, 2, 0, 1)$	9.15	7.21	(3.49)	6.29	(5.13)	7.95	(8.57)
$(1, 1, 1, 1, 0)$	27.84	24.91	(25.58)	31.63	(34.62)	28.81	(27.14)
$(1, 1, 1, 1, 1)$	3.19	1.65	(0.00)	2.02	(1.28)	5.09	(4.29)
$(1, 1, 0, 2, 1)$	9.78	13.46	(12.79)	2.02	(6.41)	0.00	(8.57)
$(0, 2, 2, 0, 1)$	1.63	0.40	(0.00)	1.76	(0.00)	0.00	(0.00)
$(0, 2, 1, 1, 1)$	9.56	14.17	(11.63)	17.95	(20.51)	10.80	(10.00)
$(0, 2, 0, 2, 0)$	13.83	24.50	(30.23)	24.66	(24.36)	25.03	(30.00)

Table 27: The distribution of signatures of the SRN in the resulting populations of the EA. The initial distribution was calculated by randomly generating 10000 networks. The numbers within parentheses are calculated by only looking on the signature of the successful networks.

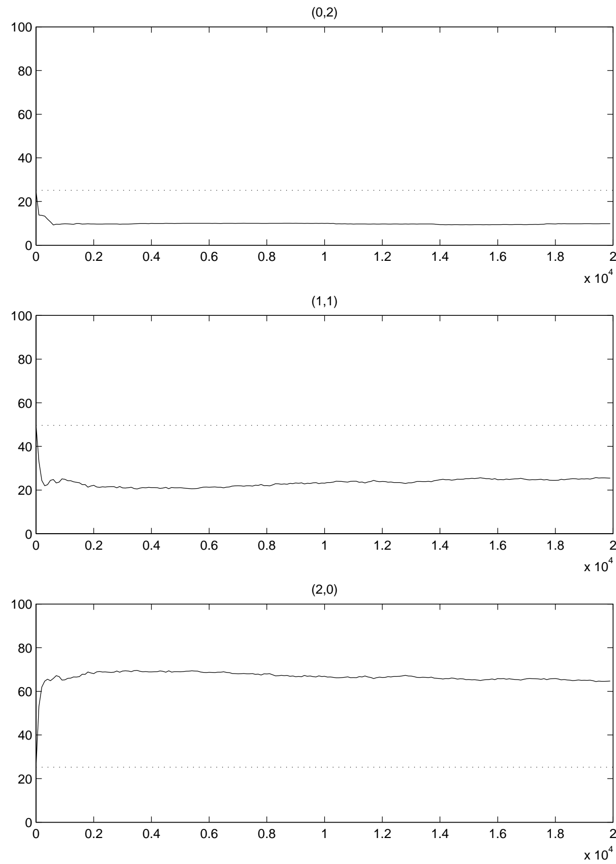

Figure 23: The distribution of different signatures in the EA-population for the SRN plotted over the generations. The dotted lines represent the distribution of the, randomly generated, initial populations.

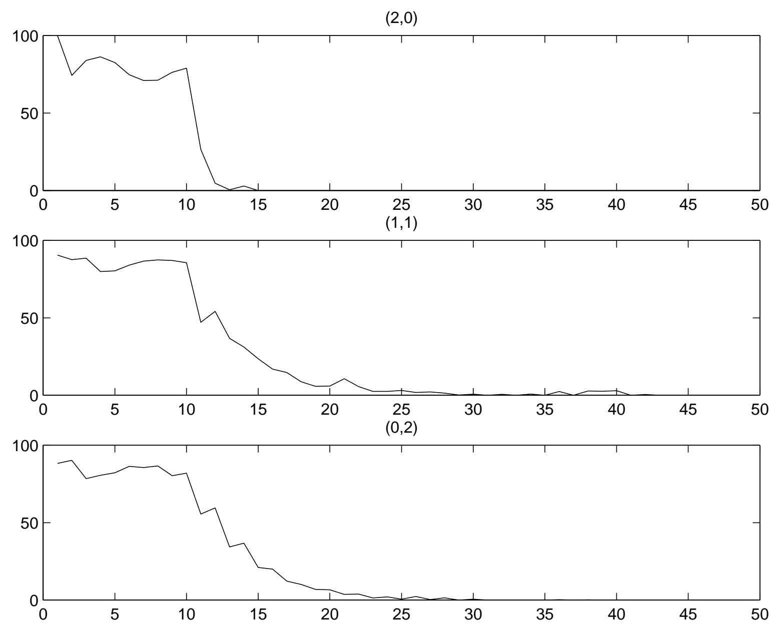

Figure 24: The generalization capacity of SRNs, trained by the EA, with different signatures. These results are based on the average of all successful networks of the three EA experiments.

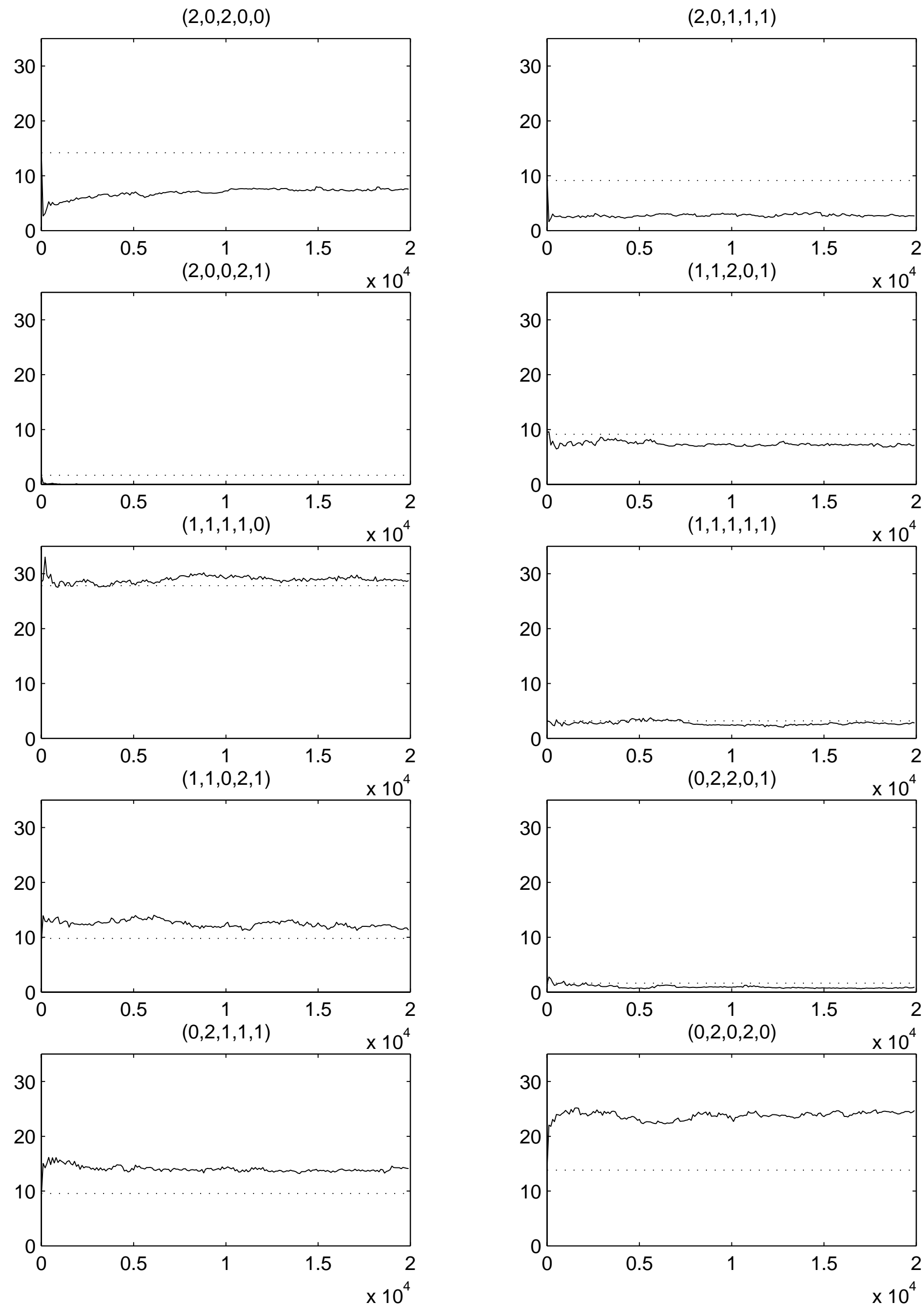

Figure 25: The distribution of different signatures in the EA-population for the SCN. The dotted lines represent the initial distribution of signatures.

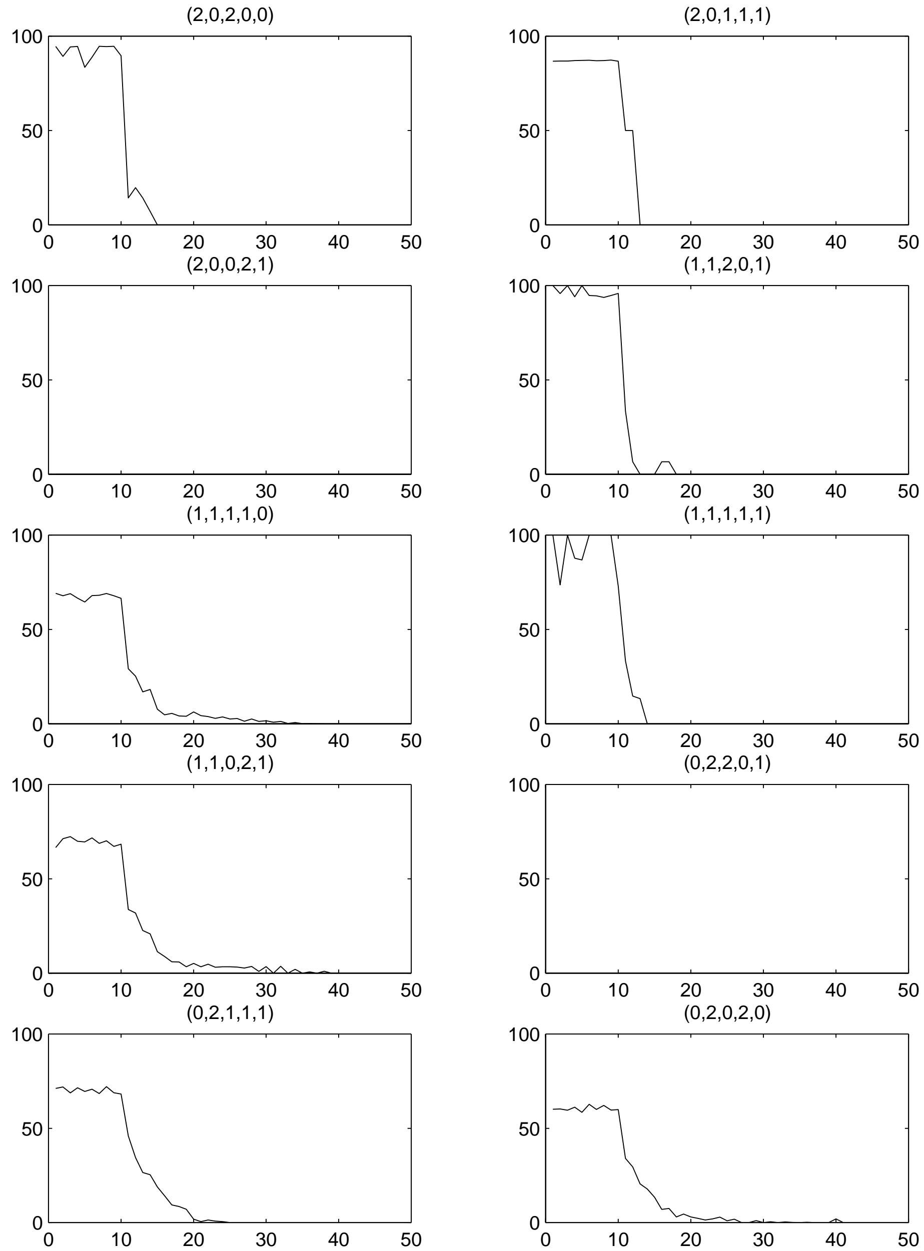

Figure 26: The generalization capacity of SCNs, trained by the EA, with different signatures. These results are based on the average of all successful networks of the three EA experiments. No successful networks with signatures $(2, 0, 0, 2, 1)$ or $(0, 2, 2, 0, 1)$ were found so therefore the corresponding diagrams are empty.

Chapter 6

Conclusions

The high-level goal of this project has been to map the differences between first- and second-order recurrent neural networks. The comparison was limited to simple instances of these network types. The aim was focused on describing and analysing how, and how well, the networks solved the problem after training. The training of the network was also part of the analysis.

The analysis in chapter 3 showed that SCNs have a bigger repertoire of dynamical behaviours in how to solve language problems than the SRNs have. This is in line with the results in [MG93] and [GGCC94] that showed that, given some constraints, the SCN can solve problems which the SRN can not. The results of this dissertation have shown that the SCN can for example interpret and process its own internal state in completely different ways, depending on the input. The SRN are always bound to interpret its state the same way. However, the input does, in a way, influence how the state of the SRN is transformed into the next state by implementing something like a dynamic bias to the context node. This influence is limited however, since the signs of the self-derivatives cannot be changed at runtime. If a self-derivative of one

context node is positive for one input instance it must be positive for all possible input instances. The SCN can change its dynamical behaviour by changing the sign of the self-derivative of the internal activation. The internal activation at any time can also be differently interpreted depending on the input, something which is demonstrated by the input-dependent hyperplanes of the SCN.

The results from the experiments indicated that those solutions that were possible only for the SCN, were not fully realized when the networks were trained on the A^nB^n-grammar. The analysis using signatures, i.e. a classification scheme based on the dynamical behaviours of the networks, showed that some types of SCN-signatures did not exist among the solutions, especially those that were fully monotonic for A and fully oscillating for B or vice versa. Obviously there was no use for these types of solutions within this domain.

Both BP and the EA performed better on the SRN than on the SCNs in terms of reliability, in most cases, which indicates that this type of network was easier to optimize. The SRNs were also slightly better on generalizing to longer strings than the SCNs, which also indicates the SRN was better in terms of training and performance. The bad results of the BP-training of the SCN compared to, e.g., [Pol91, MG93, HG95] can perhaps be explained by factors which could not be controlled. These results were however obtained for BP, which does not take into account the temporal structure of the language problem and it would be preferable to make this comparison with BPTT instead.

The use of signatures as a classification scheme allowed for different solutions to be automatically classified which helped to clarify what types of solutions were preferred by the training algorithms. For example, the results indicated that BP, which was

generally worse than the EA, was biased towards solutions which were deemed less successful by the EA[9]. The use of signatures also helped to identify other features of the different types of solutions, such as how well they generalized etc. It was found that the networks benefited from implementing oscillating solutions if the ability to generalize is considered. Such automatic classification of RNN solutions can be very helpful and more work should be conducted to refine this technique.

This dissertation has shown that, in spite of the theoretical proof of equivalence for general first- and second-order networks, the second order networks appear to have a higher computational power due to the bigger repertoire of dynamical behaviours. This is of course due to the fact that only restricted instances of first- and second-order networks have been regarded. It has also been shown that there are quantitative differences in the training of these networks in the CFL-problem which was tested but that these differences heavily depend on other features of the training algorithm than just the network type, e.g. learning/mutation rate etc.

6.1 Future work

More detailed analysis of the solutions would be interesting for a more complete understanding of the training of the networks, e.g. analysis of how the efficiency, consistency etc. are affected by the signature of the solution. In [TBW99] the training is configured such that it is biased towards solutions which are known to be more successful on the A^nB^n-language, i.e. oscillating solutions. Similar approaches could be taken on other problem types if the successfulness of different dynamical behaviours of the networks are mapped for the specific problem.

[9] By that the EA did not exploit these signatures.

More work could be done on the existing classification scheme of signatures. An analysis of how the signatures relate to each other, i.e. which signatures are "neighbours" in the search space. This analysis could then be compared to the "paths" of the training algorithms, i.e. how they pass through solutions corresponding to different signatures. This knowledge could then be used to optimize the training algorithms.

The signatures used in this dissertation are quite primitive. They only take into account the self-derivatives of the context nodes and the "cross"-weights are ignored. It is also difficult to generalize to networks with more layers or more context nodes since the self-derivatives may then have different signs depending on the state. Refinement of the classification techniques of dynamical behaviours is important since this classification may be very helpful in the understanding of the networks, the training of them and in an extension, the problem which the networks are solving.

Acknowledgments

I am the author of this dissertation. But that does not mean that I am the only one that has put an effort into this work. Several people have helped, directly or indirectly, to complete this dissertation. Even if mentioning a few of them means excluding others, who rightfully belong here, I would still like to explicitly show my gratefulness to the following persons.

I want to thank my supervisors, Mikael Bodén and Tom Ziemke, for giving me the initial idea and many helpful hints along the way. Their effort to keep me focused on the central issues held me back from attempting to solve more problems than would be possible during these few months.

I would also like to thank the people of my MSC-class this year who made this year endurable by a number of successful parties. We did have great fun, even when the work was laid upon us as a pile of garbage on the city dump. Thanks to each other, we endured.

I also want to thank my supporting parents. During these four years I have been studying in Skövde, they have help me many times with all sorts of things. Without their help, my time here would have been far more complicated. My brother and sister have also given me support, perhaps without knowing it themselves, just by being there.

And last, but not least, I would like to give a thousand thanks to the girl of my life, Louise Holm, who endured many lonely evenings when I worked on this dissertation. I am definitely in debt to her, in terms of time, and I do intend to pay her back.

Bibliography

[Bäc96] Thomas Bäck. Evolution strategies: An alternative evolution computation method. In J-M. Alliot, E. Lutton, E. Ronald, and M. Schoenauer, editors, *Proceedings of Artificial Evolution 1995*, pages 3–20. Springer-Verlag, 1996.

[Bul97] J.A. Bullinaria. Analyzing the internal representations of trained artificial neural networks. In A. Browne, editor, *Neural Network Analysis, Architectures and Applications*, pages 3–26. TOP Publishing, 1997.

[BWTB99] M. Bodén, J. Wiles, B. Tonkes, and A. Blair. Learning to predict a context-free language: Analysis of dynamics in recurrent hidden units. In *Proceeding of ICANN 99*, pages 359–364, Edinburgh, 1999. IEE.

[Dar72] C. Darwin. *The origin of species.* John Murray, London, UK, 1872.

[Dev92] R. L. Devanay. *A first course in chaotic dynamical systems.* Addison-Wesley, 1992.

[Elm90] J. L. Elman. Finding structure in time. *Cognitive Science*, 14:179–211, 1990.

[GGCC94] M. W. Goudreau, C. L. Giles, S. T. Chakradhar, and D. Cheng. First-order vs. second-order single layer recurrent neural networks. *IEEE Trans. on Neural Networks*, 5(3):511–518, 1994.

[Gol89] D.E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning.* Addison-Wesley, 1989.

[GSC+90] C. L. Giles, G. Z. Sun, H. H. Chen, Y. C. Lee, and D. Chen. Higher order recurrent networks and grammatical inference. In D. S. Touretzky, editor, *Advances in Neural Information Processing Systems*, volume 2. Morgan Kaufman, 1990.

[HB90] S. J. Hanson and D. J. Burr. What connectionist models learn: Learning and representation in connectionist networks. *Behavioral and Brain Sciences*, 13:471–518, 1990.

[HG95] B. G. Horne and C. L. Giles. An experimental comparison of recurrent neural networks. *Neural Information Processing Systems*, 7, 1995.

[Hol75] J.H. Holland. *Adaptation in Natural and Artificial Systems.* Univ. of Michigan Press, 1975.

[HSW90] K. Hornik, M. Stinchcombe, and H. White. Universal approximation of an unknown mapping and its derivatives using multilayer feedforward networks. *Neural Networks*, 3:551–560, 1990.

[KÑF98] S. C. Kremer, R. P. Ñeco, and M. L. Forcada. Constrained second-order recurrent networks for finite state automata induction. In L. Niklasson,

M. Bodén, and T. Ziemke, editors, *Proceedings of the 8th International Conference on Artificial Neural Networks*. Springer, 1998.

[Kol94] J. F. Kolen. *Exploring the Computational Capabilities of recurrent neural networks*. PhD thesis, The Ohio State University, Department of Computer and Information Sciences, 1994.

[Lip87] R. P. Lippman. An introduction to computing with neural nets. *IEEE Acoustics, Speech, and Signal Processing Magazine*, pages 4–22, April 1987.

[Mee96] L.A. Meeden. An incremental approach to developing intelligent neural network controllers for robots. *IEEE Transactions on Systems, Man, and Cybernetics, Part B: Cybernetics*, 26(3):474–85, 1996.

[MG93] C. B. Miller and C. L. Giles. Experimental comparison of the effect of order in recurrent neural networks. *International Journal of Pattern Recognition and Artificial Intelligence*, 7(4):849–872, 1993.

[Mit95] M. Mitchell. *An Introduction to Genetic Algorithms*. MIT Press, 1995.

[Moz89] M. Mozer. A focused back-propagation algorithm for temporal pattern recognition. Technical Report 88–3, Univ. of Toronto, 1989.

[MP69] M. Minsky and S. Papert. *Perceptrons*. MIT press, 1969.

[Pol86] J. B. Pollack. Cascaded back-propagation on dynamic connectionist networks. Technical Report MCCS–86–67, Computer Research Lab., Mexico State University, 1986.

[Pol90] J. B. Pollack. Language acquisition via strange automata. In *The Twelfth Annual Conference of the Cognitive Science Society*, pages 678–685. Lawrence Erlbaum Associates, Inc, 1990.

[Pol91] J. B. Pollack. The induction of dynamical recognizers. *Machine Learning*, 7:227–252, 1991.

[RHW86] D.E. Rumelhart, G.E. Hinton, and R.J. Williams. Learning internal representations by back-propagating errors. *Nature*, 323:533–536, 1986.

[RWE99] P. Rodriguez, J. Wiles, and J. L. Elman. A recurrent network that learns to count. *Connection science*, 11:5–40, 1999.

[Sie99] H. T. Siegelmann. *Neural Networks and Analog Computation, Beyond the Turing Limit.* Birkhäuser, 1999.

[SJ90] N. E. Sharkey and S. Jackson. An internal report for connectionists. In R. Sun and L. Bookman, editors, *Computational architectures integrating neural and symbolic processes*, pages 223–244. Kluwer Academic Press, Boston, USA, 1990.

[SS94] H. T. Siegelmann and E D. Sontag. Analog computation via neural networks. *Theoretical Computer Science*, 131:331–360, 1994.

[TBW99] Brad Tonkes, Alan Blair, and Janet Wiles. Inductive bias in context-free language learning. In *Proceedings of the Ninth Australian Conference on Neural Networks*, Brisbane, Australia, 1999.

[WE95] J. Wiles and J.L Elman. Learning to count without a counter: A case study of dynamics and activation landscapes in recurrent neural networks.

In *Proceedings of the Seventeenth Annual Conference of the Cognitive Science Society*, pages 482–487. Cambridge MA: MIT Press, 1995.

[Wer94] P. Werbos. Backpropagation through time: what it does and how to do it. *Neurocontrol*, 1994.

[Whi90] H. White. Connectionist nonparametric regression - multilayer feedforward networks can learn arbitrary mappings. *Neural Networks*, 3:535–549, 1990.

[Whi95] D. Whitley. Genetic algorithms and neural networks. In J. Periaux and G. Winter, editors, *Genetic Algorithms in Engineering and Computer Science*. John Wiley & Sons Ltd., 1995.

Appendix A

Artificial neural networks

An artificial neural network (ANN) is built up from two simple components; neural nodes and weighted connections between the nodes (see figure 27). As the name suggests, they are inspired by the neural system of animals, such as the brain of humans. The term “neurons” is however a bit misleading since the biological relevance of ANNs is quite limited. The operation of an ANN is determined by its architecture and weights. When a network is trained, the weights of the connections are adjusted to give the network its appropriate function. It has been shown mathematically that ANNs can approximate any continuous function to an arbitrary degree of accuracy given that it has enough hidden nodes in the network [Lip87, HSW90, Whi90]. The training is often *supervised* which means that the network is trained on a set of input and output examples, sampled from the domain we want the network to model. This set of examples is called the *training set*.

A.1 Feed forward networks

The most commonly used ANN, is the *multilayer perceptron*, also called *feed forward network*. A feed forward network is a set of fully interconnected layers of nodes. Layers between the input- and output-layer are called *hidden* layers.

Figure 27 describes a generalized picture of a feed-forward network. The basis of how the nodes and weights will be referred to in the rest of this material is also described, informally, in this figure.

Each node $X_{\ell,j}$, where ℓ is used as the index for the layers of nodes, has an activation $a_{\ell,j}$ which in the input layer is determined from an external source and in the following layers is determined by the activation of preceding layers and the weights of the connections between them. The weights are denoted $W^{\ell}_{i,i}$, i.e. a weight from node $X_{\ell,i}$ to node $X_{\ell+1,j}$. Each node (except the input nodes) also has a *bias* connection, $W^{\ell}_{b,i}$, which can be viewed as a weighted input connection from a constantly active node, i.e. with $a = 1.0$. The bias node is not shown in in figure 27 but instead considered as internal properties of the nodes. Sometimes the bias nodes are explicitly shown in the figures for the purpose of clarity.

The computation in the network is done by propagating the activation from layer to layer. First the activation of each input unit is set according to the input we want to process. In the next layers the activation is calculated by first calculating the *net*-value for each unit by summarizing the incoming activation:

$$net_{\ell,i} = W^{\ell}_{b,i} + \sum_{i=1}^{n_\ell} a_{\ell-1,i} W^{\ell-1}_{i,j} \tag{20}$$

Where $W^{\ell-1}_{i,j}$ corresponds to the weight between the nodes $X_{\ell-1,i}$ and $X_{\ell,j}$ in the

network of figure 27. The activation of the node is then calculated from the *net*-value by an activation function

$$a_{\ell,i} = \Gamma(net_{\ell,i}) \tag{21}$$

where $\Gamma(x)$ often is a sigmoidal[10], or "squashing function", function, e.g. the *logistic function* which results in a value in the interval $[0, 1]$ as viewed in figure 28. The logistic function is defined as

$$\Gamma(x) = \frac{1}{1 + e^{-x}} \tag{22}$$

which has a derivative[11] of

$$\Gamma'(x) = \Gamma(x)(1 - \Gamma(x)) \tag{23}$$

[10] Sigmoid means S-shaped.

[11] The derivative will be used in the training of the network with backpropagation, see appendix B.

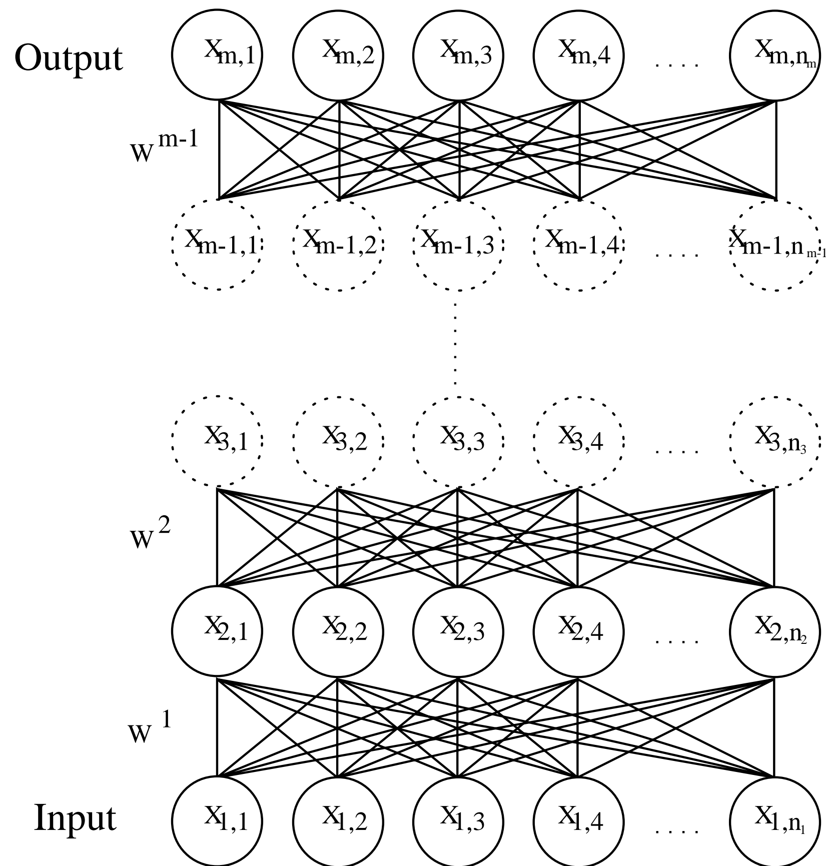

Figure 27: A generalized feed-forward network. Activation is fed into the network at the input layer and then propagated through the network via the weighted connections. The bias connections has been left out due to readability and is considered as internal properties of the nodes instead.

This computation is done for each node, layer by layer, until the activation of all the output nodes has been calculated. For convenience the activation in the output layer will be referred to as the vector o, where o_i corresponds to the activation $a_{m,i}$, i.e. node $X_{m,i}$ in figure 27. The activation of the output layer is the result of the computation in the feed-forward ANN. The training of the feed-forward network using backpropagation is explained in appendix B. Training can also be conducted with an *evolutionary algorithm*. This method is described in detail in appendix C.

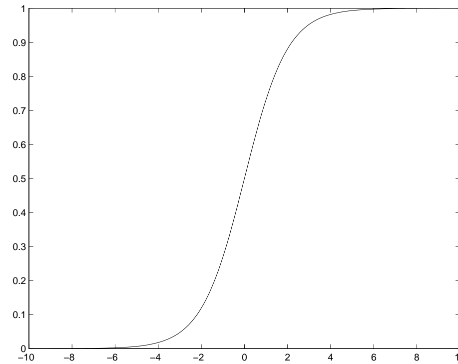

Figure 28: The logistic function as defined in equation 22.

A.2 Recurrent neural networks

If a feedback connection of any kind is added to the architecture in figure 27, such that the operation of the network at time t is influenced by previous activations and/or inputs to the network, the result will be a *recurrent* neural network (RNN). RNNs cannot only map input at time t to output at time t, as feed-forward ANNs do, but also classify patterns of input presented sequentially to the network, i.e. it can handle temporal tasks and "remember" previous input patterns to the network. RNNs have

become very popular in the study of temporal and sequential tasks which require a representation of time and/or structure [Elm90, Pol90, Pol91]. Language acquisition is a set of temporal problems with quite different requirements on the RNN depending on what language that is considered. Several different RNN architectures exist today in the connectionist arena. Two important groups of recurrent networks RNNs, SRNs and SCNs, are compared in this report. The difference of these networks is how the feedback is handled and that the second order network has multiplicative activation functions[12].

A.2.1 First-order recurrent networks

The first-order RNNs are basically feed-forward networks with a *copy-back connection* of some of the nodes in the hidden layer to some extra nodes in the input layer (see figure 29). After training, the weights typically remains fixed[13]. The first order recurrent network that will be considered here is the *simple recurrent network* (SRN)[Elm90], also called "Elman-networks" after their inventor. The hidden nodes which are copied back will be referred to as the *context nodes*[14], i.e the activation of the context nodes at time t will be part of the input at time $t+1$ Beside the copy-back connection of the context nodes, the operation of each node in the network is exactly like in the nodes of the feed-forward network.

The SRN can be trained with the standard backpropagation (BP) algorithm [RHW86] described in appendix B. The drawback of this algorithm for RNNs is that it does not remember previous inputs that may contribute to the error of the

[12] What multiplicative activation function means will be explained in A.2.2

[13] Mentioned only because the second order network has non-fixed weights

[14] This is perhaps not the terminology adapted by Elman but it suits our purposes since we want to use the same terminology for both types of networks tested in this project.

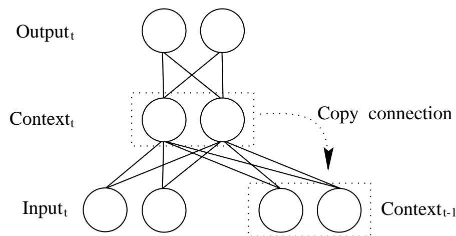

Figure 29: An example of a Simple Recurrent Network (SRN). The copy-back connection means that the activation of the hidden nodes, i.e. the context, at time t becomes part of the input at time $t + 1$.

network by affecting the context units. For problems where the output, at any time, depends mostly on the most previous input(s), this does not affect the performance of the algorithm strongly. Many SRNs have been successfully trained with standard BP, however, even when the problem was time-dependent, e.g. in [Elm90].

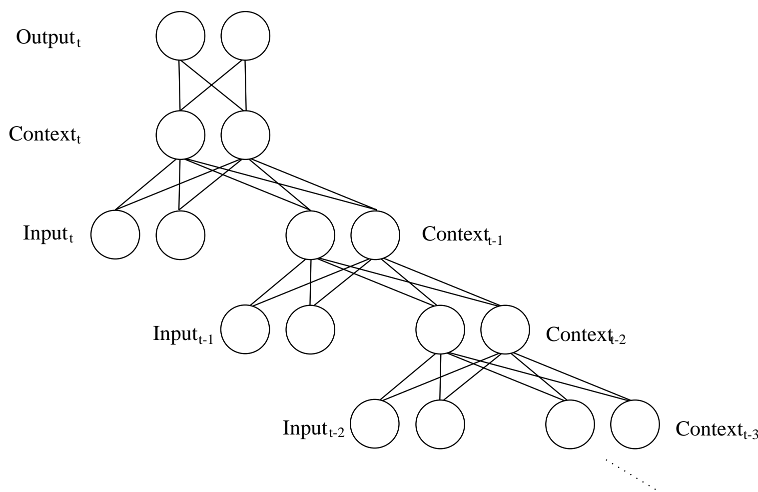

Figure 30: The result of unfolding the network in figure 29 with the BPTT algorithm. The unfolded network allows BP to account for more than just the previous input and context.

For some problems it is more suitable to use an algorithm that remembers a series

of inputs in the past and thus takes into account the fact that the network is recurrent and time dependent. Algorithms of this kind have a higher probability of finding a solution. One such algorithm is *back-propagation through time* (BPTT) which is described and analysed in [Wer94]. BPTT "unfolds" the network in time to produce a deeper non-recurrent network which allows the standard backpropagation algorithm, described in appendix B, to operate on the network. The result of unfolding the network in figure 29 is described in figure 30. The number of time-steps that the network should be unfolded is predetermined to satisfy the characteristics of the problem. This is a disadvantage of this algorithm since it is hard to know beforehand how many time-steps should be relevant for the problem at hand.

According to [Moz89] BPTT has two additional drawbacks. First, it is computationally expensive since the unfolding must take place once for every input. Secondly, the unfolding procedure creates a deeply layered network where the error contribution of each unit becomes dispersed. There are more disadvantages however, the problems with local optima in the training of non-recurrent networks are inherited and amplified with BPTT. For recurrent networks and time-dependent sequential problems the error-landscape is even more problematic for a gradient descent search. Figure 2 on page 7 shows a small sample of how the landscape may look like.

A.2.2 Second-order recurrent networks

In second-order RNNs, some of the weights are themselves a function of previous activation values, these weights will here be called *dynamical weights*. This means that the function of the network will be changed for each time step.

The second order network that will be considered in this project is Pollack's Sequential Cascaded Network [Pol86] (SCN) which can be thought of as consisting of two subnetworks: a *function network* that computes the activation of the output node(s) and a *context network* that computes some (or all) of the weights the function network. The weights of the context network will here be called *second order weights* and these typically remain fixed after the training.

To understand the recurrent SCN it may be best to first consider a non-recurrent second order network, e.g. one solving the XOR-problem. The XOR problem is a problem where a network should learn the logical function $XOR(x_1, x_2)$ where x_1 and x_2 are either 0 or 1 and $XOR(x_1, x_2) = 1$ if $x_1 \neq x_2$, otherwise $XOR(x_1, x_2) = 0$. In figure 31 a simple, non-recurrent, second order network, originally described in [Pol86] which solves the XOR-problem is shown. This network solves the problem with only 4 (fixed) weights instead of 7 (including the bias weights) which must be used to solve the problem with a first order network. It is also reported in [Pol86] that the second order network typically learned to solve the XOR-problem with approximately 5 times fewer epochs.

A second-order *recurrent* network changes its dynamic weights according to the internal state, i.e. the activation of the context nodes. The nodes which are the input to the context network will here be called *context nodes*. The activation of the context nodes updates the weights of the function network in every time-step, through the weights in the context network. Figure 32 describes a generalized topology of SCNs.

The dynamic weights are determined by

$$W_{ix}(t) = c_1(t-1)W_{c_1 W_{ix}} + c_2(t-1)W_{c_2 W_{ix}} + \cdots + c_m(t-1)W_{c_m W_{ix}} + W_{bW_{ix}} \quad (24)$$

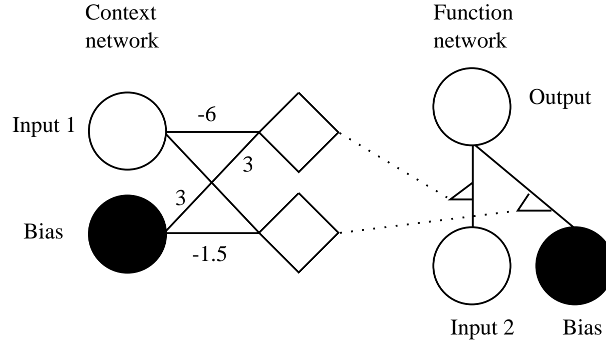

Figure 31: A second order non-recurrent network that solves the XOR-problem. (Figure redrawn from [Pol86]). The diamonds represent linear units where the *net*-value is directly copied to the activation, a, without any "squashing"-function as in equation 21. The dotted connections indicate that the activation of the linear units is copied to the weights. The black nodes are constantly active nodes and the weights of the connections from them corresponds to the biases.

for a weight from input node i to a context or output node x. The activation x of any context or output node in the architecture of figure 32 at time t is determined by

$$\begin{aligned} x(t) = \Gamma(& c_1(t-1)(i_1(t)W_{c_1W_{i_1x}} + i_2(t)W_{c_1W_{i_2x}} + \cdots + i_n(t)W_{c_1W_{i_nx}} + W_{c_1W_{bx}}) + \\ & c_2(t-1)(i_1(t)W_{c_2W_{i_1x}} + i_2(t)W_{c_2W_{i_2x}} + \cdots + i_n(t)W_{c_2W_{i_nx}} + W_{c_2W_{bx}}) + \\ & \vdots \\ & c_m(t-1)(i_1(t)W_{c_mW_{i_1x}} + i_2(t)W_{c_mW_{i_2x}} + \cdots + i_n(t)W_{c_mW_{i_nx}} + W_{c_2W_{bx}}) + \\ & i_1(t)W_{bW_{i_1x}} + i_2(t)W_{bW_{i_2x}} + \cdots + i_n(t)W_{bW_{i_nx}} + W_{bW_{bx}}) \end{aligned} \tag{25}$$

where $W_{\alpha W_{\beta x}}$ is the second order weight from context node α to the dynamical weight between input node β and x. b is the "bias node" which always has an activation of 1.0.

As can be seen in equation 25, the activation from the input and the context nodes

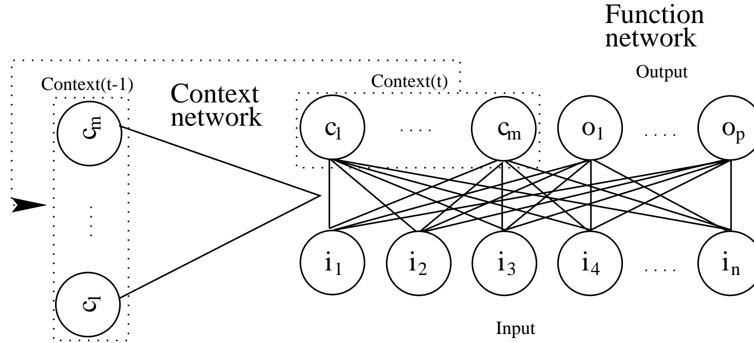

Figure 32: A sequential cascaded network (SCN), as proposed in [Pol86]. The activation of the context units at time t is the input of the context network at time $t+1$. The biases have been left out in this figure, and so have the linear units which of course still are present, one for each weight in the function network. This figure is not the most general, there might be more layers in the context network and layers may also be added on top of the output nodes in the function network.

are not only summed up, but also multiplied. Therefore the activation function of SCNs is sometimes called *multiplicative* [Kol94].

The training of SCNs using a variant of BP is explained in appendix B. Training of ANNs in general, using evolutionary algorithms, is described in appendix C.

SCNs have been used in language acquisition tasks in [Pol86], [GSC+90], [Pol91], [Pol90] and [KÑF98] to mention a few examples.

Appendix B

Backpropagation

B.1 BP training of SRN

The *backpropagation* algorithm (BP) was proposed in [RHW86] as a gradient descent search method to adapt feed-forward networks with hidden layers. The algorithm will here be described as a part of a supervised training regime, but the algorithm can be generalized to other training regimes as well, e.g. as part of reinforcement learning, e.g. [Mee96]. The algorithm is based on the algorithm for training *perceptrons*, which do not have any hidden layers. The basic idea of the algorithm is to calculate the derivative of the error given the parameters of the networks, i.e. the weights and biases of the network. The parameters will be adapted, using the information obtained by the derivation, to minimize the error of the network. The error is often calculated as the *summed squared error* (SSE), based on the difference between the desired and actual output of the network. The error when training with a supervised regime is summed up for all examples in the training set. In detail, the error for one known

input-output example is calculated by

$$e = \frac{1}{2}\sum_{i=1}^{n_m}(o_i - \tau_i)^2 \tag{26}$$

where i is the input vector to the network, τ the target vector and o the actual output vector from the network. The total error of the network, i.e. for all examples in the training set is given by

$$E = \sum_{p=1}^{k} e_p \tag{27}$$

where k is the number of examples in the training set. The principle that guides a gradient descent search-method is to adjust each parameter, stepwise, in the direction of the error landscape which appears to lead towards a lower error. The direction is in this case calculated using the derivative (see figure 33).

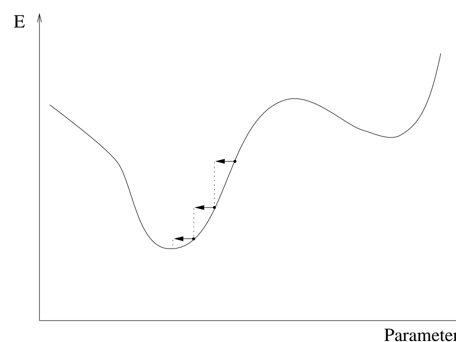

Figure 33: The principle that guides the gradient descent search is to calculate the derivative $\partial E/\partial Parameter$ and adjust the parameter (e.g. a weight), step by step towards a lower error.

The derivative of the error is calculated for each activation in the network. This

will, if the logistic activation function is used (equation 22), result in

$$\frac{\partial E}{\partial o_i} = (o_i - \tau_i)\Gamma'(o_i) \tag{28}$$

for the nodes in the output layer. Then the error of the hidden layers is calculated by propagating the error from the layers above to the preceding layers (remember that $a_{m,i}$ corresponds to o_i, where m is the number of layers):

$$\frac{\partial E}{\partial a_{\ell,i}} = \left(\sum_{j=1}^{n_{\ell+1}} W_{i,j}^{\ell} \frac{\partial E}{\partial a_{\ell+1,j}}\right) \Gamma'(a_{\ell,i}) \tag{29}$$

When the error-derivative has been calculated given the activations of the nodes it is possible to calculate the derivatives of the error for all the weights.

$$\frac{\partial E}{\partial W_{i,j}^{\ell}} = \frac{\partial E}{\partial a_{\ell+1,j}} a_{\ell,i} \tag{30}$$

And equally for the biases of the nodes.

$$\frac{\partial E}{\partial W_{b_i}^{\ell}} = \frac{\partial E}{\partial a_{\ell+1,j}} \tag{31}$$

The updating of the weights and biases is then done according to the principle in figure 33.

$$\partial W_{i,j}^{\ell} = -\eta \frac{\partial E}{\partial W_{i,j}^{\ell}} \tag{32}$$

And likewise for for the biases

$$\partial W_{b_i}^{\ell} = -\eta \frac{\partial E}{\partial W_{b_i}^{\ell}} \tag{33}$$

where η is the *learning rate*, typically a low value, e.g. 0.1 or 0.01.

B.2 BP training of SCN – The "backspace trick"

The "backspace trick" was suggested by Pollack in [Pol91] and is a backpropagation method designed for training of SCNs specifically. The problem when training this type of network is that only part of the target output is know at a time. The gradients of the weights from the input nodes to the context nodes can not be calculated directly since there is no "target" context. The trick is described in figure 34.

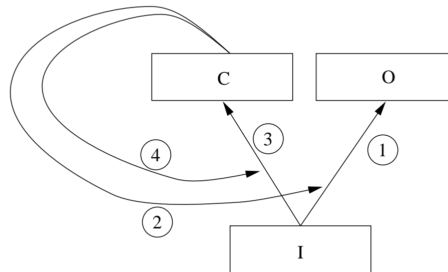

Figure 34: The principle behind the backspace trick. First the error gradients of the dynamical weights that lead to the output nodes are calculated (step 1). These gradients are then used to calculate the gradients of the second order weights that lead to these dynamical weights (step 2). Now the error can be propagated back to the context nodes and then through the remaining dynamical (step 3) and second order weights (step 4).

Equation 25 on page 112 can be simplified to

$$x(t) = \Gamma(\sum_{k=1}^{m}\sum_{j=1}^{n} W_{c_k W_{i_j x}} c_k(t-1) i_j(t)) \tag{34}$$

which makes the following equations easier to derive. The gradients of the bias-weights has been left out of the following equations but can easily be derived from them.

First the error gradient of the weights leading to the output node must be calculated. As for the SRN, we start by calculating the error gradients of the output nodes.

$$\frac{\partial E}{\partial o_k(t)} = (o_k(t) - \tau_k(t))\Gamma'(o_k(t)) \tag{35}$$

where $\tau_k(t)$ is the target output at time t. Then the errors of the dynamical weights leading to the output nodes from the input nodes can be calculated (step 1 in figure 34)

$$\frac{\partial E}{\partial W_{i_j o_k}(t)} = \frac{\partial E}{\partial o_k(t)} i_j(t) \tag{36}$$

Then the gradients of those second order weights that leads to the output nodes can be calculated (step 2 in figure 34)

$$\frac{\partial E}{\partial W_{c_l W_{i_j o_k}}} = \frac{\partial E}{\partial W_{i_j o_k}(t)} c_l(t-1) \tag{37}$$

The error of the context nodes can now be calculated

$$\frac{\partial E}{\partial c_l(t-1)} = \sum_{k=1}^{m}\sum_{j=1}^{n} \frac{\partial E}{\partial W_{c_l W_{i_j o_k}}} \frac{\partial E}{\partial W_{i_j o_k}(t)} \tag{38}$$

where m is the number of output units and n is the number of input units. The error gradients of the dynamical weights from input nodes to the context nodes can now be calculated by using the previous input (step 3 in figure 34)

$$\frac{\partial E}{W_{i_j c_l}(t-1)} = \frac{\partial E}{\partial c_l(t-1)} i_j(t-1) \tag{39}$$

and finally the second order weights for the context nodes can be calculated (step 4 in figure 34).

$$\frac{\partial E}{W_{W_{i_j c_l}}} = \frac{\partial E}{\partial W_{i_j c_l}(t-1)} c_j(t-2) \tag{40}$$

The updating of the second order weights is then carried out by using the error gradients from equation 37 and 40.

$$\partial W_{c_l W_{i_j x_k}} i = -\eta \frac{\partial E}{\partial W_{c_l W_{i_j x_k}}} \tag{41}$$

where x_k is either an output or context node.

Appendix C

Evolutionary algorithms

The search space when training ANNs is often nonlinear and full of "trap-holes" that a gradient descent search (such as BP) may fall into. Therefore, it is desirable to have a search method less sensitive to the structure of the search space. *Evolutionary algorithms* (EAs) are a class of search algorithms that handle this kind of search-space better. They were originally suggested in [Hol75] and other good references are [Gol89] and [Mit95]. EAs are based on the principles of how nature evolves well-adapted life on earth as first described by Charles Darwin in 1872 [Dar72].

The evolutionary algorithms all have some kind of *genetic representation* of the individual solutions. These are called *genotypes*. The solutions which are encoded in the genotypes are called *phenotypes*. When we consider training of neural networks, the genotypes may be vectors of real numbers that encode the weights and the phenotypes are represented by the final networks. More flexible types of EAs, specially developed for the adaption of ANNs, exist also. They may for example also include the topology of the network in the genotypic representation. Training of ANNs using EAs is discussed in [Whi95].

Figure 35 shows a schematic illustration over an EA. An initial population of typically randomly generated individuals is first created. The individual genotypes are then evaluated by testing their corresponding phenotypes on the problem which the EA is to solve. This evaluation results in a fitness value for each individual which should reflect how well they solve the problem. Then a subset of the population is selected for reproduction. This selection may be implemented in many ways but is always biased toward selecting the best individuals of the population, i.e. those that, according to the fitness value, solve the problem best.

The reproduction can also be implemented in many ways, but typically the EA uses two operators on the genotypes, *crossover* and *mutation*. Crossover means that the genes of a pair of individuals are combined into a new individual, i.e. sexual reproduction. This is however not often used when training ANNs since it is difficult to genetically encode ANNs into a representation that allows for meaningful recombinations, see [Whi95] for more details of this. "Mutation" means that individual elements of the genotype are manipulated randomly. Since ANNs are the subject of optimization here, mutation will be the only operator considered in the implementation.

Figure 36 shows a simple example of how a phenotype may be encoded in the genotype and how the evaluation may end up in a fitness value. The genotypic representation used in this project is simply a vector of real numbers, one real number per weight and bias.

There are many different types of EAs. They work with different encodings in the genotypes and are therefore suitable for different kinds of problems. One of the most common type of evolutionary algorithms is genetic algorithms (GAs) which

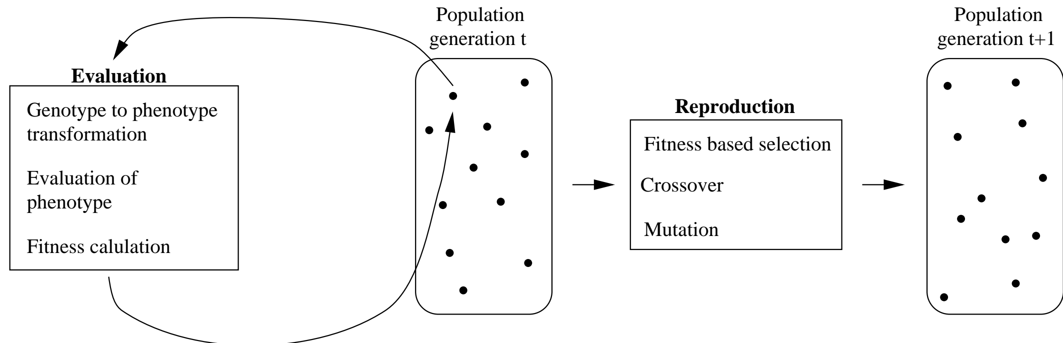

Figure 35: A schematic illustration over the basic principals of the operations in an EA. The process is iterated for several generations until a termination criteria is satisfied.

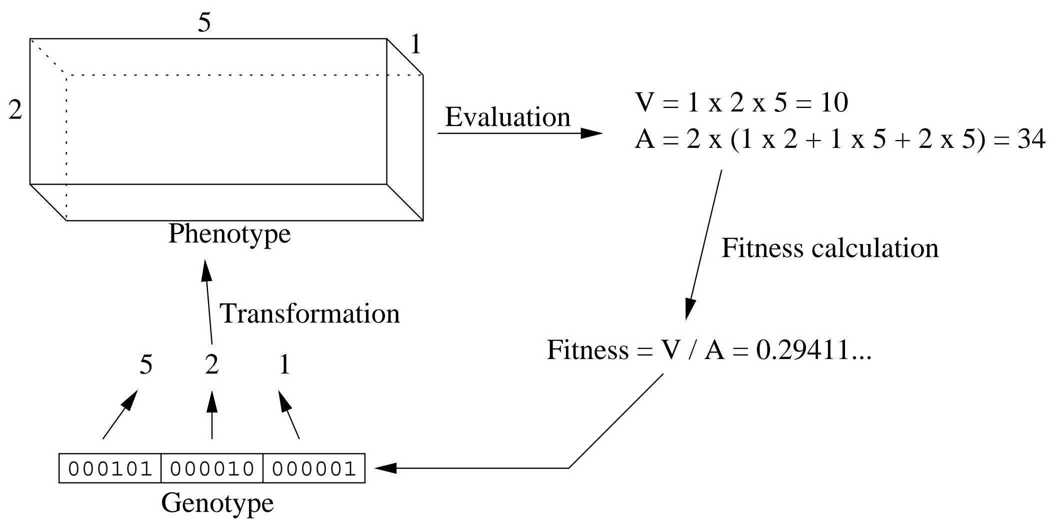

Figure 36: An example of an evaluation of an individual. In this example the fitness will be high for phenotypes with large volumes and small surface areas. The evaluation process is essential for the algorithm since it determines what features that will be promoted in the evolution.

have binary representations in the genotypes and are suitable for a wide range of problems [Hol75][Gol89]. They are however less suitable for problems where a real-valued parameter space is mapped to the problem instances, e.g. as is the case for the weights in a neural network [Whi95]. The binary representation of the GA is often decoded to its corresponding integer values (like in figure 36) and mapping the integer linearly to intervals of real values. The problem with using a GA for a real-valued problem is (taken partly from [Bäc96]):

- We are bound to the chosen intervals. No individuals may explore beyond these intervals. If there is only little background knowledge about the problem then the reasonable interval is hard to measure. In the ANN domain this would mean that we have to specify in advance, the maximum and minimum values of the weights in the network.
- The solution is bound to a discrete set of solutions since the bit strings are discrete by nature. There is, at least in theory, a possibility that a solution to the problem cannot be represented by the binary representation. This can partly be solved by using larger bit-strings to get a richer space of candidate solutions, but this increases the search space for the GA.
- One of the biggest disadvantages is that the bits in the genotypic string will have different influence on the corresponding real-valued parameters depending on their position in the string, e.g. the leftmost bits in the genes of figure 36 affects the corresponding phenotype more than the rightmost bits. This decreases the performance of the GA since a mutation on the leftmost bit will change the corresponding parameter radically if an integer encoding is used.

> We do not want mutations to make too large leaps in search-space, since a too big mutation is essentially the same as a "random guess" which has a low probability of success.

This means that GAs have inherent drawbacks when it comes to adaptation of real-valued parameters instead of boolean. One proposed solution is called evolution strategies [Bäc96] (ESs) which operate on real-valued parameters instead of binary strings. This means that an ES is potentially better at optimizing problems with real-valued parameters [Bäc96]. The biggest difference between a GA and an ES is the mutation-operator.

One of the most common approaches is to let all the genes be mutated by adding *normal* or *Gaussian-distributed* values. The Gaussian distribution has the probability density function

$$p(x, \sigma, \mu) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \tag{42}$$

where μ is the centre of the distribution, normally set to zero in EAs since mutations should not be biased towards any special direction. σ is the standard deviation of the distribution and approximately $\frac{2}{3}$ of the distribution lies within $-\sigma$ and σ. The higher the value of σ the higher probability of large mutations (see figure 37).

Small mutations will be more common than large when using the Gaussian distribution, and what is referred to as the *mutation rate* or *mutation parameter* is really the σ in the distribution. It is possible to use a uniform mutation rate, i.e. the same σ for all genes and individuals or one mutation rate for each gene and individual, a mutation rate that is passed on to the next generation together with the gene itself. The idea behind this is that the mutation-parameters will be adapted, just as the

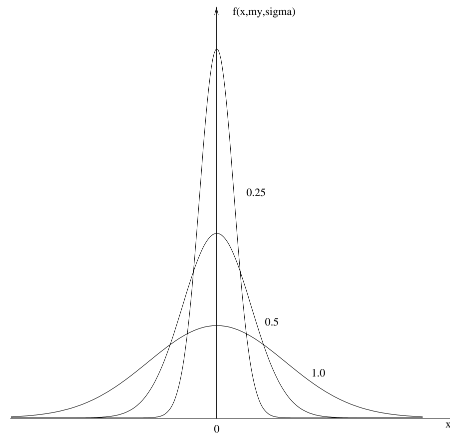

Figure 37: The Gaussian distribution when $\mu = 0$ (i.e. distribution centered around zero) and different values of σ.

parameters themselves. The mutation rates will then decrease over time, when the population converges closer to a solution [Bäc96]. The EA used in this project is a simplification of ES since our EA uses a fixed mutation parameter for all individuals and generations.

Appendix D

Dynamic systems theory

Dynamic systems theory (DST) offers a way to describe neural networks as an *iterated function systems* (IFS) [Kol94]. An IFS is basically a system consisting of a state space $\mathcal{S}$ and a *transformation function* F such that the the state at time t, $x_t \in \mathcal{S}$, determines the state at the next time-step x_{t+1}, i.e. $x_{t+1} = F(x_t)$. This can also be written as *iteratives* of F, i.e. $x_n = F^n(x_0)$. For example, $F^2(x_0)$ is the same as writing $F(F(x_0))$.

In a RNN the state space is the activation of the context units and the function is the network itself which, given a fixed architecture, is determined by the weights of the network and the input to the network, i.e.

$$x_{t+1} = F_W(x_t, i_t) \tag{43}$$

where W is all the fixed weights in the network. Depending on W and the sequence of inputs, the system will have different behaviours. We will look into the characteristics

of these behaviours by looking on a simple nonlinear function without any input:

$$x_{t+1} = f_a(x_t) = a(1 - x_t)x_t \quad 0 < x_t < 1, \quad 0 < a < 4 \tag{44}$$

When the function is iterated we get a sequence of x, $x_0, x_1, x_2, \ldots x_n$, which will be referred to as an *orbit*. The dynamics of iterating f_a are different for different values of a, just as the dynamics can be different in RNNs with different weight configurations (see figure 38).

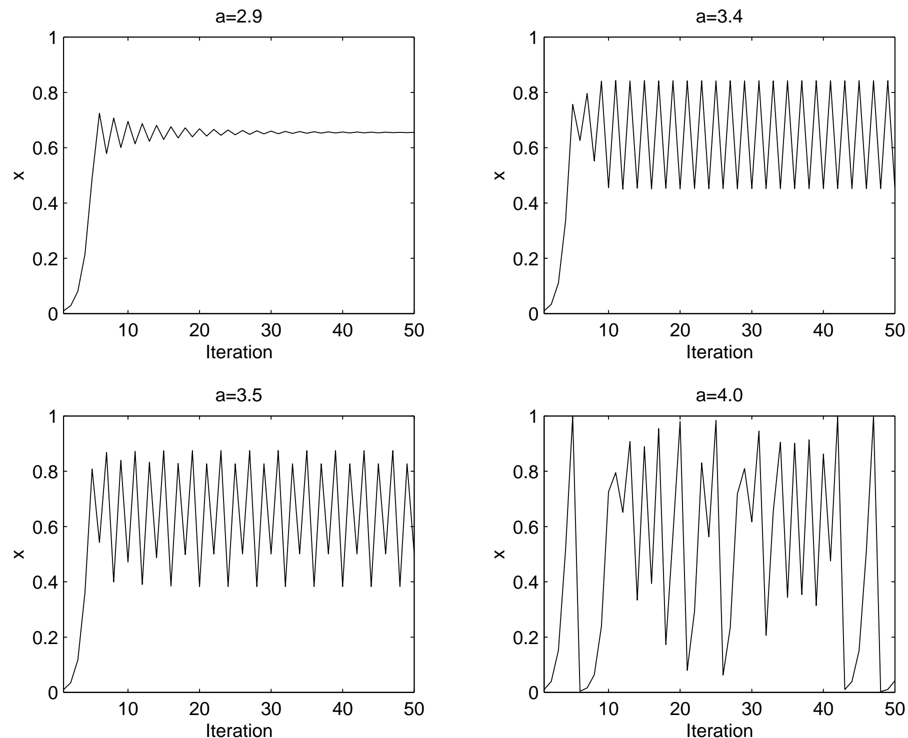

Figure 38: Different dynamic behaviours of f_a, with different values of a. This shows an example of a system that may have more than one dynamic behaviour. The same may hold for recurrent neural networks that may adopt different dynamics to solve different problems.

D.1 Attractors

D.1.1 Fixed point attractors

To understand the concept of *attractors* we must first understand *fixed points.* A fixed point is a point of the state space which will be transformed into itself by the transformation function F, i.e. if x_0 is a fixed point then $x_0 = F(x_0)$. There are different types of fixed points, the following definition is taken from [Dev92]:

Definition 5 Suppose x_0 is a fixed point for F. Then x_0 is an *attracting fixed point* if $|F'(x_0)| < 1$. The point x_0 is a *repelling fixed point* if $|F'(x_0)| > 1$. Finally, if $F'(x_0) = 1$, the fixed point is called neutral or indifferent.

Some examples of orbits, based on functions on real numbers, with different types of fixed points are shown in table 28.

Function	Orbit, i.e. $x_0, x_1, x_2 \ldots x_\infty$	Type	Fixed point(s)
$F(x) = 0.9x$	$1.0, 0.9, 0.81, 0.729, 0.6561 \ldots 0.0$	Attracting	0.0
$F(x) = 1.1x$	$1.0, 1.1, 1.21, 1.331, 1.4641 \ldots \infty$	Repelling	0.0
$F(x) = -0.9x$	$1.0, -0.9, 0.81, -0.729, 0.6561 \ldots 0.0$	Attracting	0.0
$F(x) = -1.1x$	$1.0, -1.1, 1.21, -1.331, 1.4641 \ldots NaN$	Repelling	0.0
$F(x) = x$	$1.0, 1.0, 1.0, 1.0, 1.0 \ldots 1.0$	Neutral	$\forall x \in \mathcal{R}$

Table 28: Some simple examples of attracting, repelling and neutral fixed points.

It is also interesting to distinguish between two different types of orbits towards, or from, the fixed point.

Definition 6 Suppose $x_0, x_1, x_2 \ldots x_n$ is an orbit for F where x_0 is in the vicinity of the fixed point attractor. Then this orbit is *monotonic* if $F'(x_i) > 0$ for $0 < i < n$. If $F'(x_i) < 0$ for $0 < i < n$ then the orbit is *oscillating.*

This definition is narrow in the sense that, for some systems, the orbit may actually be both oscillating and monotonic at different times. But for our purposes and for fixed point attractors, this definition is enough for parts of the orbit.

Now, we know that fixed points comes in three flavours, attracting, repelling and neutral. A *fixed point attractor* is simply defined as an attracting fixed point. Attractors are often easier to analyse when dealing with IFSs in practice since repelling fixed points will be only be visited if the system starts exactly in the fixed point. Any small deviation from the fixed point will be amplified if it is repelling.

D.1.2 Periodic attractors

If there is an orbit of F which is repeatedly returning to a previously visited state then it has a *periodic fixed point.*

Definition 7 Let $x_0, x_1, x_2 \ldots x_n$ be a part of an orbit determined by F. Then if $x_0 = x_n$ the orbit has an n-cycle. In other words, we say that x_0 is a *periodic fixed point of period i.*

To determine if a periodic point fixed point x_0 of period n is attracting or repelling we can use the same principle as for fixed points. In [Dev92] it is shown, using the chain-rule for derivatives, that the derivative of $(F^n)(x_0)$ can be calculated by

$$(F^n)'(x_0) = F'(x_0) \cdot F'(x_1) \ldots F'(x_{n-2}) \cdot F'(x_{n-1}) \tag{45}$$

The same set of rules as for fixed points can then be used to classify the periodic fixed point into attracting, repelling and neutral periodic fixed points, i.e. the periodic fixed point is attracting if $|(F^n)'(x_0)| < 1$, repelling if $|(F^n)'(x_0)| > 1$ and neutral

if $|(F^n)'(x_0)| = 1$. Two examples of periodic attractors is found in figure 38. If a parameter is changed in the system, the period of the periodic attractor may change, this is viewed in figure 39.

D.1.3 Strange attractors

Strange attractors are the third class of attractors which only will be covered briefly here (a more detailed description is found in [Dev92]). A strange attractor is an attracting subpart of the state-space which leads the system into an orbit which will never repeat itself. This type of attractor is also called a *chaotic* attractor since it is sensitive to initial conditions and therefore cannot be predicted, i.e. the observer will consider the system to be chaotic. A small deviation in the initial state will result in widely different orbits. For example, if you place two identical pieces of wood in a small stream, their paths will eventually be divided, no matter how close you put them together. The sensitivity of the initial condition must not be confused with the sensitivity associated with repelling fixed or periodic points since these are no attractors and will "push" the system's state away from themselves. The strange attractor *is* attracting and the sensitivity to initial conditions will only result in different orbits within the attractor. A simple example of an orbit is found in table 29.

The fact that systems with strange attractors and chaotic behaviour are so sensitive to initial conditions make them hard to predict. The weather is an example of a system where we cannot exactly know the initial condition, so no matter how good models we have of the weather processes, we cannot predict more than a few days into the future.

x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8
0.1000	0.3600	0.9216	0.2890	0.8219	0.5854	0.9708	0.1133	0.4020

Table 29: The beginning of the orbit started from $x_0 = 0.1$ when $a = 4$ in the function f_a from equation 44. The attractor is strange and is in the limited interval [0,1]. In theory, with a truly real-valued x, the orbit will never repeat itself. If $x_0 = 0$ or $x_0 = 1$ the system is attracted to the fixed point 0, i.e. the system has two different attractors.

D.1.4 Basin of attraction

Some system may have more than one attractor at the same time, even of different types. Each attractor has a region in the state space that will eventually lead to the attractor, this region is called *the basin of attraction* of that attractor. Depending on which basin the system starts in it will, eventually, end up in the corresponding attractor.

If we discuss the training of networks, using BP or BPTT, from a dynamic systems point of view, a successful network will be a fixed point attractor with a basin of attraction corresponding to the initial networks that will eventually lead to the successful network.

Figure 39: The attractors of f_a as a function of a. from $a = 0.0$ to $a = 3.0$ the attractor is a fixed point. At $a = 3.0$ a *bifurcation* occurs when the attractor becomes periodic. At higher values of a the attractor sometimes is strange, or chaotic.