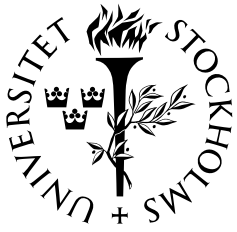

EXAMENSARBETEN I MATEMATIK

MATEMATISKA INSTITUTIONEN, STOCKHOLMS UNIVERSITET

Space limitations in the formal language acquisiton of $a^n b^n$

av

Julia Uddén

2007 - No 2

MATEMATISKA INSTITUTIONEN, STOCKHOLMS UNIVERSITET, 10691 STOCKHOLM

Space limitations in the formal language acquisiton of $a^n b^n$

Julia Uddén

Examensarbete i matematik 20 poäng, fördjupningskurs

Handledare: Karl Magnus Petersson

2007

Abstract

In the attempt to relate cognitive and neural descriptions of mental functions, using the human language faculty as a model, a fundamental difficulty is to determine which theoretical description to start with. When comparing theories, it is of great importance to develop concepts describing biologically relevant properties common to all languages and other mental functions, such as recursive structures. We describe the symmetric language pattern $a^n b^n$, generated by the recursive process of adding the component ab in the middle of the pattern in each step. Our contribution is to formalize this intuition through implementing recognition of this pattern in abstract system descriptions of cognition (theoretical machines such as finite automata and neural networks). As a result of taking space limitation in the physical brain into account, we propose finite state machines as an interesting conceptual framework.

Acknowledgements

Taking the freedom of writing a lengthy acknowledgement in spite of a modest thesis - first and foremost I wish to thank Karl Magnus Petersson, a too versatile man to be able to approximate in a sentence. Besides his work and comments, Richard Feynman's writing, Gilbert Strang and Roger Penrose's lectures were the most spectacular intellectual injections during writing this thesis. Thanks to Yishao Zhou and the mathematical institution at Stockholm University for your flexibility and patience. I would also like to thank some people that have been important guides along the pleasurable trip towards understanding of what the field of mathematics is and the wonderful things that emerge when we apply it: Markus and Anna-Karin Kallioinen, Barbro Ödlund, Mihaj Lazarescu, Paul Vaderlind, Sören Holst, Lars Gråsjö, Susanne Gennow, the Chapovalova family, $\aleph_0$ and the girls. The Lansner group and the summer schools in Paris and Frankfurt opened the door to the promising field of computational neuroscience, Martin Ingvar provided helpful practical constraints (as always with wit) and François Grimbert dared to confront me with additional intellectual constraints. Many thanks to Giosue Baggio for presenting the topic through the linguistic looking glass. The Fortaians, Andjeas Ejiksson, Joel Uddén, Martin Eriksson with his GEB/transhumanism study group and all you fellas at FCDC and SBI (you know who you are) covered the range from unhealthy to divine inspiration.

Notation

$\mathcal{C}$ (the class of computable functions)

$P[l_1, l_2, l_3, \ldots \rightarrow l_p]$

Instruction to program P to compute on input l_1, l_2, l_3 and store the result in l_p

$\{a, b, ...\}^+$

all non-empty, finite strings from the alphabet $\sum = \{a, b, ...\}$

$\sum^*$ (Kleene closure)

the set of all finite strings over $\sum$ including the empty string.

$\ominus$ (cut-off subtraction, or monus)

$x \ominus y = 0$ if $x < y$
$x \ominus y = x - y$ if $x \geq y$

$a^n b^n$

the language of stings consisting of n a's followed by n b's, where $n \in N$

aaaaaaaabbbbbbbb
aaaabbbb
aabb
ab

Contents

Chapter 1

Introduction

In april 2006, a paper appeared in Nature (Gentner, Fenn, Margoliash & Nusbaum, 2006) stating that European Starlings (a singing bird species) can learn recursive syntactic patterns. The strings presented to the birds were taken from the language $a^n b^n$, which capture the concept of centrally embedded recursive structures. Since then, the linguistic community, all the way up to it top theoreticians, has debated if this capacity has implications on theories about the origin of language. The results have been interpreted as contradictory to the hypothesis that recursive embedded structure is at the core of the unique human language faculty (Fitch & Hauser, 2004). One counter argument is that the results are dismissible since the Starlings could have counted the a's and b's (sung to them as syllables with male or female voices) and thus classify strings without a proper sense of grammar. Our theoretical investigation shows that being able to count (or something equivalent to the counting functionality) might be the only way to learn to categorize this kind of recursive structures. The scope of this thesis is to introduce useful concepts from the mathematics of symbols, computability theory, the neural networks literature and a tinge of dynamical systems theory, to shed light on this discussion.

We start with a review of how cognitive functions can be implemented in some simple theoretical mechanisms or machines (called automata, for a full introduction see Cohen, 1997). At some level of abstraction, these can be seen as models for both the bird and human cortex. However, some of these systems might be considered too unrealistic with respect to biological constraints. That is why we develop special cases of earlier proved general results about formal language recognition or computation in the framework of unlimited register machines (Cutland, 1970), analog neural networks (Siegelmann, 1999) and simple recurrent networks (Elman, 1991). In other words, we are going from simple abstractions to biologically inspired implementations by gradually introducing well chosen constraints. The two later frameworks touch on assumptions on space limitation and limited processing or architectural precision. Simply put, these assumptions are relevant since the human brain is finite and thus most likely has

a finite storage capacity. In addition, noise sources introduce a limitation on the precision with which the brain can process information. The most prominent sources are probably imprecision in the structural architecture, synaptic noise such as the probabilistic nature of synaptic transmission and variability in the post synaptic potential (Koch, 1999). Thermal noise is of course also present, but most likely negligible at room temperature. We conclude that research from last ten years make a strong case in favor of the finite state architecture of mind (Minsky, 1967, Petersson, 2005, Wells, 2005), defined below. This is of importance for the foundations of linguistics. The Chomsky hierarchy of grammars fades into a sand castle of platonic play, since complexity levels above the finite state architecture rely on the assumption of an infinite external memory capacity. Although it might seem trivial, we are not infinite users of finite means, since we can not produce or recognize language patterns that require such fantastic amounts of external memory. However, we have gained something more interesting, a well grounded outlook on the implementation level, where one can seek further understanding of the neural underpinnings of syntactic processing (e.g. by conducting empirical research on the syntax learning phase in infants and second language adult learners. With simple artificial grammar learning, more robust physiological phenomena can be studied, since they can be taught to other animal species and human learners, even when affected by pathologies, for instance dysgraphia, dyslexia and developmental aphasia).

1.1 Formal languages

Let us begin with a **symbol**, the smallest syntactic entity. An **alphabet** $\sum$ is a finite set of symbols. A **string** ω of an alphabet is a finite sequence of symbols of that alphabet. The **set of all strings** over an alphabet is denoted $\sum^*$ and any subset of this set is called a **language**. The empty set $\emptyset$ is a language, over all alphabets, consisting of no string at all. We observe that the set of all strings contains infinitely many strings over any non-empty alphabet. Furthermore, if we allow infinite strings over an alphabet with two or more symbols the set of strings contains an uncountable infinity of strings, which is easily proved by a diagonal argument. Usually, this is avoided by regarding the Kleene closure $\sum^*$ of finite strings as described above. The set of strings in a infinite language L must be extensionally defined as in the scheme:

$$L = \{\omega \in \sum^* : \omega \text{ has the property P}\}$$

However, it might not be transparent from a given ω whether it has the property P or not. Suppose we want to find a systematic way of answering the question "Is ω in L?". Formally, this is to find a language recognition **algorithm**, i.e. a finite set of ordered instructions for solving this problem in a finite number of steps.
Another task is "Generate a string in L". This is done by language generators.

Grammars are the most spread form of language generators and in a sense, they can also be seen as being finite definitions of languages. Typically, grammars are non-deterministic generative devices and not deterministic algorithms since it is not specified in which order the instructions should be performed; but they can be viewed as non-deterministic algorithms. One might think that absence of the deterministic constraint could increase the computational power, in the sense of giving possibilities of specifying additional languages. However, this turns out not to be the case, since a non-deterministic device can be simulated in exponential time by a deterministic equivalent that goes through all possibilities of each computational step in a systematic way (Lewis & Papadimitriou, 1998).

1.2 Languages as functions

In order to relate the mathematics of symbols to other fields of mathematics, it is important to establish the equivalence between formal languages and indicator functions. Any function with a binary range can be viewed as defining a language and any language corresponds to an indicator function in the sense that the set of domain elements mapped to one stipulates the language.

$$f(\mathbf{x}) = \left\{ \begin{array}{lll} 1 & iff & \omega \in L \\ 0 & iff & \omega \notin L \end{array} \right\}$$

It turns out that the interesting class of computable functions corresponds to the formal languages described in the Chomsky hierarchy, which we return to when introducing grammars. The next chapter is a case study of this equivalence where we show that $a^n b^n$ is a computable function.

Chapter 2

Computability

Many approaches have been made to stipulate the class $\mathcal{C}$ of computable functions, including Turing machines, Goedel's and Kleene's Recursive functions, Church's λ-calculus and the symbol manipulation systems of Post and Markov. We will outline a relatively recent framework centered on **unlimited register machines** (URM), which can be viewed as a mathematical idealization of computers, or more specifically the CPU of a computer.

Definition 1 *An URM has a countably infinite number of registers $R_1, R_2, R_3, \ldots$ containing natural numbers $r_1, r_2, r_3, \ldots$. The URM changes the content of the registers according to finite sets of instructions $I_1, I_2, I_3, \ldots, I_k$ called **programs**. After executing instruction I_i it proceeds to I_{i+1} (with the exeption of the jump instruction, defined below).*

The class of computable functions can be computed by the URM if we allow the programs to be composed of the following instructions:

1. **Zero instruction** Z(n) changing r_n to 0.
2. **Successor instruction** S(n) increasing r_n by 1.
3. **Transfer instruction** T(m,n) changing r_n to r_m
4. **Jump instruction** J(m,n,q) evaluating if r_n equals r_m and if so, jump to instruction q.

The URM must be loaded or provided with an initial configuration; that is, a finite, non-empty sequence of natural numbers in the registers which are called the input to the computation.

2.1 Recursion

In order to establish computability for some functions needed to prove the computability of our language, we need the following theorem showing that $\mathcal{C}$ is closed under definition by recursion. The crucial idea is the introduction of a time dimension, specifying in which order the values of the constructed function has to be obtained. This is a theme reoccurs in the network constructions, see further Elman 91.

Theorem 1 *Let* $\boldsymbol{x}=(x_1, x_2, ..., x_n)$ *and suppose f(*$\boldsymbol{x}$*) and g(*$\boldsymbol{x}$*, y, z) are computable functions. Then there is a unique function h(x,y) satisfying*

h($\boldsymbol{x}$*,0)=f(*$\boldsymbol{x}$*)*
h($\boldsymbol{x}$*,y+1)=g(*$\boldsymbol{x}$*,y,h(*$\boldsymbol{x}$*,y))*

and this function is computable.

Proof The uniqueness is garuanteed by the definition as follows. Suppose we have two different functions h_1 and h_2. Now for an arbitruary $\mathbf{x}$, $h_1(\mathbf{x}, 0) = h_2(\mathbf{x}, 0) = f(\mathbf{x})$ which gives the basis step. Suppose $h_1(\mathbf{x}, p) = h_2(\mathbf{x}, p)$, then $h_1(\mathbf{x}, p+1) = h_2(\mathbf{x}, p+1) = g(\mathbf{x}, p+1, h(\mathbf{x}, p+1))$. By induction, $h(\mathbf{x}, y)$ we have shown that $h_1(\mathbf{x}, y)$ and $h_2(\mathbf{x}, y)$ have equivalent maps for any choice of $\mathbf{x}$ and y, thus $h_1 = h_2 = h$.

It is clear that h is computable since we have specified an algorithm for producing it, which is the essential definition of computability. To see this from the URM point of view, simply concatenate the instructions of the programs we know exist for f and g into one program, h. For details on register organization, see next proof.

Cut off subtraction, $\ominus$ by one, that is x-1 on $\mathbb{N}$ is then computable if we take f($\mathbf{x}$)=0 and g($\mathbf{x}$, y, z)=x, i.e.:

$0 \ominus 1 = 0$
$(x+1) \ominus 1 = x$

In the same manner, for the so called signum function sg(x)

$$\text{sg(x)}= \left\{ \begin{array}{lll} 0 & if & x = 0 \\ 1 & if & x \neq 0 \end{array} \right\}$$

We take $f(\mathbf{x}) = 0$ and $g(\mathbf{x}, y, z) = 1$

sg(0)=0
sg(x+1)=1

2.2 Substitution

Theorem 2 *Let $x=(x_1, x_2, ..., x_n)$ and suppose $f(y_1,y_2,\ldots,y_k)$ and $g_1(\boldsymbol{x}),g_2(\boldsymbol{x}),\ldots,g_k(\boldsymbol{x})$ are computable functions. Then the function*
$h(\boldsymbol{x},y) = f(g_1(\boldsymbol{x}),g_2(\boldsymbol{x}),\ldots,g_k(\boldsymbol{x}))$
is computable.

Proof Let F be the program that computes f and $G_1, G_2, \ldots, G_k$ compute $g_1,g_2,\ldots,g_k$. In order to clearify the possibility of building one program in spite of just having one row of registers, first compute $g_1,g_2,\ldots,g_k$ and store the results in the first, second, ..., kth register not affected by any of the programs. Now use F to compute f and store the result in R_1.

Now we have that, by substitution|x-y|is computable since
|x-y| = (x $\ominus y$)+($y \ominus x$)

2.3 An URM that decides a^nb^n

The function we will proceed to compute has input strings $\omega \in \{a, b, ...\}^+$ and the element mapped to one should be the elements from the language a^nb^n (all else are mapped to zero). Let us start with encoding each string as a natural number, exchanging each a's with 1's and b's with 0's so that we have the language 1^n0^n. Now each string belonging to $\{a, b, ...\}^+$ has a unique representation. Let the natural number x be stored in the first register, R_1. Let F be a program that computes $div(x, y)$, the indicator function for 'is x dividable by y', mapping an accepting answer to one and a rejecting answer to zero. Let G be a program that computes $qt(x, y)$, producing the quotient when x is divided by y. Let H be a program that computes $|x - 1|$, in order to count down whatever the sucsessor function has been counting up (see figure 2.1).

Since we will show that F, G and H are finite, we know that there is a register R_p which is unaffected by each program P, denoted $\rho(P)$. We use the output convention that the output of F is stored in this p=$max(\rho(F), \rho(G), \rho(H))$. The program starts with letting p+2 store a 1 and p+3 a 0 and p+4 a 10 for reference, using the successor function. Now the rest of the program decides if $\omega \in a^nb^n$ and stores the output in R_1, accepting ω with a 1 and rejecting with a 0. See further explanation in the figure 2.1.

1 F[1,p+4 → p]
2 J(p,p+3,7)
3 S(p+1)
4 G[1,p+4 → 1]
5 F[1,p+4 → p]
6 J(1,1,1)

7 H[p+1 → p+1]
8 G[1,p+4 → 1]
9 J(p+1,p+3,13)
10 F[1,p+4 → p]
11 J(p,p+3,7)
12 J(1,1,14)
13 J(1,p+3,15)
14 Z(p+2)
15 T(p+2,1)

The preceding program is not the shortest possible but it serves the purpose of simplifying the transitions to the cellular automata described in the next chapter, which will process the input in a similar way.

To conclude the case study of computability of the indicator function of $a^n b^n$, we need to show that the programs div(x, y) and qt(x,y) are computable. We assume that rm(x,y), that is the reminder when y is divided by x and with rm(0,y)=y, is computable. Then qt(x,y) and div(x,y) are computable.

Proof Div(x,y)=$\overline{sg}(rm(x, y))$ is computable by substitution. We write out the recursive step in the definition of qt(x,y):

$$qt(x, y+1) = \left\{ \begin{array}{ll} qt(x,y)+1 & if \quad rm(x,y)+1 = x \\ qt(x,y) & if \quad rm(x,y)+1 \neq x \end{array} \right\}$$

$$qt(0, y) = 0$$
$$qt(x, y+1) = qt(x, y) + \overline{sg}(|x - (rm(x, y) + 1)|)$$

Since we have shown a recursive definition from other computable functions, qt(x,y) is computable. In order to put this example into perspective we conclude that computable functions are functions which can be solved by algorithms, for instance the algorithms represented as programs that can be built from the four elementary instructions in the URM-framework. Using only definition by recursion and substitution, the class of **primitive recursive functions** can be generated. Interestingly, primitive recursive functions are the indicator functions of a type of formal languages called regular languages (Davis, Weyuker & Sigal 1994). These can be parsed by the finite state machine, which we define in the next chapter. However, all computable functions are not primitive recursive. To see this we will use Cantor's diagonal process following Lewis and Papadimitriou (1998).

As we will see also in the case of regular languages, the key point here is enumeration. Each primitive function can be specified with the finite set of symbols denoting basic functions (instructions in the URM-case), and definition by recursion and substitution. Primitive recursive functions can thus be ordered in lexicographic order. Now, let

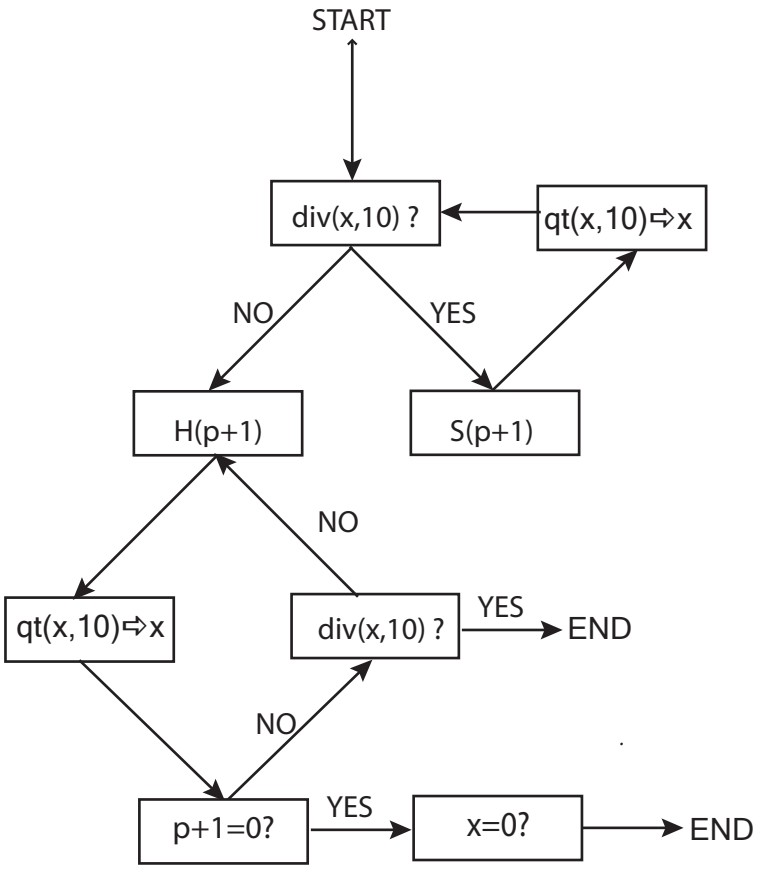

Figure 2.1: Suppose we have a finite $\omega \in \{1,0\}^+$. In the initial state ω, coded as the natural number x, is in R_1. In the first loop the URM test if the last digit is a 0, erase it and increase a counting register R_{p+1} by one. When the last digit is a 1, it will proceed to the next loop without the possibility of going back to erase 0's. Here, 1's are erased and the counting register is instead decreased by one. As soon as an additional 0 is encountered, the machine rejects the input. The URM tests if the counter is zero each turn. When it is, there should be no remains of the string, otherwise the string is rejected. Thus ω must have a number of consecutive 1's followed by the same number of consecutive 0's to be accepted. This is a way to implement a push-down stack memory, a memory structure which is intimately connected with context-free languages.

$f_1, f_2, f_3, \ldots,$

be the list of all primitive recursive functions. Construct g(n), as the function specified by $f_n(n) + 1$. Clearly, g(n) is computable since we have given an algorithm to construct it, with substitution from the successor function, but since it is different from each primitive recursion function for at least one n, it is not primitive recursive.

Chapter 3

Automata and grammars

We wish to deepen the discussion about computation of the indicator function for the language $a^n b^n$ which we have shown to be computable in the URM-framework. In the following section, we describe the interesting features of this language from the perspective of language recognition devices and grammars. The conclusion is that $a^n b^n$ is a context-free grammar, but not regular. Grammars and automata mechanisms are at the core of traditional linguistic theories of natural language.

3.1 Automata

Automata are abstract devices which compute functions of the input delivered on an input tape. According to the described analogy between languages and functions, automata are also language recognition devices. The language accepted by the automaton M is denoted L(M). Generally, we can compare the computational expressivity or complexity of automata by the class of languages that they can recognize. When comparing the automata, it is important to distinguish between the machine complexity and the complexity of its memory organization (Petersson, 2005). In the first example the automaton have no external memory, the second can have finite memory. Having finite memory is a restriction needed whenever we assume space limitation, as with the physical systems of bird or human cortex. It should also be clear that a finite memory makes real number processing impossible, since it is not possible to represent the real numbers in the memory. Although models of cognition using real numbers might be important to develop useful concepts, any finite cognitive system can only be an approximately close to such a model, a property that we call finite precision computing.

Definition 2 *A finite automaton, or finite state machine is a quintuple* $M = (Q, \sum, q_I, q_H, f)$ *Where* Q *is a finite set of states,* $\sum$ *is a input alphabet,* q_I *and* q_H *are the initial and halting states, and* f *is a transition function*
$f : Q \times \sum \rightarrow Q$

In order to emphasize similarities between the automata, we use the convention of a single halting state instead of a set $F \subseteq Q$ of final states used in some definitions. Equivalent **input output maps**, that is the total function which associates an element from the output set to every element of the input set, are easy to construct by adding a single halting state with transitions from all the final states given the empty string as an input.

Definition 3 *A push-down automaton is a 6-tuple* $M = (Q, \sum, \Gamma, q_I, q_H, f)$ *Where Q is a finite set of states,* $\sum$*, is a input alphabet,* Γ *is the stack alphabet,* q_I *and* q_H *are the initial and halting states, and f is a transition function* $f : Q \times \sum \times \Gamma^* \rightarrow Q \times \Gamma^*$ *(where both the range and domain are fininte sets)*

The **stack** is the first example of external memory. Communication between the finite control part of the automata and the stack is done by two operations, pushing symbols down to the stack, thus adding a new top element, or popping, which erase the top element.

Definition 4 *A p-stack machine is defined by a (p+4)-tuple* $(Q, q_I, q_H, \theta_0, \theta_1, \theta_2, \ldots, \theta_p)$

Where Q, q_I *and* q_H *are as above,* θ_0 *map configurations, that is a combination of a state and an input, to Q (intuitively, encoding the transitions).* θ_i*, i=1,...,p, map the configurations to a stack operation on stack i. The stacks are here unbounded.*

A p-stack machine with two or more stacks is computationally universal, i.e. it has the same computational power as a Turing machine. However, we chose to introduce the p-stack machine in order to make the input mode clear when describing how the machine will compute $a^n b^n$. The input will be stored in one stack, which is then popped.

3.1.1 A P-stack machine computing $a^n b^n$

In this example, two states, Q = $\{q_I, q_H\}$, and two stacks are necessary and sufficient for M to decide the $a^n b^n$, here encoded as $0^n 1^n$. Mapping the input symbols to 0's and 1's is relevant in mimicking sensory organization as described with a number sensors being on or off. The configurations are here encoded as a state, that can be thought of as the compound of the state of the central processing unit and the state of the stacks. We also introduce some metadata of the stacks, an empty stack predicate encoding an empty stack with a 0 and a non-empty stack with a 1. This is to avoid introducing a third symbol in Γ denoting an empty stack.

Two states are needed from the definition and one stack is insufficient since

we are using the input convention that ω is encoded in one stack in the initial state. We observe that with this particular input convention and only one stack, the 1-stack machine becomes equivalent to a finite state machine (FSM), having no external memory. In general, context-free grammars like 0^n1^n can not be decided or recognized by finite state machines. This will be proven below. In brief however, this is because the FSM has a finite number of states and to parse a string longer then the number of states (such a string can be picked from L(G)) one state has to be visited twice. Since the FSM had no external memory, there is nothing that can distinguish the first time this state is visited from the second. Thus, the loop created could be traveled any number of times without the FSM changing its output. The only grammars that allow arbitrary recurrences of at least one substring, concatenated in a right phrase linear manner (i.e. adding the recurrent pattern on the right) are the regular grammars, as we will see below.

θ_0
$q_I \times (0,0,1,0) \longmapsto q_I$
$q_I \times (0,0,1,1) \longmapsto q_I$
$q_I \times (1,0,1,1) \longmapsto q_H$
$q_H \times (1,0,1,1) \longmapsto q_H$
$q_H \times (0,0,0,0) \longmapsto q_H$
$q_I \times$ all other inputs $\longmapsto q_H$
$q_H \times$ all other inputs $\longmapsto q_H$

As seen in the table above, the input is encoded as a vector with four entries (top element stack 1, top element stack 2, empty stack predicate stack 1 and empty stack predicate stack 2). We note that this example is parallel to the visualized URM. The difference is that the interaction with the registers, here pooled into levels of a stack, are explicitly modeled as numerical operations on numbers. We want to encode the time evolution of the stack itself as a row of cleverly chosen numbers and stack operations then correspond to a certain computation with this number as an input. Specifically, the stack operations are encoded vectors which we will use later in constructing a neural network. For a fully worked out example of the stack-dynamics see the appendix.

It is clear how closely the URM-framework and the p-stack machine are related and thus we have bridged the gap between, mathematics and theoretical linguistics, computability theory and automata theory. These two descriptions support each other in relation to thinking about applications, the URM-framework being closer to computer science and the automata theory more commonly used in discussions about cognitive function. The perhaps most popular automaton is the Turing machine defined below.

Definition 5 *A Turing machine (TM) is a A Turing machine is a 7-tuple*

$(Q, \sum, \Gamma, b, q_I, q_H, f)$

Where Q, q_I and q_H are as above. Γ is here called the tape alphabet and b is an element in Γ called the blank symbol (the only symbol allowed to occur on the tape infinitely often at any step during the computation). $\sum$ is a subset Γ not including b and called the input alphabet. Finally f is the partial function

$f : Q \times \Gamma, \rightarrow Q \times \Gamma \times \{L, R\}$

In the visualization of the TM, L, R corresponds to moving the tape, alternatively the tape head, left or right.

In terms of computational strength, Turing machines are stronger then PDA's which in turn are stronger then FSM's. This is because of the external unlimited memory device provided in the Turing architecture and PDA's. Interestingly, no abstract digital device can have more capabilities then Turing machines, a result known as the **Church-Turing thesis**.

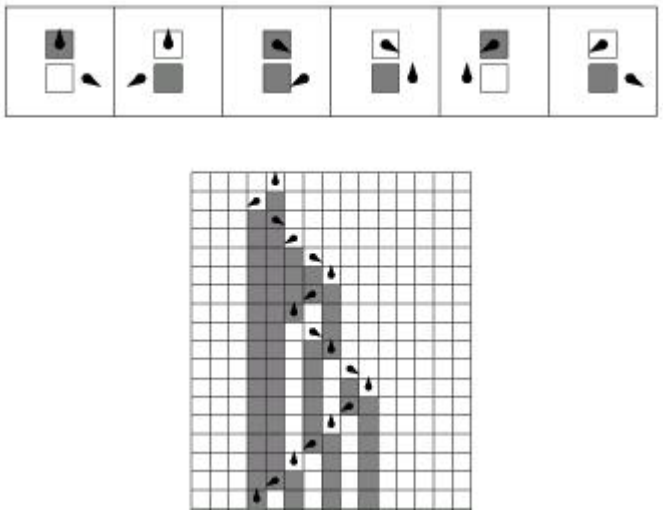

Figure 3.1: A TM with three states and a tape alphabet of black and white shading. The instructions have two rows. The first one is depicting each state as the orientation of the pointing tape head and the tape alphabet as black or white shading. The other one is indicating if the TM will shade the underlying square or not, and if the tape head will move to the left or to the right and which orientation it will take in the next step. Notice that the instructions are on the above mentioned form $f : Q \times \Gamma, \rightarrow Q \times \Gamma \times \{L, R\}$

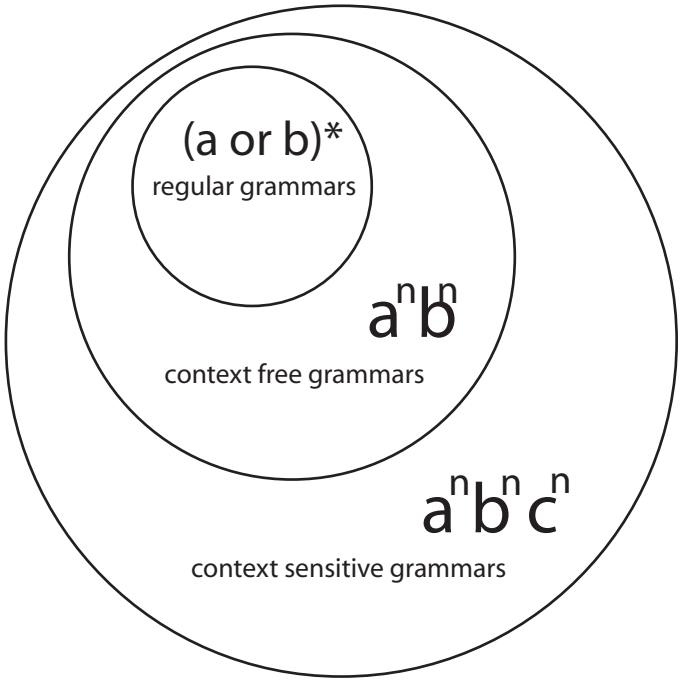

Figure 3.2: Venn diagram of the hierarchy of grammars, with examples.

3.2 Grammars

Definition 6 *A **grammar** or **context-sensitive grammar** is a quadruple* $G = (V, \sum, R, S)$ *where* V *is an alphabet,* $\sum$ *is a set of terminal symbols,* S *is the start symbol which is a member of* $(V - \sum)$*,* R *is the set of rules, a finite subset of* $(V^*(V - \sum)V^*) \times (V^*)$

If all the rules of G are of the form $(V - \sum) \times (V^*)$ the grammar is called a **context-free grammar**. If all the rules of G are of the form $(V - \sum) \rightarrow a$ or $(V - \sum) \rightarrow aV$ where $a \in \sum$the grammar is called a **regular grammar**.

Since strings are obtained by applying the rules to the start symbol, each grammar determine a set of strings that can generated from its rules. This set we call L(G), the language generated by G. In our example, L(G)=$a^n b^n$ and the grammar can be specified as $G = (V, \sum, R, S)$ where $V = \{S, a, b\}$, $\Sigma = \{a, b\}$ and the rules are
$S \longrightarrow aSb$

$S \longrightarrow e$

This way of writing the rules as productions makes it clear that the crucial difference between the context-free and the context-sensitive grammars is on the left-hand side. The context-free case restricts the left-hand side to single nonterminals. In addition, the right phrase linear structure of the regular grammars, mentioned above, come from the further restriction of rules to $(V - \sum) \rightarrow a$ or $(V - \sum) \rightarrow aV$. It is clear that rules of the second form can create a subset of strings in L(G) with an arbitruary finite number of a's in a row as a substring.

Our language is a specified as a context-free language, since it is generated by a context-free grammar. However, since all regular languages are also context-free, we will proceed to prove that $a^n b^n$ is not regular. The following theorem will be helpful.

Theorem 3 ***(Pumping Lemma)*** *Let L=L(M) where M is a finite state machine with p states. Let $x \in L$ where $|x| \geq p$. Then we can write x= uvw where $v \neq 0$ and $uv^i w \in L$ for all $i = 0, 1, 2, 3, \ldots$*

The simple idea is to use the pigeon-hole principle to see that in parsing x at least one state, say state i, has to be visited more then once. Let u be the part of the string parsed before the first time the FSM goes into state i. Let v be the part of the string parsed between the first and the second time. Now it is clear that in terms of input-output behavior, the machine is not sensitive to the number consecutive v's since it can go through this loop any number of times and still end up in the same place. Let w be the reminding part of the string. Hence, $uv^i w \in L$.

We now show that $a^n b^n$ is not regular to convince the reader that the p-stack machine is indeed the right automaton to chose for implementation.

Being a substring, v is on the form $a^{l_1} b^{l_2}$ where $l_1, l_2 \leq$ n and l_1 or $l_2 > 0$. If both l_1 and $l_2 > 0$ $v^i \notin L$ when $i > 1$, since it contains the illegal substring ba. If one of l_1 and $l_2 > 0$ $v^i \notin L$ when $i > 1$ since starting with an equal number of a's and b's, adding a's or b's exclusively will inevitably produce asymmetry.

Chapter 4

Analog Recurrent Neural Networks

The next step in approaching a biologically relevant model for recognition of $a^n b^n$ is to embody the idea of a finite set of states into neurons as the physical entities being in the states. The goal of this chapter is to show recognition of $a^n b^n$ by **analog recurrent neural networks**, a recent construction from the revived field of analog computation. The whole chapter builds on the three first chapters of Siegelmann's "Neural Networks and Analog Computation: Beyond the Turing limit" where she introduces her model, showing the increasing complexity that arise from integer, rational weights (see below), corresponding to the FSM and the TM architecture, respecticely. Siegelmann also shows that extending the presented model with real weights yields computational strength richer than that of the Turing machine and that in fact, a very rich class of time-discrete analog dynamical systems correspond to the class of ARNNs. However, we limit ourselves to introducing some of the basic concepts.

4.1 Neural Networks

A neural network consists of N processors called neurons and a map $\mathcal{F}$ defining the dynamics of the network. The activation vector x with N components is updated according to $\mathcal{F}$, given the input vector u_j, j=1,...,M. The most general network discussed in this thesis use binary inputs and rational activation values:

$\mathcal{F} : \mathbb{Q}^N \times \{0,1\}^M \rightarrow \mathbb{Q}^N$

Component-wise

$$x_i(t+1) = \sigma\left(\sum_{J=1}^{N} a_{ij}x_j(t) + \sum_{J=1}^{M} b_{ij}u_j(t) + c_i\right) \text{ i=1,...,N}$$

a_{ij}, b_{ij} and c_i are called the weights of the network and σ is the piecewise linear function:

$$\sigma(x) = \left\{ \begin{array}{lll} 0 & if & x < 0 \\ x & if & 0 \leq x \leq 1 \\ 1 & if & x > 1 \end{array} \right\}$$

The given weights constitutes $\mathcal{F}$, u_j is the discretized input, and x_j is the previous state.

4.2 Analog computation

Analog computation is defined as computation in continuous space and time (Siegelmann & Fishman, 1998). The last fifteen years, the field of analog computation has become popular in the attempt to relate computational theories to cognition. The idea is to deduce classical cognitive functions from the state space dynamics of **dynamical systems**.

Definition 7 *A finite-dimensional continuous-time smooth dynamical system can typically be defined by a set of ODEs:*

$$\frac{d\boldsymbol{x}}{dt} = \boldsymbol{F}(\boldsymbol{x}(t)) \tag{4.1}$$

where $\boldsymbol{x}$ is a d-dimensional vector and $\boldsymbol{F}$ a d-dimensional vector function.

The dimensionality of the system corresponds to the number of neurons in the neural network and also the number of coupled equations if we would write equation (4.1) on component form. A particular $\mathbf{x}$ is a state in the d-dimensional state space. The state to which a system flows is the output and the initial condition is the input. The flow itself, represented by a trajectory in state space, is equivalent to what we have called processing, but could be more precisely specifyed as information processing. The class of dynamical systems which has been preferred in the attempt to explain cognition is called dynamical recognizers. These are discrete-time dynamical systems with a given initial starting point in a space $\mathbb{R}^n$, called an alphabet $\sum$. For each symbol in $\sum$, the dynamical recognizer has functions that maps $\mathbb{R}^n \rightarrow \mathbb{R}^n$ and an accepting region in $\mathbb{R}^n$.

We note that the neural network as stated in the previous section is a dicrete time dynamical system and that is also how the particular network will be constructed. The reason is that the process of going from discrete time to continous time is technically consuming and it is not the only, or the most important sense in which an analog system is analog. One can also expect that there is a least time-scale relevant to real neural systems (e.g. refractory period between spikes, which is in the order of microseconds).

Definition 8 *An **analog system** is a system in which:*

1. Real constants are present and influence the macroscopic behavior of the system.
2. Continuity in the dynamics.
3. Continuous time update.
4. Discrete input/outputs: discrete input in order to relate complexity to the classical framework and discrete output that correspond to probing state space with finite precision (that is, the only way it can be probed).

The first condition is indeed typically believed to be a property of all physical systems, but it is a curious fact that we can never measure these constants with infinite precision. Instead, we have to postulate them and see what the calculations yields.

Siegelmann's analog recurrent networks use a finite number of neurons, which can be viewed as analog registers, but infinite precision in the processing (which amounts to an assumption of infinite memory capacity). The similarities to the idea of the Turing machines are clear but needs to be shown in detail, which we do in the special case of $a^n b^n$.

Theorem 4 *Let $\psi : \{0,1\}^+ \to \{0,1\}^+$ be the total function such that for every $\omega \in \{0,1\}^+$, $\psi(\omega) = 1$ iff $\omega \in \{0^n 1^n\}$. This function is computable by a p-stack Turing machine in time $T : N \to N$. Then there exists a network N with rational weights that computes $\psi(\omega)$ in time $4T$.*

We simulate time slowed down by a factor of four since this work consumes less technical detail while still giving the relevant insight to the equivalence between Turing machines and ARNN's.
Let M be a P-stack machine that computes ψ. We shall construct a formal network N that simulate M.

4.3 Analog recurrent neural network simulation

4.3.1 Encoding the stack

We use the stack alphabet $\sum$ = 0,1 and we must encode possibly infinite sequences this by encoding functions. We use the 4-Cantor sets, which can we described as having a finite and an infinite component. The idea is to introduce gaps between the numbers, making a fast retrieval of the most significant bits possible.

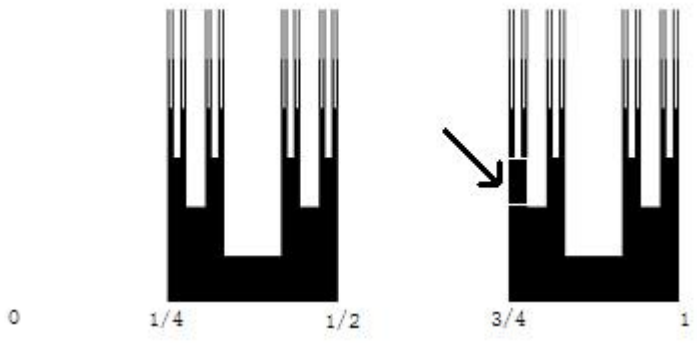

Figure 4.1: The 4-Cantor set. Each black square in the figure corresponds to a stack encoding g. For instance, the stack 0.333_4 corresponds to the square to which the arrow point. Now, reading the top of the stack simply corresponds to two steps. First the linear operation 4g-2 stretch and translate the $[\frac{3}{4},1)$ to [1,2) and $[\frac{1}{4},\frac{1}{2})$ to [-1,0). Now $\sigma(4g-2)$ gives us the top element.

4.3.2 Dynamical system description

Let M be simulated by equations of the following form.

$\beta_{ij} : \{0,1\}^4 \to \{0,1\}$ where i,j $\in \{1,2\}$
$\beta_{ij}(\mathbf{x}) = 1$ where $\mathbf{x} \in \{0,1\}^4$ encodes a transition from state j to state i.
$\gamma^k_{hj} : \{0,1\}^4 \to \{0,1\}$ where h,j $\in \{1,2\}$ and k $\in \{1,2,3,4\}$
$\gamma^k_{hj}(\mathbf{x}) = 1$ where $\mathbf{x} \in \{0,1\}^4$ encodes stack operation k at stack h when in state j.

The stack operations 1-4 correspond to no change, push 0, push 1 and pop respectively. Now it is clear that we can translate the transitions and stack operations used to create the P-stack machine M, into β and γ functions. A worked out example of this translation can be found in the appendix.

We let the states x_i and the stacks g_i be updated as follows:

$$x_i^+ = \sum_{j=1}^{s} \beta_{ij}(a_1,\ldots,a_p,b_1,\ldots,b_p)x_j$$

$$\begin{aligned} g_i^+ &= (\textstyle\sum_{j=1}^{s} \gamma^1_{hj}(a_1,\ldots,a_p,b_1,\ldots,b_p)x_j)g_i \\ &+ (\textstyle\sum_{j=1}^{s} \gamma^1_{hj}(a_1,\ldots,a_p,b_1,\ldots,b_p)x_j)(\tfrac{1}{4}g_i + \tfrac{1}{4}) \\ &+ (\textstyle\sum_{j=1}^{s} \gamma^1_{hj}(a_1,\ldots,a_p,b_1,\ldots,b_p)x_j)(\tfrac{1}{4}g_i + \tfrac{3}{4}) \\ &+ (\textstyle\sum_{j=1}^{s} \gamma^1_{hj}(a_1,\ldots,a_p,b_1,\ldots,b_p)x_j)g_i(4g_i - 2(\sigma 4g_i - 2) - 1) \end{aligned}$$

These equations are on dynamical system form, but we want to proceed by also introducing the biologically inspired transfer function σ which works as a filter on each computation.

4.3.3 Network description

It can be proved that for each function β and γ there exist vectors v_r and scalars c_r such that for each $d_1, d_2, ..., d_t, x \in \{0, 1\}$ and each g $in[0, 1)$
$\beta(d_1, d_2, ..., d_t)x = \sum_{r=1}^{2^t} c_r \sigma(v_r \cdot \mu)$

and

$\beta(d_1, d_2, ..., d_t)xg = \sigma(g + \sum_{r=1}^{2^t} c_r \sigma(v_r \cdot \mu)\text{-}1)$

where we denote $\mu = (1, d_1, d_2, ..., d_t, x)$ and $\cdot$ denotes the inner product. For a worked out example, again see appendix.

These equation are now on the desired neural network form. In the neural network literature, the coupled equations are often divided into layers. This organization provides a time dimension so in our case of a four time slow down, we use four layers to describe the system. Se appendix for figure.

Chapter 5

Simple Recurrent Networks

The simple recurrent network (SRN) architecture was introduced by Elman 1988 as a model for sequential processing. Elman 91 further showed that an SRN can to prediction predict the next word from the training sentences. Christiansen and Chater (1999) showed that the SRN can reproduce phenomena known from human syntax processing. Rodriguez explicitly proposed the architecture as a model for language processing (Rodriguez, 1999b).

A SRN is a neural network with first order recurrent connections. Here recurrence is the possibility for re-entry of input signals or, more importantly, internal states. Adding recurrent connections can be described as adding a temporal component to the model. The SRN architecture is essentially a finite memory **Markov process**, but in the case of finite precision processing of strings of finite length, as in the case of $a^n b^n$ the SRN reduces to a network analogue of the finite state machine.

Definition 9 *Let I be a countable set, a state space (for instance $\mathbb{Q}^N$ as in the definition of a neural network). Now take X to be the current activation value in the neural net, but here the state transitions are interpreted as probabilistic, i.e. X has probability λ_i of being in state $i \in I$. A transition NxN matrix P can thus describe each state transition, each row and column adding up to one. The chain of activation values X_n is called $Markov(\lambda, P)$ if*

$\mathbb{P}(X_0 = i_0) = \lambda_{i_0}$ *and*
$\mathbb{P}(X_{n+1} = i_{n+1} | X_0 = i_0, \ldots, X_n = i_n) = p_{i_n i_{n+1}}$

The last row is interpreted as the next state being dependent on the current state but independent of the earlier states in the chain. However, the Markov chain, the SRN and the FSM can be described as having quite an amount of internal memory, although distributed in the architecture, since many units can have recurrent connections. The connections are usually realized as a line of

neurons called copy units or context units, through which an earlier state of another neuron propagates one unit per time step.

Typically, the network is described as having a small number of **hidden layers** a concept introduced in the neural networks literature going from single layer networks called perceptrons (which can only solve linearly separable problems) to multilayer feedforward networks. This assembly of neurons are not directly connected to the input and not producing the final output, but are rather projecting the input to a higher, say n-dimensional space where the input set can be classified through applying some discretization of state space.

One interesting issue is how to chose the appropriate number of hidden units. One suggestion has been to divide whatever data provided into a training set and a test set (Trappenberg, 2002). Now, increasing the number of hidden units clearly minimize the error the network creates on input taken from the training set, but of course this amount to overfitting. Thus, the way to go is to find an optimal number of hidden units for minimizing the error on the test set.

5.1 Predicting next input

Prediction has gained popularity as a task for computational models of the mind (see, temporal difference learning, the Rescorla-Wagner rule (Dayan & Abbott, 2001)). Stemming from behaviorist ideas of conditioning as the foundation of behavior, reinforcement learning algorithms uses prediction as a way to bridge the gap between the time points of action and reward or punishment. In the SRN case the internal error signal provides additional information of direction. As Elman (Elman, 1990) puts it, the prediction task can also be seen as forcing the network to develop an internal representation of time. Thus, we leave the task of deciding if inputs belong to a language or not for the more biologically relevant task of online understanding of languages from a certain class. However, the described concept of a stack is still needed. The system will not be able to predict when the first b comes, but in order to have predict the beginning of the next string, that is the next a, the system need some analogue to a stack. We will see that this is solved by properties of the dynamics that can be interpreted as a counting mechanism.

5.2 Learning in neural networks

The system is given error signals $e_j = y_j - t_j$ where t_j is the target, that is the wanted output at neuron j. Given input y_i, the output of neuron i, the error is a function of the weights and we can apply the method of gradient descent to

find what point in weight space will minimize this error. Below we derive the back-propagation formula following Haykin 1999.

First, we introduce some important concepts. The **induced local field** at time step n $v_j(n) = \sum w_{ji}(n)y_i(n)$ is the signal that reached neuron j, summing what is left of the inputs when passed through the weights and adding a bias w_{j0}. The output signal $y_j(n) = \sigma_j(v_j(n))$ is the result of applying the activation function to the induced local field. The **error energy** $E = \frac{1}{2}\sum e_j^2$ is the sum of squared error signals taken over all the output neurons. Now, we want to change each weight by reducing the weights in the direction determined by the gradient of the error energy function with respect to the particular weight.

$$\frac{\partial E}{\partial w_{ij}} = \frac{\frac{1}{2}\partial(t_j - y_j)^2}{\partial y_j}\frac{\partial y_j}{\partial w_{ij}} = -(t_j - y_j)\frac{\partial y_j}{\partial v_j}\frac{\partial v_j}{\partial w_{ij}} = -(t_j - y_j)\sigma'(v_j)\frac{\partial v_j}{\partial w_{ij}}$$

$$= -(t_j - y_j)\sigma'(v_j)\frac{\partial y_i w_{ij}}{\partial w_{ij}} = -(t_j - y_j)\sigma'(v_j)y_i$$

We shorten this formula by taking $\delta_j = -\frac{\partial E}{\partial v_j} = -\frac{\partial E}{\partial e_j}\frac{\partial e_j}{\partial y_j}\frac{\partial y_j}{\partial v_j} = e_j\sigma_j'(v_j)$ to be the **local gradient**.

Now, given a learning rate η, determining how big steps will be taken, we end up with the following formula, also incorporating the direction relative to the gradient with the minus sign.

$$\Delta w_{ij} = -\eta\frac{\partial E}{\partial w_{ij}} = -\eta\delta_j y_i$$

However, since the local gradient is dependent on the error signal, which we only have at the neurons in the output layer, we must find a way to propagate the error back throughout the network to find local gradient for all neurons. The idea is quite straightforward, to pass the local gradients from the next layer (we index neurons from this layer k) back through the weights to the hidden neuron j and summing.

$$\delta_j = \sigma_j'(v_j)\sum \delta_k w_{kj}$$

To see this, we start with the earlier definition of the local gradient.

$$\delta_j = -\frac{\partial E}{\partial e_j}\frac{\partial e_j}{\partial y_j}\frac{\partial y_j}{\partial v_j} = -\frac{\partial E}{\partial y_j}\frac{\partial y_j}{\partial v_j} = -\frac{\partial E}{\partial y_j}\sigma_j'(v_j)$$

Now, from differentiating both sides in the definition of the error energy, error signal and output signal, respectively, and substituting into the derived expression for δ_j, we arrive at the desired formula.

$$\delta_j = \sigma_j'(v_j)\sum e_k\frac{\partial e_k}{\partial y_j} = \sigma_j'(v_j)\sum e_k\frac{\partial e_k}{\partial v_k}\frac{\partial v_k}{\partial y_j} = \sigma_j'(v_j)\sum e_k\sigma_k'(v_k)w_{kj} = \sigma_j'(v_j)\sum \delta_k w_{kj}$$

To introduce the concept of momentum, we make a further simplification of this

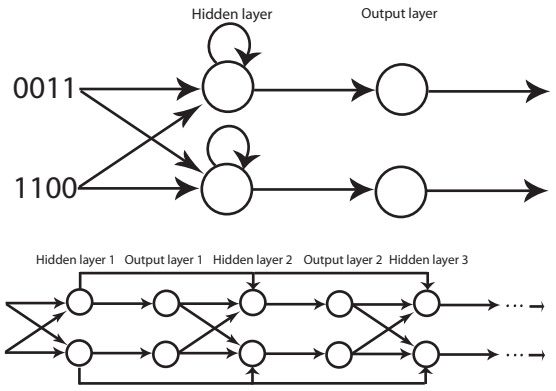

Figure 5.1: The SRN and its unfolded analogue.

derivation to the one-dimensional case, which also serves as good mnemonic. Take the error energy as a function of the weights w. We wish to find the change w by a term Δw, so that the new weight w^* minimizes the error energy. So, doing the first order Taylor expansion around w^* gives us $E(w) = E(w^*) + \Delta w \frac{dE}{dw}$ where $w_2 = w_1 + \Delta w$. The point was that $E(w_2)$ should be smaller than $E(w_1)$, which is fulfilled if we take $\Delta w = \eta \frac{dE}{dw}$ where η is some small positive number. Now, it is quite clear that we can speed up the convergence to the optimal wieght in most cases by introducing a second or third order momemtum term into the learning process.

5.3 Back-propagation through time

When training recurrent networks, the networks N are transformed to a feedforward analogue N^* and then trained with back-propagation. This transformation can be described as a unfolding N into layers of the N^*, so that each time point of N is a layer in N^*, each such layer contains a copy of each neuron in N. In N^*, a connection between neuron j in layer l and neuron i in layer l+1 is made if the corresponding neurons were connected in N.

However, since we now know the target response for neurons in many layers, the weight updates are calculated by back-propragating the error from the output neurons in each layer, alternatively between hidden units in two layers. It is somewhat unclear from the literature how the network should be folded back.

This is also dependent on how many time steps it take to present one pattern, or a batch of patterns (depending on how often the weights are chosen to be updated). One possible solution is to take the average of all the weight updates done to the neuron j and its analogues, an other would be to regard the later layers as more important. We failed to find an appropriate template shedding light on this rather basic issues and thus, we instead proceed by showing the behavior of a SRN with prespecified weights from Rodriguez 1999. It predicts $a^n b^n$ in the sense that b's are predicted throughout the string (since there is no way to know when the first b will come), while the first a of the next string is correctly predicted.

5.4 Description of dynamical system analysis of SRN

The activation function σ is a non-linear sigmoid function. It is important for the learning mechanism that the non-linerity is smooth, so that the function is differentiable. For example, one can use the hyperbolic tangent function

$\sigma(x) = tanh(x)$ with the derivative

$\frac{dtanh(x)}{dx} = \frac{1}{cosh(x)^2} = 1 - tanhx^2$

After the weights are trained, or in our case, with prespecified weights, the activations of the hidden units can be plotted for a stream of input to visualize the behavior of the system.

There will be one vector field for the input a and one for the input b. Since the networks is written as a system of equations, we will linearize it, write it as a matrix and find its eigenvalues λ and eigenvectors v. In turn, we can interpret these as the rate of expansion and contraction of the system, and the axis of this change, respectively.

$|\lambda| < 1$

An attracting fixed point to which the system is contracting.

$|\lambda| > 1$

A repelling fixed point from which the system is expanding.

$|\lambda_j| > 1$ for some non consecutive j $\in \mathbb{Z}$ and if $|\lambda| < 1$ for all other $i \in \mathbb{Z}$

Repelling point called saddle point, giving unstable system behavior. The system is expanding in directions given by the eigenvectors v_j and contracting in

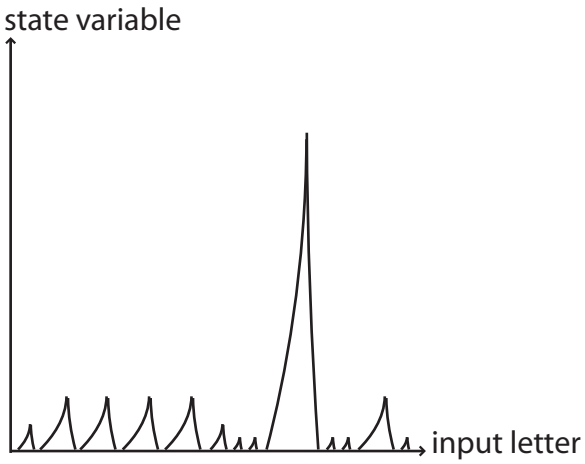

Figure 5.2: Sequence of the sum of state variables for the handcoded solution. Each peak is one string, the higher the peak, the longer the string. We see that the SRN in this case has a accepting region at the x-axis, since the trajectory returns there for in the end of each string.

the directions v_i.

The system is linearized according to the standard technique of linearization (a multidimensional equivalent to analyzing local extreme points by analyzing the derivative). Here, we take the partial derivatives as the fixed point, assembled as a matrix called the Jacobian, which is a linear system.

By finding the fixed points, evaluating the partial derivative at the fixed point and analyzing the linearized system in terms of its eigenvectors and eigenvalues, the two networks described in the figures above can be separated and there different counting behavior can be understood. Although much more mathematics is needed to conclusively understand why the saddle point is the appropriate solution, it is suggestive that it has to do with the dynamical system equivalent of copying over the number of a's. That is, the first network can indeed count, but it can not compare the counting of a's and b's.

In discussing the possibilities and limitations of the SRN to learn natural language, we find the approach taken by Velde (Velde et al, 2004) worth discussion. They encoded words as basis vectors in R^{20} and then trained the network on about 150 000 sentences consisting of about 850 000 items. This is a training set comparable to Elman's seminal work from 1991. However, the SRN could not generalize from the training set, in the sense of, for example, understanding *boy sees girl* and *dog hears cat* and even more complex sentences present in the training set, but not *boy hears girl*. This points to some kind of overfitting to the data-set. Apparently, the SRN has failed to introduce a representation for word categories that would group the examples in such way that generalization was trivial. Though, is it hard to say if this is just a negative result (for instance a result of the particular training set and regime) or if the behavior of the SRN in this case is inherent to the architecture. One way to answer this question would be to formalize the SRN framework and the different levels of the input. In any case, it is not evident that the SRN has to be able to generalize to be considered a relevant model. In principle, a big enough SRN could encode each transition as a level, and then an additional SRN could be working as a meta system on this SRN, encoding the transitions at the next level of generalization.

Many have prosed formalizations of the SRN framework as an important future task for the field. Rodriguez writes "one would like to have a formal analysis using dynamical systems theory to specify how frequency information in the input and the computational properties of an SRN determine its abilities to process linguistic sequences". Levelt has criticized the standard approach, which he describes as demonstrating that networks can represent some domain of knowledge by showing that this domain of knowledge can be taught to the network. What he envisions is instead a formal theory of learnability "Hornik

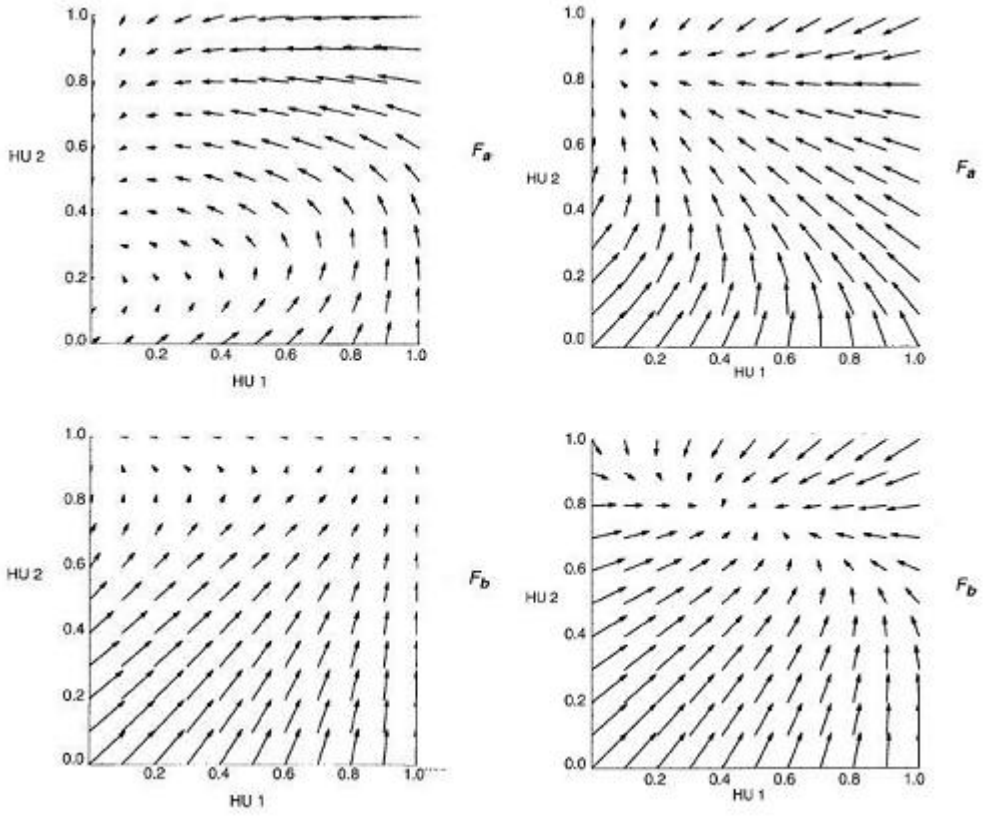

Figure 5.3: Adopted from Rodriguez (1999). Here, the two vectors fields F_a (input a) and F_b (input b) are shown for two networks having trained with different learning rates. The two figures on the left correspond to a network that failed to generalize. The two figures on the right to a second network, that where presented with fewer sweeps of the input, but other learning rates and initial weights, so that is finds a solution that generalize from n=11 to n=16. By varying learning rate and the number of sweeps, Rodriguez found that the crucial difference was the initial weights. The lower left panel is misleading, since it looks like F_b developed an attracting point. At higher resolution, this point turns out to be a saddle point.

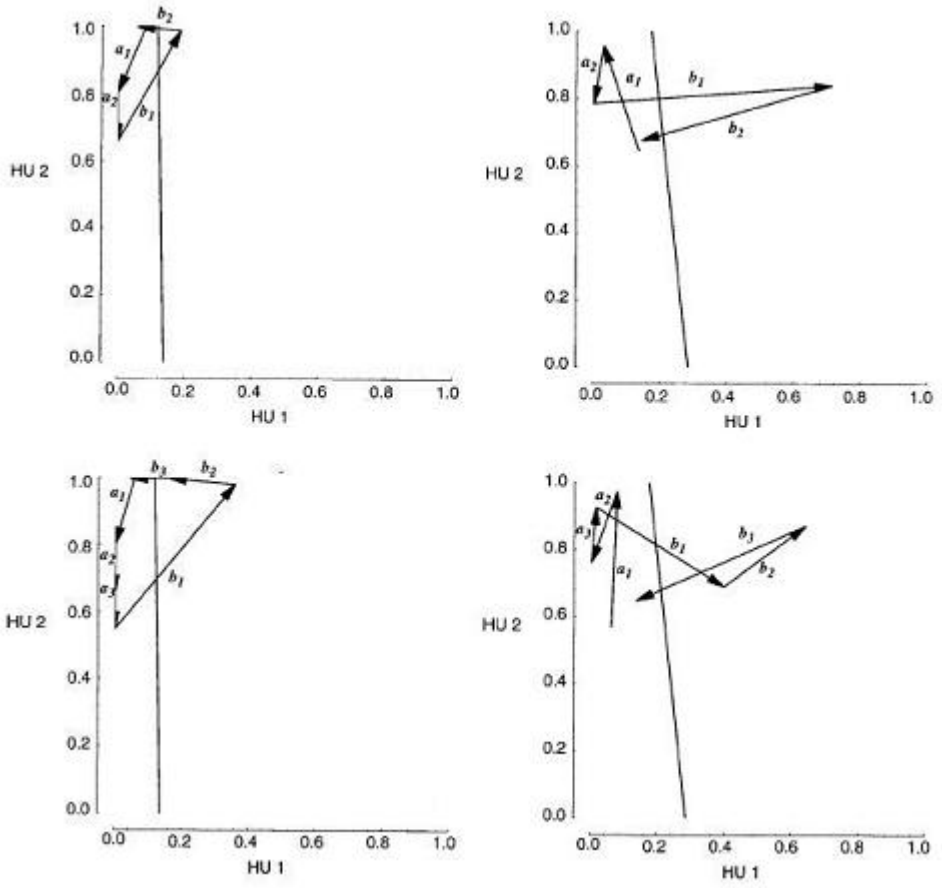

Figure 5.4: Adopted from Rodriguez (1999). Each vector is representing one time step, that is one input, going from the first pair of hidden unit activations to the next. Thus, the top figures have string length n=2 and the bottom figures n=3. See how the two vector fields are combined into the behavior of the system. Here we see that the left network tried a more intuitive solution to the counting problem, but in a sense it is overfitted to the data so that is fails to generalize.

et al.'s theorems on the generative power of networks could form the starting point".

Chapter 6

Conclusion

Cognitive neuroscience approaches the human brain as a cognitive system: a system that functionally can be conceptualized in terms of information processing. In general, we consider a physical or biophysical system as an information processing device when a subclass of its physical states can be viewed as cognitive/representational and when transitions between these can be conceptualized as a process, operating on these states by implementing well-defined operations on the representational structures. Information processing, (i.e., the state transitions) can thus be conceptualized as trajectories in a suitable state-space.

It is possible that the brain has implemented a stack as in PDAs or ARNNs, but this stack is then highly likely to be finite. Thus, at this level of abstraction, both of these models can be reduced to a finite state machine while the SRN can be regarded as a time discrete possibly analog (if using infinite precision processing) network version of the same. We have show that the SRN behaves like dynamical recognizer in the case of a formal language and that standard methods from dynamical system theory can be useful in analyzing how these networks can be said to develop cognitive processing, in a very general sense. The language of mathematics and dynamical systems thus provides a first approximation of what a cognitive information process is: categorization of and computation on the multidimensional input stream by discretizing and making transistions through state space, respectively. The output of the process is the generation of a motor response.

The framework of classical cognitive science and artificial intelligence field assumes that information is coded by structured representations or data structures and that cognitive processing is accomplished by the execution of algorithmic operations or rules on the basic representations such as symbols making up compositionally structured representations (Newell & Simon, 1976). This processing paradigm suggests that cognitive phenomena can be modeled within the framework of Church-Turing computability and effectively takes the view that isomorphic models of cognition can be found within this framework (cf. e.g.,

Cutland, 1980; Davis, Weyuker & Sigal, 1994; Lewis & Papadimitriou, 1998). Language modeling in theoretical linguistics and psycholinguistics represents one example in which the classical framework has served (reasonably) well and all common formal language models can be described within the classical framework (cf. e.g., Partee, ter Meulen, & Wall, 1990).

The perhaps most closely related application of this thesis is in artificial grammar learning, "which is a relevant model for aspects of language learning in infants, exploring species differences in learning and second-language learning in adults" (Petersson 2004, cf., Gomez & Gerken 2000, Friederici 2002). It is also a promising model for investigating adults with learning pathology. Petersson, Grenholm and Forkstam (2005) showed that the SRN can learn a simple artificial grammar and interestingly, they used principle component analysis to extract the grammar from the state space dynamics of the network. This shows that SRN and similar extended simulations are relevant and perhaps also becoming popular in cognitive science. If robust phenomena can be observed at this abstract level, it might be possible to rule out which part of the variation is due to different kind of surface structure of natural languages when conducting empirical experiments. There is a potential for cross-fertilization in the sense that besides biological inspiration, the simulations could also be inspired by more transient cognitive phenomena. Though, a more modest next step would be to continue introducing the more suitable biological constraints. For instance one could use spiking (pulsed) neural networks that take the timing of the input (presented as a time series of spikes) into account. The mathematically oriented research question at hand would then be if there are similarities between the state space dynamics of abstract and the more realistic models. If so, it might be important to introduce concepts to describe these invariants.

Coming back to the European Starlings, it can not be excluded that the appropriate way to interpret the successful behavior of the birds is that they actually aquired the "grammar", even if it was through some kind of counting mechanism. At least if we take the formal analysis of what that might mean into account. Birds and humans could have implemented similar mechanisms for trying to solve the problem of recognizing or predicting the language $a^n b^n$.

References and bibliography

1. Cohen, D. (1997). *Introduction to computer theory* (2nd ed.), John Wiley & Sons, Inc.

2. Cutland N.J (1980). *Computability*, Cambridge university Press (chapter 1-3)

3. Davis, M., Weyuker, E., Sigal, R. (1994) *Computability, Complexity And Languages: Fundamentals of Theoretical Computer Science*, Elsevier Science & Technology

4. Dayan, P. Abbott, L. F. (2001) *Theoretical Neuroscience: Computational and Mathematical Modeling of Neural Systems*, MIT press.

5. Elman, J. L. (1990). Finding structure in time. *Cognitive Science*, 14:179-211.

6. Elman, J. L. (1991) Distributed representations, simple recurrent networks, and grammatical structure. *Machine Learning*, 7:195–224.

7. Fitch, W. T., Hauser, M. D., (2004) Computational Constraints on Syntactic Processing in a Nonhuman Primate, *Science* 303, 377-380.

8. Friederici, A.D. , Steinhauer K., Pfeifer E., Brain signatures of artificial language processing, *Proc. Natl. Acad. Sci. USA* 99 (2002) 529534.

9. Gentner, T. Q., Fenn, K. M., Margoliash, D., and Nusbaum, H. C. (2006) Recursive syntactic pattern learning by songbirds. *Nature*, 440:1204–1207

10. Gomez, R.L., Gerken L., (2000) Infant artificial language learning and language acquisition, *Trends in Cognitive Neuroscience* 4 178186.

11. Haykin, S., (1998). *Neural Networks: A Comprehensive Foundation* (2nd ed.), Prentice Hall.

12. Koch, C., (1999) *Biophysics of Computation*, Oxford University Press

13. Levelt, W.J.M., (1990) Are Multilayer Feedforward Networks Effectively Turing Machines? *Psychological Research* 52(2-3): 153-7

14. Lewis H.R., Papadimitriou, C.H. (1998). *Elements fo the theory of computation*, Prentice Hall. (chapter 1-3).

15. Minsky, M. (1967) *Computation: Finite and Infinite Machines*, Prentice Hall.

16. Newell, A., Simon, H. (1976). Computer science as empirical inquiry: Symbols and search. *Commun. Assoc. Comput. Machinery*, 19, 111-126.

17. Norris, J.R., (1997) *Markov Chains*, University of Cambridge

18. Partee, B. H., ter Meulen, A., Wall, R. E. (1990). *Mathematical Methods in Linguistics*. Dordrecht, The Netherlands: Kluwer Academic Publishers.

19. Petersson, K. M., Forkstam, C., Ingvar, M. (2004). Artificial syntactic violations activate Broca's region. *Cognitive Science*, 28, 383-407.

20. Petersson, K. M. (2005). On the relevance of the neurobiological analogue of the finite-state architecture. *Neurocomputing*, 65-66, 825-832.

21. Petersson, K. M., Grenholm, P., Forkstam, C. (2005). Artificial grammar learning and neural networks. *Proc. Cogn. Sci. Soc.*, 1726-1731.

22. Rodriguez, P., (1999a). A recurrent neural network that learns to count. *Connection Science* 11: 5-40.

23. Rodriguez, P., (1999b). Mathematical foundations of simple recurrent neural networks in language processing. Unpublished doctoral dissertation, University of California, San Diego.

24. Rodriguez, P., (2001). Simple recurrent networks learn context-free and context-sensitive languages by counting. *Neural Computation* 13: 2093-2118.

25. Siegelmann, H. T., (1999). *Neural Networks and Analog Computation*, Birkhäuser.

26. Siegelmann, H. T., Shmuel Fishman (1998) *Physica D.* 120, pp 214-235.

27. van der Velde F., van der Voort van der Kleij G., de Kamps M.2 (2004) Lack of combinatorial productivity in language processing with simple recurrent networks. *Connection Science*, 16(1):21-46

28. van der Velde, F. (1993), Is the Brain an Effective Turing Machine or a Finite-state Machine? *Psychological Research* 55(1): 71-79

29. Wells, A., (2005) *Rethinking cognitive computation: Turing and the science of the mind*, Palgrave Macmillan.

30. Trappenberg P. M. (2002) *Fundamentals of Computational Neuroscience*, Oxford University Press.

Appendix

The stack operations on the P-stack machine are listed below

θ_1	θ_2
$q_I \times (0,0,1,0) \longmapsto (4,-2,-1)$ $q_I \times (0,0,1,1) \longmapsto (4,-2,-1)$ $q_I \times (1,0,1,1) \longmapsto (4,-2,-1)$ $q_H \times (1,0,1,1) \longmapsto (4,-2,-1)$ $q_H \times (0,0,0,0) \longmapsto (\frac{3}{4},0,\frac{3}{4})$	$q_I \times (0,0,1,0) \longmapsto (\frac{1}{4},0,\frac{1}{4})$ $q_I \times (0,0,1,1) \longmapsto (\frac{1}{4},0,\frac{1}{4})$ $q_I \times (1,0,1,1) \longmapsto (4,-2,-1)$ $q_H \times (1,0,1,1) \longmapsto (4,-2,-1)$
$q_I \times$ all other inputs $\longmapsto (\frac{1}{4},0,\frac{1}{4})$ $q_H \times$ all other inputs $\longmapsto (\frac{1}{4},0,\frac{1}{4})$	$q_I \times$ all other inputs $\longmapsto (1,0,0)$ $q_H \times$ all other inputs $\longmapsto (1,0,0)$

The idea behind the P-stack machine that decides 0^n1^n is as follows. Suppose a finite $\omega \in \{0,1\}^+$. In the initial state ω is in stack 1. The only way the machine can proceed without halting with rejecting output is to have a arbitruary number of 0's at the top. These will, according to θ_2, be pushed onto stack 2. When the first bunch of 0's are popped, the only way for the machine to proceed is to have a number of 1's at the top. The first one will, according to θ_0, make the machine go into the halting state, without any possibility of turning back to popping 1's without rejecting the string. If this number is smaller then the number of 0's in the previous bunch, whatever comes after will make the machine halt with output no. If this number is larger, the same will happen. Thus, the only way to get to a halt with accepting output is for ω to have a number of consecutive 0's followed by the same number of consecutive 1's.

$(4,-2,-1)$ is here a pop, $(\frac{3}{4},0,\frac{3}{4})$ is push a 1, $(\frac{1}{4},0,\frac{1}{4})$ is push a 0 and $(1,0,0)$ is leaving the stack as it is.

6.1 β and γ functions

We now list the inputs that are mapped to 1 for each function. All inputs that are not referred to are mapped to 0.

q_I	q_H
$\beta_{11}(0,0,1,0)=1$ $\beta_{11}(0,0,1,1)=1$ β_{21}(any other input)$=1$	β_{22}(any input)$=1$

State 1	**State 2**
γ^1_{21}(all inputs not mentioned in other γ^k_{21})$=1$	γ^1_{22}(all inputs not mentioned in other γ^k_{22})$=1$
$\gamma^2_{21}(0,0,1,0)=1$ $\gamma^2_{21}(0,0,1,1)=1$	$\gamma^2_{12}(0,0,0,0)=1$
γ^1_{11}(all inputs not mentioned in other γ^k_{11})$=1$	γ^1_{12}(all inputs not mentioned in other γ^k_{12})$=1$
$\gamma^4_{11}(0,0,1,0)=1$ $\gamma^4_{11}(0,0,1,1)=1$ $\gamma^4_{11}(1,0,1,1)=1$ $\gamma^4_{21}(1,0,1,1)=1$	 $\gamma^4_{12}(1,0,1,1)=1$ $\gamma^4_{22}(1,0,1,1)=1$

Following Siegelmann's constructions showing that such v_r and c_r exist, let v_r be the following vectors, corresponding to each 2^4 (second to fifth element). We also list the terms in eq 3.13 which this corresponds to and marks if this is a valid input in the construction.

v_1	(0,0,0,0,0,1)	c_1	Valid
v_2	(-1,1,0,0,0,1)	c_2d_1	
v_3	(-1,0,1,0,0,1)	c_3d_2	
v_4	(-1,0,0,1,0,1)	c_4d_3	Valid
v_5	(-1,0,0,0,1,1)	c_5d_4	Valid
v_6	(-2,1,1,0,0,1)	$c_6d_1d_2$	
v_7	(-2,1,0,1,0,1)	$c_7d_1d_3$	Valid
v_8	(-2,1,0,0,1,1)	$c_8d_1d_4$	
v_9	(-2,0,1,1,0,1)	$c_9d_2d_3$	
v_{10}	(-2,0,1,0,1,1)	$c_{10}d_2d_4$	Valid
v_{11}	(-2,0,0,1,1,1)	$c_{11}d_3d_4$	Valid
v_{12}	(-3,1,1,1,0,1)	$c_{12}d_1d_2d_3$	
v_{13}	(-3,1,0,1,1,1)	$c_{13}d_1d_3d_4$	Valid
v_{14}	(-3,1,1,0,1,1)	$c_{14}d_1d_2d_4$	
v_{15}	(-3,0,1,1,1,1)	$c_{15}d_2d_3d_4$	Valid
v_{16}	(-4,1,1,1,1,1)	$c_{16}d_1d_2d_3d_4$	Valid

We now list an example of c_r for each β and γ that follows the constuction.

State Transitions	β_{11}	β_{21}	β_{22}
c_1	0	1	1
c_4	1	-1	0
c_5	0	0	0
c_7	0	0	0
c_{10}	0	0	0
c_{11}	0	0	0
c_{13}	0	0	0
c_{15}	0	0	0
c_{16}	0	0	0

State 1	γ^1_{21}	γ^1_{11}	γ^2_{21}	γ^4_{11}	γ^4_{21}
c_1	1	1	0	0	0
c_4	-1	-1	1	1	0
c_5	0	0	0	0	0
c_7	1	1	-1	-1	0
c_{10}	0	0	0	0	0
c_{11}	0	0	0	-1	0
c_{13}	-1	-1	0	2	1
c_{15}	1	1	-1	-1	0
c_{16}	0	0	0	0	-1

State 2	γ^1_{22}	γ^1_{12}	γ^2_{12}	γ^4_{12}	γ^4_{22}
c_1	1	0	1	0	0
c_4	0	1	-1	0	0
c_5	0	1	-1	0	0
c_7	0	0	0	0	0
c_{10}	0	0	0	0	0
c_{11}	0	-1	1	0	0
c_{13}	-1	-1	0	1	1
c_{15}	0	0	0	0	0
c_{16}	1	1	0	-1	-1

6.2 Handcoded SRN

The following piece of MATLAB-code produces figure 5.2.

$\mathrm{w}_h = [0.5 - 5 - 5; -5 - 1 - 5; 1 + 5 - 5];$

$w_o = [0.500; 220; 000];$

$L = [1122311122231112223111222311122231122312312311112222312312311122231231 23];$

$a = [L == 1]; b = [L == 2]; c = [L == 3];$
$PT = [a; b; c];$
$P = PT'; T = [P(2 : end, :); 100]';$

$\mathrm{v}_h = (w_h * PT(:, 1));$
$if v_h > 1$
$y_h = 0;$
$else if v_h < 0$
$y_h = 0;$
$else$
$y_h = v_h;$
end

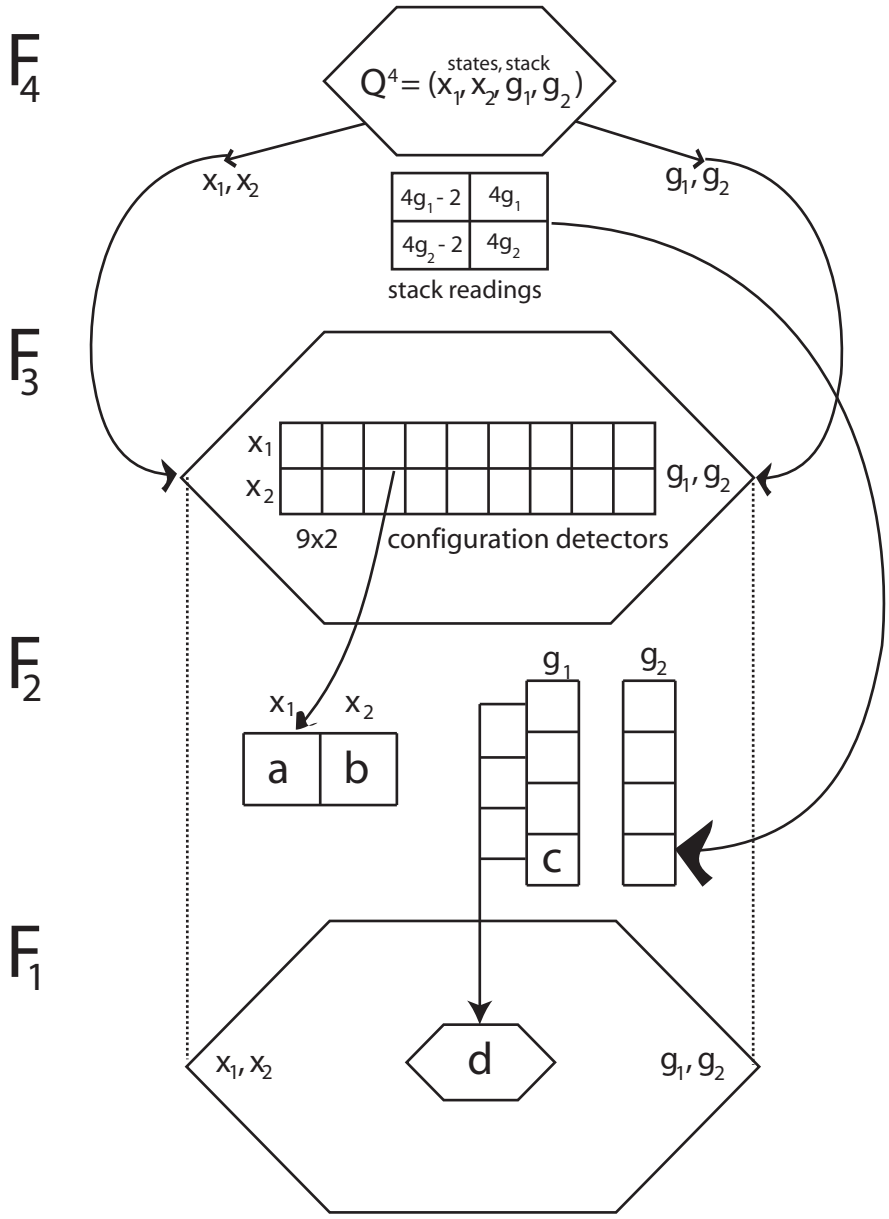

Figure 6.1: The four layers of the network, each computing one step of the update equations described in dynamical system form. The configuration detectors combines states and stack readings, thus corresponding to instructions in the URM case and the tape reader in the turing machine. Here a= $\sigma(\sum c_{r1}\sigma(v_{r1}\mu)$ b= $\sigma(\sum c_{r2}\sigma(v_{r2}\mu)$ c= $\sigma(4g_1 - 2(\sigma(4g_1 - 2)) - 1 + \sum_{j=1}^{2}\sum_{r=1}^{3^p} c_{rj}\sigma(v_r\mu_j) - 1$ d= $\sigma(g_{11} + g_{12} + g_{13} + g_{14})$

$visualize = y_h$

$i = 2$
$whilei < length(L)$

$v_o = w_o * y_h;$
$v_h = (w_h * PT(:, i));$
$v = v_o + v_h;$

$if v > 1$
$y_h = [0; 0; 0];$
$elseif v < 0$
$y_h = [0; 0; 0];$
$else$
$y_h = v;$
end

$visualize = [visualize, y_h]$
$i = i + 1;$

end

$plot(visualize(1, :) + visualize(2, :))$