FOUNDATIONS OF RECURRENT NEURAL NETWORKS

BY HAVA (EVE) TOVA SIEGELMANN

A dissertation submitted to the

Graduate School—New Brunswick

Rutgers, The State University of New Jersey

in partial fulfillment of the requirements

for the degree of

Doctor of Philosophy

Graduate Program in Computer Science

Written under the direction of

Professor Eduardo D. Sontag

and approved by

_____

_____

_____

_____

_____

New Brunswick, New Jersey

October, 1993

ABSTRACT OF THE DISSERTATION

Foundations of Recurrent Neural Networks

by Hava (Eve) Tova Siegelmann, Ph.D.

Dissertation Director: Professor Eduardo D. Sontag

"Artificial neural networks" provide an appealing model of computation. Such networks consist of an interconnection of a number of parallel agents, or "neurons." Each of these receives certain signals as inputs, computes some simple function, and produces a signal as output, which is in turn broadcast to the successive neurons involved in a given computation. Some of the signals originate from outside the network, and act as inputs to the whole system, while some of the output signals are communicated back to the environment and are used to encode the end result of computation. In this dissertation we focus on the "recurrent network" model, in which the underlying graph is not subject to any constraints.

We investigate the computational power of neural nets, taking a classical computer science point of view. We characterize the language recognition power of networks in terms of the types of numbers (constants) utilized as weights. From a mathematical viewpoint, it is natural to consider integer, rational, and real numbers. From the standpoint of computer science, natural classes of formal languages are regular, recursive, and "all languages." We establish a precise correspondence between the mathematical and computing choices. Furthermore, when the computation time of the network is constrained to be polynomial in the input size, the classes recognized by the respective networks are: regular, P, and Analog-P, i.e. P/poly. Among other results described in this thesis are a proper hierarchy of networks using Kolmogorov-complexity characterizations, the imposition of space constraints, and a proposed "Church's thesis of analog computing."

Table of Contents

List of Figures

Chapter 1

The Recurrent Neural Network Model

"Artificial neural networks" provide an appealing model of computation. Such networks consist of an interconnection of a number of parallel agents, or "neurons." Each of these receives signals as inputs, computes some simple function, and produces a signal as output, which is in turn broadcast to the successive neurons involved in a given computation. Some of the signals originate from outside the network and act as inputs to the whole system, while some of the output signals are communicated back to the environment and are used to encode the end result of the computation.

It is possible to classify neural networks according to the architecture of the network. The simplest form of neural network is a *feedforward* network. Such nets consist of multiple layers, where the input of each layer is the output of the predecessor layer. The interconnection graph is acyclic. Note that there is no feedback loop from any layer to its ancestor layer. Therefore, the time of computation is bounded by the number of layers. Long term memory is not supported.

The second model—the *recurrent* neural network model—incorporates memory into the computation. There is no concept of "layers" in this model, as feedback loops are supported. Here, the concept of computation time must be carefully defined. Our goal is to develop mathematical foundations for recurrent networks. Such foundations have been the focus of much research in the feedforward case (see e.g. [ASM93, Bar92, BH89, BR90, CD89, Cyb89, DS92, DDGS93, Fra89, Hor91, HSW90, Jud90, LLPS93, MSS91, Maa93, MS93, MP88, Ros62, Son92b, Son92a, SS91b, Sus92, SW90]) but have yet to be fully developed for the recurrent model. In the sequel, when we use the term "neural network" we always mean recurrent nets.

The model that we study consists of a synchronous network of processors. Its architecture is specified by a general directed graph. The input and output are presented as streams. Input letters are transferred one at a time via M input channels. A similar convention is applied to the output, which is produced as a stream of letters, where each letter is represented by p values. The

Figure 1.1: Feedforward and Recurrent Networks

nodes in the graph are called "neurons." Each neuron updates its activation value by applying a composition of a one-variable nonlinear function with a polynomial function of the activations of all neurons $x_j,\ j = 1,\ldots,N$ and the external inputs $u_k,\ k = 1,\ldots M$. Thus the value of x_i is updated by means of a formula of the following type:

$$x_i^+ \ = \ \sigma\left(\sum_{(|\alpha|+|\beta|)\leq k} a_{i,\alpha,\beta} x^\alpha u^\beta\right) \quad i = 1..N\ , \tag{1.1}$$

where α, β are multiindices, "$|\ |$" denotes their magnitudes (total weights), $k < \infty$, and

$$x^\alpha = x_1^{\alpha_1}\ldots x_N^{\alpha_N}\ , \qquad u^\beta = u_1^{\beta_1}\ldots u_M^{\beta_M}\ .$$

We use a superscript "+" to indicate the value of x_i at the next instant of time. The vector x is referred to as the activation of the network, and the integers N and M are the dimension of the network and the number of input channels, respectively. The output (or read-out map) is given by

$$y = Cx$$

where $C \in \mathbb{R}^{p\times N}$ for some integer p. The integer p is called the number of outputs. As controlled dynamical systems (see [Son90]), networks can be viewed as discrete time systems built by combining delay lines with memory-free elements, each of which performs a nonlinear transformation on

its input. When σ is the identity and all polynomials have degree one, they are the classical linear systems used in engineering. In this work, however, we are primarily interested in nonlinear σ's.

The function σ appearing in the equations is assumed to be the same for all i. It is typically a non-decreasing function, often with a bounded range. In the experimental artificial neural network field it is customary to consider the hyperbolic tangent $\tanh(x)$. This function approximates the sign function when the "gain" γ is large in $\tanh(\gamma x)$. Under a simple change of variables, one obtains the logistic function $\frac{1}{1+e^{-x}}$, also called the "classical sigmoid." Also common in practice is a piecewise linear function, $\pi(x) := x$ if $|x| < 1$ and $\pi(x) = \text{sign}(x)$ otherwise; this is sometimes called a "semilinear" or "saturated linearity" function. Historically, much of the mathematical modeling of neurons in theoretical computer science has been based on a discontinuous activation function, namely, $\text{sign}(x) = x/|x|$ (and -1 or 1 for $x = 0$) or the threshold function $\mathcal{H}(x) = 1$ if $x > 0$, $\mathcal{H}(x) = 0$ otherwise. We believe that it is unreasonable to assume that a physical device may discern sharply between two values which are arbitrarily close; thus we concentrate in this work on continuous activation functions, and, for mathematical convenience, mostly on the above function π.

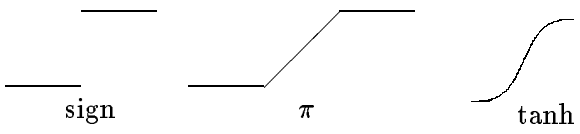

Figure 1.2: Different Functions σ

This thesis is concerned with the development of theoretical foundations for recurrent neural networks. It emphasizes the understanding of the dynamical behaviors that such networks can implement, and how their computational power is influenced by different constraints on weights and variations on architecture. In Section 1.1 below, we explain some of the motivations for this work. We continue in Section 1.2, describing the precise model studied. Section 1.3 provides a summary of previous related work. Section 1.4 describes the organization of this dissertation. Section 1.5 includes a brief overview of several different classical models studied in the field of computational complexity; this is provided as background for those readers not familiar with that field.

1.1 Motivations for Studying This Model

The study of recurrent networks has many different motivations. First of all, they constitute a very powerful model of computation, as shown in this work. They are also capable of approximating rather arbitrary dynamical systems, and this is of use in adaptive control and signal processing applications (see [Son92c], [Mat92], and [PI91]).

Recurrent nets have also been proposed as models of large scale parallel computation, since they are built of potentially many simple processors or "neurons".

One of the primary motivations for their study is as a first approximation of biological neural systems. Some authors have been motivated by this loose analogy to neural systems —hence the terminology— and for this reason the parameters $a^i_{\alpha,\beta}$ are sometimes called "weights" or "synaptic strengths" and σ —taken to be of a sigmoidal type— is called the "activation function". The activation function represents how each neuron x_i responds to its aggregate stimulus. In this context, the outputs $y(t)$ can be thought of as measurements recorded by probes that average the activation values of many neurons.

In speech processing applications and language induction, recurrent net models are used as identification models, and they are fit to experimental data by means of a gradient descent optimization (the so-called "backpropagation" technique) of some cost criterion (see [CSSM89, Elm90, GMC+92, Pol90, WZ89]).

1.1.1 Analog Computation

The work in this thesis could also be seen as exploring a particular approach to analog computation, one based on dynamical systems of the type used in neural networks research.

Electrical circuit implementations of recurrent networks, employing resistively connected networks of n identical nonlinear amplifiers, with the resistor characteristics used to reflect the desired weights, have been proposed as models of analog computers, in particular in the context of constraint satisfaction problems and in content-addressable memory applications (see e.g. [Hop84]).

More generally, in developing the foundations of analog computing, one should be able to model systems in which certain real numbers —corresponding to values of resistances, capacitances, physical constants, and so forth— may not be directly measurable, indeed may not even be computable

real numbers, but they affect the "macroscopic" behavior of the system. For instance, imagine a spring/mass system. The dynamical behavior of this system is influenced by several real valued constants, such as stiffness and friction coefficients. On any finite time interval, one could replace these constants by rational numbers, and the same qualitative behavior is observed, but the long-term characteristics of the system depend on the true values. This use of real numbers, closely associated to phenomena such as chaotic behavior, could be seen as a basic feature of analog computation.

It is interesting to find a class of systems which, on the one hand, is *rich enough* to exhibit behavior that is not captured by digital computation, while still being amenable to useful theoretical analysis, and in particular so that the imposition of resource constraints results in a *nontrivial reduction of computational power.* That this can be done in the context of the models currently used in neural net studies, is especially attractive.

Recurrent networks have a weak property of "robustness" to noise and to implementation error, in the sense that small enough changes in the network would not affect the computation on any finite time interval. This robustness includes changes in the precise form of the activation function, in the weights of the network, and even an error in the update. In classical models of (digital) computation, this type of robustness can not even be properly defined.

Comments on Analog and non-Turing "Computation"

In the recent, very popular –and very controversial– book [Pen89], Penrose has argued that the standard model of computing is not appropriate for modeling true biological intelligence. The author argues that physical processes, evolving at a quantum level, may result in computations which cannot be incorporated in Church's Thesis. It is interesting to point out that the work that we report here does allow for such non-Turing power, while keeping track of computational constraints –and thus embedding a possible answer to Penrose's challenge in more classical computer science. Note that Parberry, in [Par92], also insists that possible non-Turing theories should take account of such constraints, though he suggests a very different approach, namely the use of probabilistic computations within the theory of circuit complexity.

Finally, we remark that human cognition seems to be clearly based on "subsymbolic" or "analog"

components and modes of operation. As pointed out by many authors, in particular in the work of [Mac92], the issue of understanding how macroscopic symbolic behavior arises from such a substrate is one of the most challenging ones in science. Perhaps our work, with its implicit use of infinite precision for internal computations, is not at all relevant to this understanding, because neurons are often taken to be low-precision devices. On the other hand, it is also possible that the precision issue should be understood solely in terms of limitations on observers and more generally interactions with the environment, and in that respect, our model is not deficient, since input and output data are binary.

The rest of this thesis will not include any further speculative remarks on analog computation, and will deal strictly with mathematical results for recurrent nets. However, the "analog efficient Church's thesis": **efficient analog computing = polynomial time computation by recurrent nets** will be strongly suggested by our result.

1.2 The Model

For most of the results in this work, we restrict the general model given by Equation 1.1. Our neural network model is a finite and synchronized system of very simple processors. The activation of each processor is updated according to a certain type of piecewise affine function of the activations (x_j) and inputs (u_j) at the previous instant, with coefficients —also called *weights*— (a_{ij}, b_{ij}, c_i). Each processor's state is updated by an equation of the type

$$x_i(t+1) = \sigma\left(\sum_{j=1}^{N} a_{ij}x_j(t) + \sum_{j=1}^{M} b_{ij}u_j(t) + c_i\right) \quad , \quad i = 1, \ldots, N \tag{1.2}$$

where N is the number of processors and M is the number of external input signals. Note that we are specializing Equation 1.1 to the "first order" case, that is, the polynomials are of degree one. The first-order case is by far the one most commonly encountered in the literature. Unless otherwise stated, the function σ is the simplest possible "sigmoid," namely the saturated-linear function:

$$\sigma(x) := \begin{cases} 0 & \text{if } x < 0 \\ x & \text{if } 0 \leq x \leq 1 \\ 1 & \text{if } x > 1\ . \end{cases} \tag{1.3}$$

This function has appeared in many applications of neural nets (e.g. [Bat91, BGV88, Lip87, ZZZ92]). We use it because theorems are easier to prove when using this particular activation.

For notational simplicity, as mentioned earlier, we often summarize Equation 1.2, writing "$x^+(t)$" instead of "$x(t+1)$" and then dropping arguments t; we also write this in vector form, as

$$x^+ = \sigma(Ax + Bu + c) \tag{1.4}$$

where x is now a vector of size N = number of processors, u is a vector of size M = number of inputs, c is an N-vector, and A and B are, respectively, real matrices of sizes $N \times N$ and $N \times M$. (Now, "σ" denotes application of σ to each coordinate of x.) Of course, one can drop the vector c from this description at the cost of adding a coordinate $x_0 \equiv 1$ and enlarging the matrix A, but it is often useful to have c explicitly, and this allows us to take initial states to be $x = 0$, which corresponds to the intuitive idea that the system is at rest before the first input appears.

As part of the description, one may assume that there is singled out a subset of the N processors, say $x_{i_1}, \ldots, x_{i_p}$; these are the p *output processors*, and they are used to communicate the outputs of the network to the environment. (Generally, the output values are reals in the range $[0, 1]$, but later we constraint them to binary values only.) Thus a net is specified by the data (A, B, c) together with a subset of its nodes. The output is given by $y = Cx$ where C chooses the p output neurons. Indicate by the symbol Δ the time-shift $(\Delta x)(t) = x^+(t) = x(t+1)$, and by Δx the application of Δ to each coordinate of the vector x. When the constant c is included in the enlarged matrix A, Figure 1.3 describes our model.

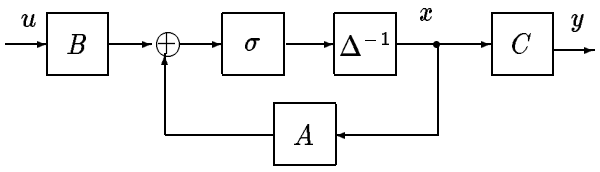

Figure 1.3: Recurrent Network

Input and output consist of streams, that is, one input letter is transferred at each time (via M binary lines) and one output letter is produced at a time (and appears in the output via p binary lines). As opposed to the I/O, the computations inside the network will in general involve

continuous real values. In our further development, we focus only on networks in which both the input and output channels carry only binary data.

The Finite Structure

We wish to emphasize that our networks are built up of *finitely many* processors, whose number *does not increase* with the length of the input. There is a small number of input channels (just two in our main results), into which inputs get presented sequentially. We assume that the structure of the network, including the values of the interconnection weights, does not change in time but rather remains constant. What changes in time are the activation values, or outputs of each processor, which are used in the next iteration. (A synchronous update model is used.) In this sense our model is very "uniform" in contrast with certain models used in the past, including those used in [Hon88], or in the cellular automata literature, which allow the number of units to increase over time and often even the structure to change depending on the length of inputs being presented.

The Simplicity of the Model

We prove in Chapter 8 that if each neuron is allowed to combine activations of the other neurons and external inputs by means of multiplications, besides just linear operations, that is, if one considers instead what are often called *high order* neural nets, as in Equation 1.1, the computational power does not increase. Even further, and perhaps more surprising, no increase in computational power (up to polynomial time) can be achieved by letting the activation function be not necessarily the simple saturated linear one in Equation 1.3, but any function which satisfies certain reasonable assumptions (e.g., the classical sigmoid function $\frac{1}{1+e^{-x}}$). Also, no increase results even if the activation functions are not necessarily identical in the different processors. Because of these later results, we concentrate on the simple model (Equation 1.2) when developing the general theory. (This is analogous to the formalism of digital computation in terms of Turing machines. Even though richer models—RAM machine, etc.—can be shown to be equivalent, proofs are often simplified by staying with the basic Turing model.)

1.3 Previous work

Most of the previous work on recurrent neural networks has focussed on networks of infinite size. As each neuron is itself a processor, such models of infinite power are less interesting for the investigation of computational power, compared to our model which has only a finite number of neurons.

There has been previous work concerned with computability by finite networks, however. The classical result of McCulloch and Pitts ([MP43]) in 1943 (and Kleene [Kle56]) showed how to implement logic gates by threshold networks, and therefore how to simulate finite automata by such nets.

Another related result was due to Pollack[Pol87]. Pollack argued that a certain recurrent net model, which he called a "neuring machine," is universal. The model in [Pol87] consisted of a finite number of neurons of two different kinds, having identity and threshold responses, respectively. The machine was *high-order*, that is, the activations were combined (as in Equation 1.1) using multiplications as opposed to just linear combinations (as in Equation 1.2). Pollack left as an open question establishing if high-order connections are really necessary in order to achieve universality, though he conjectured that they are. Pollack's conjecture was assumed as correct in the neural network literature ([SCLG91, GMC+92]). High order networks (i.e. [CSSM89, Elm90, GMC+92, Pol90, WZ89]) have been used in applications. One motivation often cited for the use of high-order nets was Pollack's conjecture that their computational power is superior to that of linearly interconnected nets. In Chapter 5, we see that no such superiority of computational power exists, at least when formalized in terms of polynomial-time computation.

Work that deals with infinite structure is reported by Hartley and Szu ([HS87]) and by Franklin and Garzon ([FG90] and [GF89]), some of which deals with cellular automata. There one assumes an unbounded number of neurons, as opposed to a *finite number fixed in advance*. See also the work by Hong [Hon88], which deals with nonuniform networks with real weights. In the paper [Wol91], Wolpert studies a class of machines with just *linear* activation functions, and shows that this class is at least as powerful as any Turing Machine (and clearly has super-Turing capabilities as well). It is essential in that model, again, that the number of "neurons" be allowed to be infinite —as a matter of fact, in [Wol91] the number of such units is even uncountable— as the construction relies

on using different neurons to encode different possible tape configurations in Turing Machines.

The work closest to our model when real numbers are utilized as weights is that on real-number-based computation started by Blum, Shub and Smale (see e.g. [BSS89]); we believe that our setup is simpler, and is much more appropriate if one is interested in studying neural networks or distributed processor environments. Also related to our work is the standard computational model of register machines.

1.4 Thesis Organization

This thesis consists of four parts. The first part, consisting of Chapter 2, introduces a high level language, which we call NEL, for NEural Language. We show how to compile NEL programs into recurrent networks while preserving computational time constraints at run time. The development of NEL and the compilation result are of potential applied importance, but we do not pursue applications in this thesis. Instead, we use this language in the sequel theoretically as a proof technique.

The second and most important part of the dissertation deals with the computational power of the model described above. This part extends from Chapters 3 through 7. A formal model of language recognition in recurrent networks is introduced in Chapter 3. In Chapter 4 we prove that networks with integer weights are (like networks with threshold devices) computationally equivalent to finite automata. This result is pretty straightforward, but is needed for completeness. Chapter 5 describes the computational power of networks with rational weights. Specifically, it shows their equivalence to the Turing machine model. (Previous versions of it have appeared in [SS91a] and [SS92].) In Chapter 6, we completely describe the computational power of networks that utilize real weights. (These findings have appeared in [SS93] and will appear in [SSar].) We establish a precise correspondence between this model and a non-uniform circuit model. We study the gap between networks utilizing rational and real weights in Chapter 7, where we reveal an infinite hierarchy in between them. We characterize the weights there in an information-theoretic manner using the notion of Kolmogorov complexity. (This has appeared in [BGSS93].)

The third part of the thesis deals with more general notion of networks. We start in Chapter 8, by proving formally that no increase in computational power—up to polynomial time—is achieved

by considering more complex networks, that is, allowing high-order connections, more complex activation functions, or heterogeneous neurons. As a corollary, we infer the robustness of our model ([SSar]). In Chapter 9 we investigate the power of rational networks under different space constraints, where "space" is defined in terms of the precision allowed in the neurons ([BGSS93]). In Chapter 10 we prove that recurrent networks of first-order neurons, with an activation function whose only known property is having finite and different limits when $x \to \infty$ and $x \to -\infty$, can simulate finite automata. This section concludes with the study of a model in which individual neurons can compute bitwise operations and rational functions. With this addition, networks become equivalent to "second class parallel models" as described in Chapter 11 ([BGSS93]).

In the fourth and final part (Chapter 12), we make some experimental remarks regarding high-order networks. Given a language, we are interested in finding the minimum size of a network required to recognize it. Estimates are obtained using the theory of power series in noncommuting variables. (Part of this chapter has appeared in [SSG92].)

A final chapter summarizes our conclusions and lists several open problems and suggestions for further work.

1.5 Computational Power - Preliminaries

In the science of computing, machines are classified according to the tasks or functions that they are capable of executing. A thorough theory of computation has been developed, in terms of the classification of the different tasks into *classes of tasks*. Two tasks are in the same class if they have roughly the same difficulty. Each class of equi-difficult tasks is associated with a model of a machine that can handle exactly this class.

We start our discussion with an automaton, or sequential machine. This is a device which evolves in time, reacting to external stimuli and in turn affecting its environment through its own actions. In computer science and logic, *Automata Theory* deals with various formalizations of this concept. In this formal sense, neural networks constitute a (very) particular type of automata. It is therefore natural to analyze the information processing and computational power of neural networks through their comparison with the more abstract general models of automata classically studied in computer science. This permits a characterization of neural capabilities in unambiguous

mathematical terms.

The components of actual automata may take many physical forms, such as gears in mechanical devices, relays in electromechanical ones, integrated circuits in modern digital computers, or neurons. The behavior of such an object will depend on the applicable physical principles. From the point of view of automata theory, however, all that is relevant is the identification of a set of *internal states* which characterize the status of the device at a given moment in time, together with the specification of rules of operation which predict the next state on the basis of the current state and the inputs from the environment. Rules for producing output signals may be incorporated into the model as well.

Although the mathematical formalization of automata took place prior to the advent of digital computers, it is useful to think of computers as a paradigm for automata, in order to explain the basic principles. In this paradigm, the state of an automaton, at a given time t, corresponds to the specification of the complete contents of all RAM memory locations as well as of all other variables that can affect the operation of the computer, such as registers and instruction decoders. The symbol $x(t)$ will be used to indicate the state at the time t. At each instant (clock cycle), the state is updated, leading to $x(t+1)$. This update depends on the previous state, as instructed by the program being executed, as well as on external inputs like keyboard strokes and pointing-device clicks. The notation $u(t)$ will be used to summarize the contents of these inputs. (It is mathematically convenient to consider "no input" as a particular type of input.) Thus one postulates an update equation of the type

$$x(t+1) = f(x(t), u(t)) \tag{1.5}$$

for some mapping f. Also at each instant, certain outputs are produced: update of video display, characters sent to printer, and so forth; $y(t)$ symbolizes the total output at time t. (Again, it is convenient to think of "no output" as a particular type of output.) A mapping

$$y(t) = h(x(t)) \tag{1.6}$$

provides the output at time t associated to the internal state at that instant.

Abstractly, an automaton is defined by the above data. As a mathematical object, an *automaton* is simply a quintuple

$$M = (X, U, Y, f, h)$$

consisting of sets X, U, and Y (called respectively the state, input, and output spaces), as well as two functions

$$f : X \times U \to X , \quad h : X \to Y$$

(called the next-state and the output maps, respectively). A *finite automaton* is one for which each of the sets X, U, and Y is finite.

The Finiteness Assumption

It would appear on first thought that it is sufficient in practice to restrict studies to finite automata. After all, only a finite amount of memory is available in any computer. However, even for digital computation, finiteness imposes theoretical constraints which are undesirable when one is interested in the understanding of ultimate computational capabilities. As a trivial illustration, assume that one wishes to design a program which reads an input string of 0's and 1's and, *after* this string ends, displays the same string in its output. A finite automaton cannot accomplish this task, obviously, since the task requires an unbounded amount of memory (unless one knows in advance that the strings to be memorized and repeated will be of no more than a certain predetermined length, in which case enough memory, represented by a certain number of states, can be preallocated for storage). On the other hand, one could certainly write a computer program, in any modern programming language, to perform this task. The program will instruct the computer to write the string into a file as it is being received, to be later retrieved when a special end-of-string symbol $u(t) = \$$ is encountered. This program will execute correctly as long as enough external storage (e.g., in the form of disk drives) is potentially available.

A mathematical model more general than finite automata allows for "external" storage in addition to the information represented by the current "internal" state of the system. This is the *Turing Machine* model, introduced by the English mathematician Alan Turing in 1936, and it forms the basis of most of modern computer science. In a Turing machine, a finite automaton is used as a "control" or main computing unit, but this unit has access to a potentially infinite read/write storage device. The entire system, consisting of the control unit and the storage device, together with the rules that specify access to the storage, can be seen as a particular type of infinite automaton, albeit one with a very special structure. It is widely accepted today by the computer science

community that no possible computing device can be more powerful, except for relative speedups due to more complex instruction sets or parallel computation, than a Turing machine.

Formally, a Turing Machine consists of a finite control and a binary tape, infinite to the right. The tape is accessed by a read-write head. At the beginning of the computation the binary input sequence is written on the left most part of the tape, followed by infinite sequence of blanks (empty symbols).

Figure 1.4: The Turing Machine Model

The machine at every step reads the tape symbol ($\alpha \in \{0, 1, \#\}$) under the head, checks the state of the control ($s \in \{1, 2, \ldots |S|\}$), and executes three operations:

1. Writing a new binary symbol under the head ($\beta \in \{0, 1\}$),
2. Moving the head one step to either right or left (never to the left of the beginning of the tape) ($m \in \{L, R\}$),
3. Changing the state of the control ($s' \in \{1, 2, \ldots |S|\}$).

The transition of the machine, thus, can be summarized by a function $f(\alpha, s) = (\beta, m, s')$. When the control reaches a special state, called the "halting state," the machine stops. Its output is

defined as the binary sequence on the tape. Thus, the I/O map, or the function, computed by a Turing Machine is defined by the binary sequences on its tape before and after the computation.

There are many variants of this basic model that yield the same power. For example, the tape may be infinite in both directions, or the machine may have several tapes, where only one of them is the input/output tape and the rest assist during the computation.

It is possible to think of other "models of computations" that are stronger than a Turing machine, in the sense that they can compute functions (or equivalently, execute tasks) that a digital computer cannot. For example, consider this issue: given a C program and an input to it, decide whether the program will enter an infinite loop and thus will never terminate. If there were a program that could answer such a question, it would be much easier to debug programs. Unfortunately, it has been proven theoretically that no computer in the accepted (Turing) sense can execute such a program. This task is known as the "halting problem". Theoretically, it is formalized as follows: given a Turing Machine and an input sequence, will the machine ever reach the halting state when starting with that input sequence? As we said, no Turing Machine, or no digital computer, can decide such a thing. Other more sophisticated models may be able to solve it. (Most of them currently exist only in research papers.) We will later introduce some of them.

1.5.1 Analog Vs. Digital I/O

In the above two examples, the finite automaton and the Turing Machine, the input to the machine was digital (or discrete), i.e., a sequence of binary numbers, that is, a word in $\{0,1\}^*$. The output was also digital. Other models of analog (continuous) input—like the real numbers π, e—are possible, and also possible are models of analog output.

However, with a few exceptions such as [BSS89], the field of theory of computation has focused by and large on digital models only. Our model computes functions with digital I/O. However, differently from the classical models, we allow for internal states of the machine that are analog (that is, the machine includes real constants that participate in the computation). As I/O is digital in our model, we are able to compare our results with previously existing computer science models of computation. However, analog inputs could be applied to our model as well, and analog output values could be allowed.

1.5.2 Time and Space Constraints

Given an input of length n (i.e., an element of $\{0,1\}^n$), it is reasonable to ask what a machine can compute in time linear or polynomial in n. This allows a quantification of time resources. Space constraints are also meaningful in general. However, since in our model of neural networks we are interested mainly in a finite and fixed number of neurons, we focus mostly on constraining the time resource.

Let $T : \mathbb{N} \to \mathbb{N}$ be a function on natural numbers. We say that a machine M *computes in time* T if for any input sequence $\omega \in \{0,1\}^*$, M halts in not more than $T(|\omega|)$ steps, where $|\cdots|$ denotes the length (number of binary digits) of ω. The class P consists of all functions computable by any Turing Machine in time polynomial in the length of the input, that is, machines that compute in time cn^k, for some real c and integer k.

1.5.3 Language Recognition Vs. Function Computing

Generally, a digital model receives a binary sequence as input and computes a binary sequence as an output. A special case is that where the output is a single binary value rather than a sequence, that is, the model computes a function of the type $L : \{0,1\}^* \mapsto \{0,1\}$. We call such a function "a language" and identify L with the set of those binary input sequences that are mapped into 1. We say in this case that the model *accepts* or *recognizes* the language L.

1.5.4 Other Models of Computation

A few other models of computation will be compared with neural networks. Here we briefly review a few of them. More details will be provided when they are introduced.

Nondeterministic Turing Machine

This popular model describes a Turing Machine which, at any point in the computation, can make one of two choices for the next step. The choices could be about what to write on the tape, how to move the head, or to which state of the finite control to go to. An input sequence ω is said to be accepted by a nondeterministic TM if there is some choice sequence that will make the machine to output 1 on the input ω. Similarly to the class P, the class NP is defined as the class of those

functions that can be computed by some nondeterministic TM in polynomial time. A well-known open research question is whether the classes P and NP coincide (see e.g. [GJ79]).

A Turing Machine That Receives Advice

This is a Turing Machine model that, in addition to its input, receives also another sequence that assists in the computation. For all possible inputs of the same length n, the machine receives the same advice sequence, but different advice is provided for input sequences of different lengths. This gives rise to possible non-uniformity (i.e., non-computability in the classical sense).

Usually, the length of the advice is bounded as a function of the input. Formally,

Definition 1.5.1 Given a class of sets C and a class of bounding functions F, the class C/F is formed by the sets A such that

$$\forall n \; \exists w \; (|w| \leq h(n)) \; \forall x \; (|x| = n)$$
$$x \in A \iff < x, w > \in B$$

where $B \in C$ and $h \in F$. □

The notation "$< x, w >$" is used to indicate that the strings x and w are concatenated, separated by a marker.

A special case is that of a Turing Machine that receives a polynomially long advice and computes in polynomial time. The class obtained in this fashion is called P/poly. This model is equivalent to that of non-uniform families of polynomial size circuits, as will be further explained in Chapter 6. See e.g. [KL82] for background on TMs that take advice.

When exponential advice is allowed, *any* binary language is computable. (This will correspond to a non-uniform family of circuits of exponential size). It is easy to see why every binary language can be recognized in that form: just prepare a table of length 2^n whose entries are all the binary sequences of size n, in the lexicographic order. In each entry (sequence) write the bit 1 if the sequence is in the language, and 0 if it is not. Concatenate all 2^n bits into a sequence and use it as the advice for inputs of length n. This sequence encodes all the required information for accepting or rejecting any input sequence of size n.

Remark 1.5.2 The notion of "advice" should not be confused with that of nondeterminism. In the latter, each input has its own "private" advice, and the issue is the existence of at least one "good, private" advice. For example the class NP is defined as follows. A language A is in NP if there is a language B in P so that

$$\forall x \quad x \in A \iff \exists w_x \ (|w_x| \leq c|x|^k) \ < x, w_x > \in B$$

where c, k are constants.

An Oracle Turing Machine

This model describes a Turing Machine M that has one additional special tape—called the *oracle tape*—and two special states: "ask" and "answered." A language L_o called the *oracle language* is associated with the oracle tape. When M enters the "ask" state, in a unit time the sequence x that was on the oracle tape is erased and only one binary letter is written upon it: 1 if $x \in L_o$ and 0 is $x \notin L_o$. In the same time unit, the machine M is automatically changed into the "answered" state.

Note that the answer to a membership query for L_o takes a unit time even for languages which are not computable by any Turing Machine. For background on oracle TMs, see e.g. [BDG90].

Chapter 2

A Neural Language

In this chapter, we introduce a high-level language which will allow us to easily prove theorems about simulations by neural networks. This chapter can be skipped if one is interested mainly in the results rather than the proof techniques. To understand why the availability of such a language is useful, consider being faced with the following type of problem:

Given a task T, prove that there is a network $\mathcal{N}$ that solves T.

We consider this as an existence problem rather than a constructive one. A possible solution is to explicitly provide such a network and to verify that it indeed executes the task T.

Example 2.0.3 (Parity check) Given a binary input sequence I (of length ≥ 2), compute the binary output sequence b so that b_t is 1 if and only if the number of 1's appearing in the prefix $I_1 \cdots I_{t-2}$ is odd.

A possible network that achieves this is given by the following update equations (where y_3 is the output, and all neurons are initialized to 0):

$$\begin{aligned} y_1^+ &= \sigma(y_1 + y_2 - I) \\ y_2^+ &= \sigma(I - y_1 - y_2) \\ y_3^+ &= \sigma(y_1 + y_2)\,. \end{aligned}$$

Even for this easy task, it is easier to both design and verify a Pascal-like program. A program that computes the same task is given by:

Program Parity (I, b);

Var b: Boolean;

Begin

b =false ;

While (not end of input) **do**

If $(I = 1)$ **then** $b = \text{not}(b)$;

Output(b)

End

A somewhat more complicated task, in which output is not constrained to binary values, is given by the following example:

Example 2.0.4 Given an input sequence I over $\{0, 1, 2, 3\}$, compute the output b that is defined recursively by (assuming $b_0 = 1$):

$$b_t = \begin{cases} \min(17b_{t-1}, 1) & \text{if } I = 0 \\ \min(\frac{1}{7}b_{t-1} + \frac{1}{4}, 1) & \text{if } I = 1 \\ \frac{1}{9}b_{t-1} & \text{if } I = 2 \\ \max(\frac{1}{11}b_{t-1} - \frac{2}{139}, 0) & \text{if } I = 3\ . \end{cases}$$

A network that computes this function is given by the following update equations:

$$\begin{aligned} y_1^+ &= \sigma(I) \\ y_2^+ &= \sigma(1 - y_1) \\ y_3^+ &= \sigma(2 - I) \\ y_4^+ &= \sigma(y_1 + y_3 - 1) \\ y_5^+ &= \sigma(I - 1) \\ y_6^+ &= \sigma(3 - I) \\ y_7^+ &= \sigma(y_5 + y_6 - 1) \\ y_8^+ &= \sigma(I - 2) \\ y_9^+ &= \sigma(y_8) \end{aligned}$$

(In these equations the variables y_2, y_4, y_7 and y_9 assume binary values, with "1" corresponding to the possible cases $I = 0, 1, 2, 3$ respectively.)

$$\begin{aligned}
y_{10}^{+} &= \sigma(17(y_{10} + y_{11} + y_{12} + y_{13}) + 17y_2 - 17) \\
y_{11}^{+} &= \sigma(\tfrac{1}{7}(y_{10} + y_{11} + y_{12} + y_{13}) + \tfrac{1}{4} + 2y_4 - 2) \\
y_{12}^{+} &= \sigma(\tfrac{1}{9}(y_{10} + y_{11} + y_{12} + y_{13}) + y_7 - 1) \\
y_{13}^{+} &= \sigma(\tfrac{1}{11}(y_{10} + y_{11} + y_{12} + y_{13}) - \tfrac{2}{139} + y_9 - 1) \\
y_{14}^{+} &= \sigma(y_{10} + y_{11} + y_{12} + y_{13})
\end{aligned}$$

(The variables y_{10}, y_{11}, y_{12} and y_{13} provide the necessary output values for the cases $I = 0, 1, 2, 3$ respectively, while y_{14} combines them into the final output of the net.) The underlying architecture, again disregarding the numerical values of the weights, is given in Figure 2.1.

Figure 2.1: Underlying Connection Graph of the Above Network

For this task, a program would be much simpler to design:

Program Long (I, b);

Var b: Real;

Begin

$b = 1$ **While** (not end of input) **do**

Case I **of**

$(0,\ \ b = \min(17b, 1))$

$(1,\ \ b = \min(\frac{1}{7}b + \frac{1}{4}, 1))$

$(2,\ \ b = \frac{1}{9}b)$

$(3,\ \ b = \max(\frac{1}{11}b - \frac{2}{139}, 0))$

End

In general, tasks may be composed of a large number of interrelated subtasks. The entire task may thus be highly complex, and designing an appropriate network from scratch becomes infeasible. We formally define a high level language, called **NE**ural **L**angage, that combines features of both Pascal and Lisp. We then prove that any program written in NEL can be executed by a network (and vice versa). Furthermore, we show how to compile a program to a network in such a manner that the running time of the program and the computation time of the associated network are linearly related. We use the language as a proof technique, since, in general, writing programs in a high level language is far easier than designing a network. One could compare this to the difference between writing a program in a high level language and coding in a machine language.

The rest of this chapter is organized into three sections: We start in Section 2.1 with a general, overview description of the language. In Section 2.2, we provide a fully detailed syntactic description of the language: We start with providing the general form of the language and then continuing with data types, expressions, statements, subprograms, and I/O primitives. We end in Section 2.3 with description of how a NEL program can always be compiled into a network. In this last section we also prove the linear relationship between running times of tasks in the two models. To keep this chapter comprehensible, most formalism and details are provided in the appendix to this chapter rather than in the body.

2.1 General Description Of NEL

NEL is a procedural, parallel language. It allows for the subprograms procedure and function. A sequence of commands may either be executed sequentially (*Begin, End*) or in parallel (*Parbegin, Parend*).

- There is a wide range of possible **data types** for constants and variables in NEL. These include:

  1. Basic types:
     (a) Finite sets: Boolean, character, and scalar type.
     (b) Numerical: integer, real, and counter (i.e., an unbounded natural number or 0).
  2. Compound types:
     (a) List - with the operations defined on lists in LISP.
     (b) Stacks - along with the operations *Top, Pop, Push.*
     (c) Sets - along with operations as *union, intersection,* and *set-difference.*
     (d) Records - defined like in the Pascal language.

  For each data type, there are a few associated predefined functions. The language is *strongly typed* in the sense that applying a function that is defined on a particular data type to a different data type may yield an error. Predicates are functions that range into Boolean values. Among the predicates of NEL are *Isempty*(stack), *In*(element, set), *Equals*(set_1, set_2), *Iszero*(counter), and more.

- **Expressions** are defined on the different data types. Examples of expressions are:

  1. $\boxed{\sum_{i=1}^{7} c_i x_i}$ for constants c and either real or integer values of the variables x_i.
  2. $\boxed{(B_1 \text{ And } B_2) \text{ Or } (x > \frac{1}{2})}$ for Boolean values B_1, B_2 and an integer value x.
  3. Function calls, including predefined predicates like *Isnull*(list), *Isempty*(stack), and predefined functions like *Union*(s_1, s_2).

- **Statements** of NEL include (among others):

  1. *Atomic* statements, such as assignments, procedure calls, and I/O statements.
  2. *Compound* statements, such as sequential compound statements (*Begin, End*) or parallel compound statements (*Parbegin, Parend*). Parallelism is constrained to the main program only and is not allowed in subroutines.
  3. *Flow control* statements, which include:

(a) Conditional statements such as *If-then, If-then-else, case*, and *cond.*

(b) Repetition statements such as *while* and *repeat.*

- **Inputs and Outputs (I/O)** appear on special channels and have numerical values.

Next, we provide a few examples of how to compile fragments of NEL programs into subnetworks:

Example 2.1.1 Let M and N assume values in $[0,1]$ and B is a Boolean expression. The conditional statement

If (B) **then** $x = M$

else $x = N$

is simulated by $x^+ = \sigma(\sigma(M + B - 1) + \sigma(N - B))$. □

Example 2.1.2 Let c be a counter and x_1, x_2 be real variables. (The operator Dec(c) decrements the counter, and Dec$(0) = 0$.) Consider the sequence:

$x_1 = x_2 = 1$

read(c)

Repeat

Dec(c)

$x_1 = \frac{1}{2}x_1$

$x_2 = \frac{1}{7}x_2$

Until (**Iszero**(c))

1. A counter variable may assume any natural value or 0. In the network compilation, to each variable c of the counter type, there is associated an activation variable z whose value is

$$z = 1 - 2^{-c} \quad .$$

The counter operations are implemented as follows:

$$\begin{array}{lll} \text{Dec } (c) & \text{is implemented as} & z^+ = \sigma(2z-1)\,, \\ \text{Inc } (c) & \text{is implemented as} & z^+ = \sigma(\tfrac{1}{2}z+\tfrac{1}{2})\,, \\ \text{Iszero } (c) & \text{is implemented as} & z^+ = \sigma(1-2z)\,. \end{array}$$

2. Real variables are implemented as real values in the range $[0,1]$.

The following network simulates the above program, assuming initial values $x'_1 = x'_2 = 1, c' = 1-2^{-c}$:

$$\begin{array}{lll} \text{Computing } x_1: & & x_1'^{+} = \sigma(\tfrac{1}{2}x'_1) \\ \text{Computing } x_2: & & x_2'^{+} = \sigma(\tfrac{1}{7}x'_2) \\ \text{Counter}: & & c'^{+} = \sigma(2c'-1) \\ & & s^+ = \sigma(3-4c'-s') \\ & & s'^{+} = \sigma(3-4c') \\ \text{Output}: & & (x_1^{'}, x_2^{'}, s)\,. \end{array}$$

Here s takes the role of a binary validation neuron which is set to 1 once, implying that the output information is ready (when the counter has the value 1, or 0 if this was its initial value). □

Full details of how to compile any element of NEL (i.e., data-types, expressions, statements and subroutines) appear in Section 2.3.

2.2 Language Syntax

A language is a tool used in order to define the syntax of programs. A program takes given values of input data, manipulates them using predefined constants, stores intermediate data in variables, and produces output data. A program consists of a concatenation of lexical tokens, including identifiers (names), reserved words (such as **begin, case**), numbers, characters ('s'), arithmetic symbols $(+,-)$, and relational symbols $(=,>)$. These tokens are concatenated in a meaningful way, following the strict syntax of the language.

We define a language whose programs can be compiled into a network having an architecture as described in Chapter 1. We define the syntax of the language in a grammar similar to the Backus-Naur Form (BNF), which was developed to describe the Algol programming language. The grammar is a set of production rules, consisting of two kinds of items:

- Non-terminals, which are syntactic categories. They are enclosed by curly brackets.
- Tokens of the language. A special case are the reserved words. These appear in boldface.

A sequence of tokens is a syntactically valid program if and only if it can be derived by the set of productions, starting from the non-terminal "{program}". The notations that we use to describe the NEL language include:

- "::=", to indicate each production.
- "|", to denote "or" when several options of production exist for the same non-terminal.
- "[x]", one or more repetition of x. We use "[x,]" as an abbreviation of $x|[x,]x$, and "[x;]" as an abbreviation of $x|[x;]x$

The general description of the language is as follows (with some comments included):

{program} ::= {program heading};
{block}.

{program heading} ::= **Program** identifier ([{channel identifier},])
{* Explanation of I/O channels is provided is Subsection 2.2.2 *}

{block} ::= {constant definitions}
{type definitions}
{variable declarations}
{procedure and function declarations}
{compound statement}

{constant definitions} ::= {empty} | **Cons** [{constant definition};]

{constant definition} ::= {identifier} = number

{* Constants can be real numbers only *}

{type definitions} ::= {empty} | {**Type** [{type definition};]

{type definition} ::= {type-identifier} = {type}

{* We describe in Subsection 2.2.1 all possible types *}

{variable declarations} ::= {empty} | **Var** [{typed variable list};]

{typed variable list} ::= [{identifier},] : {type-identifier}

{procedure and function declarations} ::= empty |
[{procedure declaration} | {function declaration}]

{procedure declaration} ::= {procedure heading}; {block};

{procedure heading} ::= **procedure** identifier ([{formal parameters},])

{formal parameters} ::= **var** {typed variable list} |
value {typed variable list}

{function declaration} ::= {function heading}; {block};

{function heading} ::= **Function** identifier ([{formal parameters},]): {type-identifier}

{type-identifier} ::= {identifier}

{compound statement} ::= **Begin** [{statement};] **End** |
Parbegin [{statement};] **Parend**

{* Statements are discussed in Subsection 2.3.6 *}

Here we include a discussion about data types and I/O. Appendix 2.3.4 includes details about expressions, statements, and subprograms.

2.2.1 Data Types

Each variable—as well as each constant, parameter, I/O datum, and function—has an associated *data type*. The type specifies the set of values that may be assumed and the operations permissible on those values.

The language NEL supports the following data types:

1. **Finite Ordered Types**

   Variables of such types take values in finite sets that have up to some fixed number of elements "Fix." (The upper bound is fixed, but different for each program. "Fix" is defined as one of the constants of a program.) The elements of the sets are indexed as $\{0, 1, \ldots\}$. The function Ord(element) returns the index of the element. The relational operators $<, >, =, \geq, \leq, \neq$ are defined between any two elements of a finite ordered set. In any of these sets, x is considered smaller than y if **Ord** $(x)<$ **Ord** (y). The following are the possible ordered types:

   (a) Values taken by **Boolean** variables consist of the elements {false, true}. The operations defined on Boolean variables are the unary operator **Not**($\neg$), and the binary operators **And**($\bigvee$) and **Or**($\bigwedge$). The order function is defined as: **Ord**(False) = 0, and **Ord**(True) = 1.

   (b) **Scalar type** is a user defined type. It is introduced either by an explicit definition of the form

   $$\text{type t} = (c_1, c_2, \ldots, c_n)\,,$$

   or by providing the lower and upper bounds of a range of any predefined scalar type or integer. The number of elements in each Scalar type is bounded by the constant Fix of the program. An example of a scalar type declarations is:

   type Weekday = (Monday, Tuesday, Wednesday, Thursday, Friday)
   type Midweek = Tuesday .. Thursday.

   The operations defined on a scalar type are: **Succ**$(c_i) = c_{i+1}$, **Pred**$(c_i) = c_{i-1}$, **Ord**$(c_i) = i$, and relational operators. The operation Succ is not defined for the last element, and Pred is not defined for the first one. The names of the scalar elements in the various

types are all distinct, and in particular they differ from Boolean and char elements. Furthermore, they should not be numerical.

(c) **Char** is a finite ordered set of characters, with the operations **Chr**(i)= c_i, **Ord**(c_i)= i, and relational operators. The function Chr(i) is undefined for a number i outside of the range.

2. **Numerical Values**

(a) **Integer** is the subset of the set of integers, consisting of those integers in a bounded range [minInt, maxInt]. (These variables are defined in each program as constants.) In case of overflow, an integer variable is saturated to one of the extreme possible values. The arithmetic operations defined on integers are $+, -$ between two integer variables, and $*$, multiplication of an integers with an integer constant. (Multiplications among two integer variables are not allowed.) Integers are an ordered set in the sense that the relational operators $<, >, =, \geq, \leq, \neq$ are defined on them.

(b) The **Counter** type consists of the set of all nonnegative integers with the operators **Inc**(c_i) = c_{i+1}, **Dec**(c_i) = c_{i-1}, and the predicate **Iszero**. The operator Dec is not defined on the element 0.

(c) The **Real** type consists of all real numbers in an interval [minReal, maxReal] (that is defined by the constants of the program). The arithmetic operations are like those defined on integers. The relational operators discussed above are not defined here. The only comparison operator defined is the unary operator $\Upsilon(\alpha, \beta, x)$, where α, β are constants so that (minReal $< \alpha < \beta <$ maxReal), and x is a real variable:

$$\Upsilon(\alpha, \beta, x) = \begin{cases} \text{false} & \text{if } x < \alpha \\ \text{undefined} & \text{if } \alpha \leq x \leq \beta \\ \text{true} & \text{if } x > \beta \ . \end{cases}$$

3. **Compound Homogeneous Data Types**

These are user-defined types which include other data types. These types are homogeneous in the sense that they may include elements from one type only.

(a) **List of** T is a chain of elements of the type T, where T is one of the finite ordered types. The operations defined on a list are:

- **Car** ({*list*}) returns the first element in a list (e.g. **Car** (a, b, c) = a).
- **Cdr** ({*list*}) returns the list without its first elements (e.g. **Cdr** (a, b, c) = (b,c)).
- **Cons**({element}, {list}) (where element and list are from the same type T) returns the new list whose car is the element and its cdr is the list (e.g. **Cons**(a, (b c)) is (a b c)).
- The predicate **Isnull** ({list}) implies if the list is empty.

Using these elementary operations, all the list manipulation operations defined in the language LISP ([Ste84], Chapter 15) are well defined here as well. This includes the operations **Nth** (c list), which returns the cth element in the list (c is of type counter); **Length** (list), **Append** (list1, list2) that concatenates two lists, **Reverse** (list), and more. (These last operations are not atomic.) Note that as the lists are "flat" in the sense that lists of lists are not defined in NEL, all of the above operations do not require recursion but rather a simple loop.

(b) **Stack of** T is a list of T (T is a finite ordered set), in which the last element to enter, is the first one to leave and the only one that can be operated on. The operations defined on a stack are:

- **Top**({stack}) returns the top element of the stack.
- **Pop**({stack}) returns the stack without its top element.
- **Push**({element}, {stack}) returns a stack having as its top the given element, followed by the previous stack.
- **Isempty**({stack}) specifies if the stack is empty.

(c) **Set of** T (T is a finite ordered set) is an unordered collection of elements of type T.

- **In**({element}, {set}) is a predicate that implies if an element is in a set.
- **Equals**({set_1}, {set_2}) implies if the sets are equal.
- **Includes**({set_1}, {set_2}) implies if set_1 is included in set_2.
- **Union**, **Intersection**, **Set-difference** receive two sets and return another set after the operation.

4. **Record** is a heterogeneous type, consisting of fixed fields of other previously defined types. More details about records can be found in any Pascal manual, e.g. [Che80].

{type}	::=	{ordered type}| **Counter** | **Real** |
		{compound type} | {record type}
{ordered type}	::=	{finite type} | **Integer**
{finite type}	::=	**Boolean** | **Char** | {scalar type}
{scalar type}	::=	{ordered type identifier} | {value} .. {value} |
		{identifier list}
{value}	::=	{integer value} | {scalar value} (e.g. 3, Monday...)
{compound type}	::=	**List of** {finite type} |
		Stack of {finite type} |
		Set of {finite type}
{record type}	::=	**Record** {field list} **End**
{field list}	:=	[{record section},] |
		{variant part} |
		[{record section},] {variant part}
{record section}	::=	[{field identifier},] : {type} | {empty}
{variant part}	::=	**Case** {identifier}: {ordered type identifier} **of** {variant}
{variant}	::=	[{case label list} : [{field list},] | {empty}
{case label list}	::=	[{ordered value},]

2.2.2 Input and Output

Input is read from synchronous streams of finite-types (given by their Ord) or numerical values, appearing on input channels. An important Boolean predicate defined on each input channel is **Eoc** which signifies the end of the stream that appears on that channel. When the program starts running, all input channels are opened automatically. Output channels are defined similarly, however, each of them is opened via an explicit command in the program.

NEL defines *reactive* programs (see e.g. Henziger, Manna and Pnueli [HMP92]), those having an on-going interaction with their environments (as opposed to transformational systems which

interact in a limited way with humans only). In this sense, NEL defines real-time software. To fully implement NEL, both time concepts and safety properties should be developed. (See more on crucial issues in real-time software in Motus [Mot93].) As we are interested in NEL from a theoretical, rather than implementable, view, we do not fully address these issues.

Full details about the syntax of expressions, statements, and subprograms can be found in Appendix 2.3.4.

2.3 Compilation Of The Language Into A Network

We next describe a compiler which compiles programs written in the language NEL into a neural network of the type described in Chapter 1. The model of compilation we use is batch-oriented. The entire source program has been written and the program will be fully compiled before the programmer can execute the program. The language itself is defined so that:

- The scope and binding of each identifier reference is determined before execution begins.
- The type of an object is determined before execution time (not like APL).
- The language does not allow the program text to be modified at run time (like APL, not like LISP).

The program is measured by its length, both statically and dynamically. The static length of the program is the number of tokens listed in the source code. The dynamic length is the number of atomic commands executed for a given input. We call the first one the *length* of the program, and the second is called the *execution* measure.

Theorem 1 *There is a compiler C that translates each program in the language NEL into a network. The constants (weights) that appear in the network are the same as those of the program, plus several rational small numbers. Furthermore, the running time of the network is O(execution measure).*

We exclude the discussion about the scanner and the semantic pass. These are implemented in standard ways. We rather concentrate on the syntactic part of the compiler.

It is easy to see that for each network, there is a program written in NEL that simulates it. The length of such a program is linear in the number of units of the network, and its constants are the constants of the network. Combining these statements, we can say that programs written in NEL and networks are linearly related.

2.3.1 Input and Output

Given a program P written in NEL with k input channels and l output channels, the network N simulating P has $2k$ input lines and $2l$ output processors. All the input into the network is numerical. Hence, if characters or scalars had to be read in the program, their Ord function is the input to the net.

The input lines are organized in pairs of

$$(\text{Input.data}_i, \text{Input.validation}_i).$$

The "input.data$_i$" line carries the information of channel i. It defaults to "no information" when no data appears on it. (In numerical value, the input equals 0). The input is synchronous in the sense that at each step, the next input datum appears on the line. The "Input.validation" line is a binary line. It is set to 1 when data is being transferred on the data line, and is 0 when no information appears there.

The output processors (lines) adhere to a similar convention: for each output channel i, there are two special processors

$$(\text{Output.data}_i, \text{Output.validation}_i).$$

An output operation into channel i sets the Output.data$_i$ processor to 1 for one step. Otherwise, it is 0. The processor Output.data carries the information that has to be written in the ith channel. It has the "no information" value when no specified value is written into the open channel.

2.3.2 Flow Control

The network operates generally in the following manner: There are N neurons. At each tick of the clock, all neurons are updated with new values. Thus, a network step consists of a parallel execution of N assignments.

When simulating the program on a network, some of the neurons represent variables, and others represent the flow control and other parts of the program. Consider a neuron that represents a specific variable. This variable has to change its activation value only when the program counter points at a command which changes the activation. At other times, the variable is not supposed to change.

To control such a flow over the network, each command in the program is associated with a special neuron, called the "statement counter" neuron. These neurons take Boolean (i.e. binary) values only. When a statement counter neuron is True, the statement is executed. All statement counters together constitute the program counter. Note that this counter has a distributed representation, and that several statement counters may assume the value True simultaneously. The statement counters themselves may have a distributed representation as substatement counters. The number of substatement counters depends on the number of layers required to execute the statement at hand, as will be described below.

Given a statement s_k, the neurons that are involved in its simulation are both those that simulate the statement and those that simulate the distributed representation of the statement counter. For example, assume a statement s_k is executed in l_k layer net. Then the statement has the l_k associated counters:

$$c_k^{(1)}, c_k^{(2)}, \ldots c_k^{(l_k)}$$

which are updated via

$$c_k^{(j)+} = \sigma(c_k^{(j-1)}) \quad j > 1 \ .$$

For $j = 1$, We consider two possibilities:

1. Assume the statement s_k appears in a sequential block (Begin, End) and the next statement is s_{k+1}. The update of the associated $c_{k+1}^{(1)}$ is given by

$$c_{k+1}^{(1)+} = \sigma(c_k^{(l_k)}) \ .$$

2. Assume the statement s_k appears in a parallel block together with p other statements, and the next statement after the Parend command is s_m. In this case, a flag indicating is the statement s_k has been executed, is required. This will be handled by the neuron c_k^{par} that

updates by

$$c_k^{\text{par }+} = \sigma(c_k^{(l_k)} + c_k^{\text{par}} - c_m^{(1)}) .$$

The first counter associated to s_m is updated using this flag, as follows:

$$c_m^{(1)+} = \sigma(c_k^{\text{par}} + \sum_{j=1}^{p} c_j^{\text{par}} - p) .$$

Below we see that each statement, including repetitions, can be decomposed into constant-time statements and the statement goto. Thus, this explanation of compilation holds for any statement. Examples of how to compile atomic statements and their counters into networks are provided below.

2.3.3 Data Types and Expressions

Each variable of any data type (except for a record) is represented via one neuron in the network.

- Boolean values are represented via the numbers $\{0, 1\}$. The logical operations are:

Operation	Network's emulation
Not(x)	$\sigma(1 - x)$
Or(x_1, x_2)	$\sigma(x_1 + x_2)$
And(x_1, x_2)	$\sigma(x_1 + x_2 - 1)$

(2.1)

Relational operations are defined in a straightforward manner: $x > y$ is $\sigma(x - y)$, $x \geq y$ is $\sigma(x - y + 1)$, and $x \neq y$ is $\sigma(\sigma(x - y) + \sigma(y - x))$.

- Scalars: there may be up to Fix different elements in each scalar type. Assume a scalar type with n elements $\{0, 1, \ldots (n-1)\}$. The ith element is represented as

$$\text{scalar } (i, n) \quad \hookrightarrow \quad \frac{2i+1}{2n} . \tag{2.2}$$

Order operations are implemented as follows:

Operation	Network's emulation
Pred(x)	$\sigma(x - \frac{1}{n})$
Succ(x)	$\sigma(x + \frac{1}{n})$
Ord(x)	$\sigma(xn - \frac{1}{2})$

(2.3)

Using this representation, it follows from the fact that n is always bounded by the constant Fix, that the relational operators of scalar type always involve numbers separated by a distance of at least $\frac{1}{2\text{Fix}}$.

Given real numbers a and b, the only way to compare them in one step is by applying the Υ-comparisons $\Upsilon(0,\beta,a-b)$ and $\Upsilon(0,\beta,b-a)$. These are only meaningful if a lower bound is given on the value $|a-b|$. Thus, the compare operation is applicable in particular for every finite set type.

- List of T is encoded via a number between 0 and 1. Assume T is general scalar type with cardinality n. The list $l = a_1 a_2 \cdots a_k$ is represented by

$$l = a_1 a_2 \cdots a_k \hookrightarrow \sum_{i=1}^{k} \frac{2a_i + 1}{(2n)^i} \quad . \tag{2.4}$$

(A special case for Boolean $n = 2$: $(\sum_{i=1}^{k} \frac{2a_i+1}{4^i})$.)

Using (2.2), we can write this also as

$$\sum_{i=1}^{k} \frac{\text{scalar } (a_i, n)}{(2n)^{i-1}} \ .$$

This is a number in the range $[0,1)$. If $\mathbf{Ord}(\mathbf{Car}(l)) = i$ $(i = 0,\ldots,(n-1))$, the value of the encoding ranges in

$$[\frac{2i+1}{2n} \ , \ \frac{2i+2}{2n}] \ .$$

The second value in the list further restricts the available range. If $\mathbf{Ord}(\mathbf{Car}(l)) = i$ and $\mathbf{Ord}(\mathbf{Cadr}(l)) = j$, the value of the encoding ranges in

$$[\frac{2i+1}{2n} + \frac{2j+1}{4n^2} \ , \ \frac{2i+1}{2n} + \frac{2j+2}{4n^2}] \ .$$

In summary, not every value in $[0,1]$ appears. The set of possible values is not continuous and has "holes". Such a set of values "with holes" is a Cantor set. Its self-similar structure means that bit (base $2n$) shifts preserve the "holes."

The advantage of this approach is that there is never a need to distinguish among two very close numbers in order to read the car of the list.

If a list l is represented via the number x, then:

Operation	Network's emulation
Car(l)	$\sigma(\frac{1}{2n} + \frac{1}{n}(\sigma(2nx-2) + \sigma(2nx-4) + \cdots + \sigma(2nx-(2n-2))))$
Cdr(l)	$\sigma((2nx - (\sigma(2nx-2) + \sigma(2nx-4) + \cdots + \sigma(2nx-(2n-2)))))$
Cons(e,l)	$\sigma(\frac{x}{2n} + e)$
Isnull(x)	$\sigma(2nx)$

(2.5)

- Stacks are represented similarly to lists, and operations on stacks are implemented analogously to operations on lists.
- A set is represented as a sorted list with no repetitions. If a finite type has cardinality n (up to Fix), n is an upper bound on the length of the list. Thus, all set operations require O(Fix) list operations.
- Records are represented by several variables (neurons), which are thought of as one unit.

The representations of the types char, integer, counter, and real appear in Appendix 2.4.

2.3.4 Compilation of Statements

Atomic Statements:

- An assignment expression is compiled in a straightforward way. Each such statement s_k has associated substatement counters $c_k^{(1)}$ and $c_k^{(2)}$, and in case it appears in a parallel block, it also has an associated flag counter c_k^{par}. A substitution $x = y$ is thus executed as follows:

$$\begin{aligned} x_1^+ &= \sigma(x - c_k^{(1)}) \\ x_2+ &= \sigma(y + c_k^{(1)} - 1) \\ x^+ &= \sigma(x_1 + x_2) \\ c_k^{(2)+} &= \sigma(c_k^{(1)}) \,. \end{aligned}$$

In case s_k appears in a sequential block, the next statement counter is updated by

$$c_{k+1}^{(1)+} = \sigma(c_k^{(2)})$$

and if it appears inside a parallel block, its flag neuron is updated by

$$c_k^{\text{par}\ +} = \sigma(c_k^{(l_k)} + c_k^{\text{par}} - c_m^{(1)})$$

and the next statement counter is updated by

$$c_m^{(1)+} = \sigma(c_k^{\text{par}} + \sum_{j=1}^{p} c_j^{\text{par}} - p)\ .$$

- Assuming that the numerical values of inputs are in the interval $[0, 1]$, already encoded as discussed earlier, the reading of an input is implemented as $x = \sigma($ Input.data + Input.validation$+c_k - 2)$. The Input.validation signal takes the role of the negation of the Eoc function. Output operations consist simply of copying the value of an output neuron.

- A Goto statement is implemented simply by a change in the neurons simulating the statement counters of the program.

- Procedure calls are implemented by storing the return address (neuron) in a predefined address and changing the statement counter. The calling program of the subroutine copies the actual parameters into the formal ones (even in the case of var parameters). For each var parameter, a copy statement from the actual to the formal parameter is evoked when updating it. (This will be further described below.)

Compound Statements:

- A *Parallel block* is translated into parallel execution of the individual statements. In the following NEL program, we associate statement counters with the associated statements, writing them to the left of the statements. Consider the block:

c_k : **Parbegin**

$c_{1,k}$: statement$_1$

$\vdots$

$c_{n,k}$: statement$_n$

Parend

c_{k+1}:

When c_k is set to 1, all substatement counters are set simultaneously to 1. The parallel block may end only when statement$_1$ to statement$_n$ finish their executions. Generally, the associated statement counters are updated as follows:

Parallel $\forall i$:

$$\begin{aligned} c^+_{i,k} &= \sigma(c_k) \\ & \quad \text{statement}_i \\ c^+_{i,k} &= 0 \\ c^+_k &= \sigma(c_{k-1} + c_{1,k} + \cdots + c_{n,k}) \\ c^+_{k+1} &= \sigma(c_k - c_{1,k} - \cdots - c_{n,k}) \ , \end{aligned}$$

The statement "Goto" is not allowed to jump from one parallel block to another. Furthermore, since in parallel execution there is no communication between statements, a variable cannot appear in more than one of the assignment statements that are executed in parallel.

The execution of serial blocks appear in Appendix 2.4.

Flow Control Statements:

- *If Statement:* The statement

  If (B) **then** stat1

  else stat2

  is implemented as

  Parbegin

  If (B) **then** stat$_1$;

  If $(\neg(B))$ **then** stat$_2$

  Parend

The statement counter of stat_1 includes the element "Or B", and that of stat_2 includes "Or $\neg(B)$"

- *Case and Cond statements:* **Case** - parallel execution of the associated **If** statements, and **Cond** is a serial execution of them.

Execution of other flow control statements as well as subroutine calls can be found in Appendix 2.4.

Appendix: Syntax of NEL

Here, we provide the syntax of expressions, statements, and subprograms.

2.3.5 Expressions

1. Atomic Expressions:
   - A constant is an expression.
   - A variable is an expression.
   - A function call is an expression.
2. Order Expressions:
   - The function **Ord** of any element of a finite-type returns an integer value.
   - **Pred** and **Succ** of an element e of a finite ordered type T returns another element of the same type.
   - **Chr** operates on an integer argument and returns a character.
3. Arithmetic Expressions:
   - Expressions consisting of finite affine combinations ($\sum_i c_i x_i + d$) are defined on either integers or reals. Here the x_i are variables and the c_i and d are constants. If all types are integers, then the expression is of integer type (include truncating when needed); otherwise, it is a real expression.
   - **Inc** and **Dec** are defined on counters and return counters.
4. Comparison Expressions: All these expressions return Boolean values.
   - Relational operators : $<, >, =, \geq, \leq, \neq$ between expressions of ordered type.
   - Relational operations on sets: **In**(element, set), **Equals**(set1,set2), **Includes**(set1,set2).
   - Predicates: **Iszero**(counter), **Isnull**(list), **Isempty**(stack).
   - Comparisons over the reals are described above, $\Upsilon(\alpha, \beta, x)$ may either produce Boolean values or is undefined.

5. Logical expressions: $\bigvee, \bigwedge, \neg$ are defined on Boolean values, and return Boolean values.

6. Compound-type expressions: Given a list of T, a stack of T, or a set of T:

   - **Car** (l) and **Top**(s) return elements of type T.
   - **Cdr** (l) and **Pop**(s) inherit the type of their arguments.
   - **Cons**(e, l) and **Push**(e, s) inherit the type of their second arguments.
   - **Union**(s_1, s_2), **Intersection**(s_1, s_2), **Set-difference**(s_1, s_2) inherit the type of their arguments. (The arguments s_1, s_2 have the same type.)

Here, we provide a detailed syntax of expressions:

{expr}	::=	{ordered expr} | {counter expr}
		{real expr} | {compound expr}
{ordered expr}	::=	{finite expr} | {integer expr}
{finite expr}	::=	{ Boolean expr} | {scalar expr} | {char expr}
{compound expr}	::=	{list expr} | {stack expr} | {set expr}
{Boolean expr}	::=	{ordered expr} {relational operator} {ordered expr} |
		{Boolean term} | {Boolean term} $\bigvee$ {Boolean term}
{Boolean term }	::=	{Boolean factor} | {Boolean factor} $\bigwedge$ {Boolean factor}
{Boolean factor}	::=	{Boolean var} | {Boolean value} |
		{Boolean function call} | {Boolean expr} |
		Succ({Boolean expr}) | **Pred**({Boolean expr})
		Υ({real constant},{real constant},{real expression}) |
		$\neg$({Boolean expr}) | {predicate}
{predicate}	::=	**Iszero** ({counter expr}) |
		Isnull ({list expr}) |
		Isempty ({stack expr}) |
		In({finite expr}, {set expr}) |
		Equals({set expr}, {set expr}) |
		Includes({set expr}, {set expr}) |

		Eoc {channel identifier}
{relational operator}	::=	$<, >, =, \geq, \leq, \neq$
{function call}	::=	{identifier} | {identifier} ([{expression},])
{scalar expr}	::=	{scalar var} | {scalar value} |
		{**Succ**({scalar expr}) | **Pred**({scalar expr})
		Car({list expr}) | **Top** ({stack expr})
{char expr}	::=	{char var} | {char value} |
		Chr({integer expr}) | {**Succ**({char expr}) | **Pred**({char expr})
{integer expr}	::=	{integer term} + {integer term}
{integer term}	::=	{integer var} | {integer value} |
		{constant} * {integer var} | **Ord**{finite expr}
{counter expr}	::=	{counter var} | {counter value} |
		Dec({counter expr}) | **Inc**({counter expr})
{real expr}	::=	{real term} + {real term}
{real term}	::=	{real var} | {real value} | {constant} * {real var}
{list expr}	::=	{list var} | {list value} |
		Cons({finite expr}, {list expr})| **Cdr**({list expr})
{stack expr}	::=	{stack var} | {stack value} |
		Push({finite expr}, {stack expr})| **Pop** ({stack expr})
{set expr}	::=	{set var} | {set value} |
		Union({set expr}, {set expr}) |
		Intersection({set expr}, {set expr}) |
		Set-difference({set expr}, {set expr})
		{* value - input value or constant (if numerical) *}

An alternative, when type matching is not imposed, is given below:

{expr}	::=	{term} | {expr} {addop} {term}
{addop}	::=	+ | − | **Or**
{term}	::=	{factor} | {term} {multop} {factor}
{multop}	::=	∗ | / | **And**
{factor}	::=	{ constant} | {variable} |
		{function call} | {predef function call} |
		{expr} | **Not**{expr}|
		{expr} {relop} {expr} |
		ϒ({constant},{constant},{expression})
{variable}	::=	{identifier}
{function call}	::=	{identifier} | {identifier} [{expr},]
{relop}	::=	< | | > | = | ≥ | ≤ | ≠
{predef function call}	::=	**Succ**({expr}) | **Pred**({expr}) |
		Car({expr}) | **Cdr**({expr}) | **Cons**({expr}, {expr})|
		Union({expr}, {expr}) | **Intersection**({expr}, {expr}) |
		Set-difference({expr}, {expr}) |
		Top ({expr}) | **Pop** ({expr}) | **Push** ({expr, {expr}) |
		Chr({expr}) | **Ord**({expr})|
		Dec({expr}) | **Inc**({expr}) |
		Iszero ({expr}) |
		Isnull ({expr}) |
		Isempty ({expr}) |
		In({expr}, {expr}) |
		Equals({expr}, {expr}) |
		Includes({expr}, {expr}) |
		Eoc {identifier}

2.3.6 Statements

{statement} ::= {label}: {unlabeled statement} | {unlabeled statement}

{unlabeled statement} ::= {atomic st} |
{compound st} |
{conditional st} |
{repetition st} |
{record st}

1. Atomic statements:
   (a) Assignment: {variable} = {expression}. The types should match.
   (b) Procedure call: {identifier} | {identifier} ([{expression},])
   (c) I/O statement:
      i. **Read**(channel identifier, {variable}) reads one input datum from the sequence of data elements appearing on the input channel specified in this command. The input is interpreted to be in the type of the variable it is read into. It is also possible to read without storing the datum: **Read**(channel identifier).
      ii. **Write**(channel identifier, {variable}) copies the value of the variable to the output.
      iii. Go-to statement: **Goto** {label}, where {label} is a counter expression. That is, labels are natural numbers or 0, and the number of labels is unbounded.
2. Compound statements (blocks):
   (a) The sequence of statements appearing between **Parbegin** and **Parend** is executed in parallel. *These statements can only be applied in the top level of the main program.*
   (b) The sequence of statements between **Begin** and **End** is executed serially.
3. Control Structure Statements. Conditional statements:
   (a) **If** {Boolean expression} **then** {statement}

(b) **If** {Boolean expression} **then** {statement} **else** {statement}.

(c) A case statement is defined as:

Case {ordered expr} **of** [{case label list}: statement] **end**

{case label list} ::= [{ordered value},] (e.g. integer 3, scalar 'Monday')

Informally, the case statement tests the value of an identifier i, and compares it to the different values v. For example, consider this program fragment:

Case i **of**

$((v_1 \quad s_1) (v_2 \quad s_2) \cdots (v_k \quad s_k))$

The value of i must be equal to exactly one of the v's in the list; the associated sentence is executed.

(d) **Cond** ([{Boolean expression} { statement},])

For example:

Cond

$((b_1 \quad s_1) (b_2 \quad s_2) \cdots (b_k \quad s_k))$

The Cond statement is a "compound if" statement. The first statement associated with a True Boolean expression is executed. (All Boolean conditions may be false. Then nothing is executed.)

4. Control Structure Statements. Repetition statements:

(a) **While** {Boolean expression} **do** {statement}

(b) **Repeat** {statement} **until** {Boolean expression}

(c) For loop has the following syntax:

{for loop} ::=	**For** {counter identifier} = {for-list} **do** {statement},
{for-list} ::=	{value} **to** {value} |
	{value} **downto** {value}
{value} ::=	{counter expression}

(d) **Dolist**({list expr} {statement}).

This loop repeats length(list) times. At each repetition i, the statement —that has one free variable— is applied to the ith element of the list.

5. **With** {record variable list} **do** {statement}

2.3.7 Subprograms

Two types of subprograms are defined to facilitate top-down design using step-wise refinement [Che80]: function and procedure. Functions return one value of predefined type, while procedures return no value. We have previously seen the syntax of declaring functions and procedures, and the syntax used when invoking them. Here, we refine our discussion of subprograms. A function f is declared by a statement like

$$\textbf{Function } f\ (p_1 : t_1,\ p_2 : t_2,\ \ldots, p_n : t_n) : T$$

and a procedure is declared by

$$\textbf{Procedure } p\ (p_1 : t_1,\ p_2 : t_2,\ \ldots, p_n : t_n)$$

The formal parameters are listed along with their types. There are four types of formal parameters: var, value, function, and procedure. Subprograms are invoked by statements such as $k(e_1, e_2, \ldots e_n)$ with a list of n expressions of the type t_i, respectively. For value parameters, the expressions are evaluated and then assigned into the formal parameters. In var parameters, the actual parameters are names of variables, and the subprogram considers these formal parameters to be equivalent to the actual parameters in which it was invoked. The actual parameters of function or procedure variables are function or procedure identifiers, respectively. Note that the NEL language is strongly typed, and the actual parameters should match the type of the formal ones.

Static declaration of subprograms may be viewed as a tree: The root is the program; a subprogram p_1 is a child of p_2 if it is defined in it. A program may invoke another one which is either its child, sibling, or sibling of its ancestor. Cyclic invoking is not allowed nor is recursion. The conventions regarding the scope of parameters is as in Pascal (e.g. [Che80] Chapter 7.)

2.4 Appendix: Compilation of NEL

2.4.1 Compilation of Data Types

- Chars are treated similarly to Scalars. We use the encoding $\mathbf{Chr}(i) = \frac{2i+1}{2n}$, where n is the cardinality of the character type.

- Integers are represented as binary numbers with up to q bits. Arithmetic operations (i.e., addition of two variables and multiplication of a variable with a constant) are executed by the update:

$$x_j^+ = \sigma(\sum_i a_i x_i + c_j)$$

where x_i are variables and a, b are either integer or rational constants. The precision q is program dependent, and is chosen such that the network operations involving σ are sensitive enough to be able to discern via the relational operations between two such integers (also to truncate) in a constant time duration.

- A counter with the value n is represented as $(1 - 2^{-n})$, that is

$$\text{counter}(n) \hookrightarrow .\underbrace{11\ldots1}_{n} \tag{2.6}$$

The operations on counters are implemented as follows:

Operation	Network's emulation
$\mathbf{Inc}(x)$	$\sigma(\frac{1}{2}(x+1))$
$\mathbf{Dec}(x)$	$\sigma(2x-1)$
$\mathbf{Iszero}(x)$	$\sigma(1-2x)$.

(2.7)

- Real values are continuous in the range $[0, 1]$. Arithmetic operations are as in the integer case:

$$x_j^+ = \sigma(\sum_i a_i x_i + c_j)$$

but with the possibility of incorporating real values. The relational operator $\Upsilon(\alpha, \beta, x)$ is emulated by

$$\sigma(\frac{x-\alpha}{\beta-\alpha}) \,. \tag{2.8}$$

2.4.2 Compilation of Statements

- *Serial blocks* are implemented in terms of parallel blocks:

  Begin

  statement$_1$

  $\vdots$

  statement$_n$

  End

  where we have to use pointer variables for the different statements: $c_1, \ldots c_n$. We assume a control line that has the value 1 once. The operations are then

  c_1 = control

  Parbegin

  If (stat$_1$ is done) **then** $c_2 = c_1$, $c_1 = 0$;

  If (stat$_2$ is done) **then** $c_3 = c_2$, $c_2 = 0$;

  $\vdots$

  If (c_1) **then** statement1 ;

  If (c_2) **then** statement2 ;

  $\vdots$

  Parend

- *A while loop*

  While (B) **do**

  statement

is compiled into the parallel execution:

l : **Begin**

If (B) **then** statement

else Goto$(l+1)$

Goto(l)

End

$l+1$:

- *A Repeat loop*

Repeat

statement

Until (B)

is compiled into

l : **Begin**

statement

If (B) **then Goto**$(l+1)$

Goto(l)

End

$l+1$:

- *For Loop*: This loop utilizes a counter to decide the halting condition

For $i = 1 \ldots n$ **do**

statement

The statement compiles into the following execution:

$i = 1$;

While $(i \leq n)$ **do**

Begin

statement;

$i = i + 1$

End

This is in turn compiled into a network as before.

- *Dolist* can be written as a regular **For** loop using the operations **Car** and **Cdr**.
- *With* record operations: records are simulated by a set of neurons, one per each field. The simulation is, then, simple.

2.4.3 Compilation of Subroutine Calls

We assume a control line named Fcall that activates the subprogram call. The subprogram is complied into the following commands:

Begin

If (Fcall) **then** $x_i = v_i \quad (i = 1, 2, \ldots)$ {* for value parameters, copying the value *}

If (Ending) **then** $x_i = 0$

$\vdots$

Function's Block

Ending = condition

End

Chapter 3

The Computational Power of Recurrent Networks: Overview

Many authors have reported successful applications when using neural networks for various computational tasks, including classification and optimization problems. Special purpose analog chips are being built to implement these solutions directly in hardware; see for instance [AA87], [EDK+89]. However, very little work has been done in the direction of exploring the ultimate capabilities of such devices from a theoretical standpoint. Part of the problem is that, much interesting work notwithstanding, analog computation is hard to model, as difficult questions about precision of data and readout of results are immediately encountered —see for instance [VSD86], and the many references there.

We take the point of view that artificial neural nets provide an opportunity to reexamine some of the foundations of analog computation from the new perspective afforded by an extremely simple yet surprisingly rich model, in a context where techniques from dynamical systems theory interact naturally with more standard notions from theoretical computer science. Starting from such a model, we derive results on deterministic versus nondeterministic computation, and we relate our study to standard concepts in complexity theory.

With the constraint of an unchanging structure, it is easy to see that classical McCulloch-Pitts—that is, binary—neurons would have no more power than finite automata, which is not an interesting situation from a theoretical complexity point of view. Therefore, and also because this is what is done in practical applications of neural nets, and because it provides a closer analogy to biological systems, we take our neurons to have a graded, analog, response.

3.1 Computational Power

Assume a machine M receives as input finite binary strings, that is, words in $\{0,1\}^*$ and outputs a single character response after a certain amount of processing. Assume without loss of generality that the response is binary. All those strings to which M responds with 1 define the language accepted by the machine.

A well-developed field in computer science deals with the characterization of languages into classes defined in terms of the complexity of the machines needed to recognize them (see for example [HU79]). These classes include for example the regular languages (those accepted by finite automata), recursive languages (those accepted by Turing machines), and many others. That is, a specific type of machine architecture is associated with the class of languages that it is capable of recognizing.

To understand the computational power of recurrent networks, we study in this work how they perform as recognizers. The findings are pretty surprising. Not only did we find a precise correspondence between the recurrent network model and different classical models, but we found that the only parameter that determines the class of languages recognized by our model is the type of numbers utilized as weights by the network. Thus, the main known computational classes can be re-described as the classes of languages recognized by recurrent networks with some specific types of weights. The hierarchy of classes of languages is then in a one-to-one correspondence with real numbers, according to their descriptional classification.

The proof in the rational and real cases make use of the language NEL introduced in Chapter 2. The integer case could be done using a restricted subset of NEL, but we will give a direct proof, as this case is very easy to deal with.

3.2 Basic Definitions

As we discussed in Chapter 1, we consider synchronous networks which can be represented as dynamical systems whose state at each instant is a real vector $x(t) \in \mathbb{R}^N$. The ith coordinate of this vector represents the activation value of the ith processor at time t. In matrix form, the equations are as in (1.4), for suitable matrices A, B and vector c.

Given a system of equations such as (1.4), an initial state $x(1)$, and an infinite input sequence

$$u = u(1), u(2), \ldots ,$$

we can define iteratively the state $x(t)$ at time t, for each integer $t \geq 1$, as the value obtained by recursively solving the equations. This gives rise, in turn, to a sequence of output values, by restricting attention to the output processors; we refer to this sequence as the "output produced by the input u" starting from the given initial state.

To define what we mean by a net recognizing a language

$$L \subseteq \{0,1\}^+ ,$$

we must first define a *formal network*, a network which adheres to a rigid encoding of its input and output. We define formal nets with two binary input lines. The first of these is a *data line*, and it is used to carry a binary input signal; when no signal is present, it defaults to zero. The second is the *validation line*, and it indicates when the data line is active; it takes the value "1" while the input is present there and "0" thereafter. We use "D" and "V" to denote the contents of these two lines, respectively, so

$$u(t) = (D(t), V(t)) \in \{0,1\}^2$$

for each t. We always take the initial state $x(1)$ to be zero and to be an equilibrium state, that is,

$$\sigma(A0 + B0 + c) = 0 .$$

We assume that there are two output processors, which also take the role of data and validation lines and are denoted $O_d(t), O_v(t)$ respectively.

(The convention of using two input lines allows us to have all external signals be binary; of course many other conventions are possible and would give rise to the same results, for instance, one could use a three-valued input, say with values $\{-1, 0, 1\}$, where "0" indicates that no signal is present, and ± 1 are the two possible binary input values.)

We now encode each word

$$\alpha = \alpha_1 \cdots \alpha_k \in \{0,1\}^+$$

as follows. Let

$$u_\alpha(t) = (V_\alpha(t), D_\alpha(t)) , \quad t = 1, \ldots ,$$

where

$$V_\alpha(t) = \begin{cases} 1 & \text{if } t = 1, \ldots, k \\ 0 & \text{otherwise ,} \end{cases}$$

and

$$D_\alpha(t) = \begin{cases} \alpha_k & \text{if } t = 1, \ldots, k \\ 0 & \text{otherwise .} \end{cases}$$

Given a formal net $\mathcal{N}$, with two inputs as above, we say that a word α is *classified in time* τ, if the following property holds: the output sequence

$$y(t) = (O_d(t), O_v(t))$$

produced by u_α when starting from $x(1) = 0$ has the form

$$O_d = \underbrace{0 \cdots 0}_{\tau-1} \eta_\alpha 000 \cdots , \quad O_v = \underbrace{0 \cdots 0}_{\tau-1} 1000 \cdots ,$$

where $\eta_\alpha = 0$ or 1.

Let $T : \mathbb{N} \to \mathbb{N}$ be a function on natural numbers. We say that the language $L \subseteq \{0,1\}^+$ is *recognized in time* T by the formal net $\mathcal{N}$ provided that each word $\alpha \in \{0,1\}^+$ is classified in time $\tau \leq T(|\alpha|)$, and η_α equals 1 when $\alpha \in L$ and is $= 0$ otherwise.

3.3 Main Results

We investigate the computational power of recurrent networks. We prove that the power of a network depends on the type of numbers utilized as its weights. The natural mathematical choices of numbers are: integers, rationals, and reals. The natural choices of classes of formal languages are: regular, recursive, and "all languages." We establish the correspondence between these three choices, respectively. Furthermore, when the computation time of the network is constrained to be polynomial in the input size, the classes recognized by the networks are regular (Chapter 4), P (Chapter 5), and non-uniform P, i.e. P/poly (Chapter 6). The results are summarized in the following table.

Weights	Capability	Polytime
integer	regular	regular
rational	recursive	(usual) P
real	arbitrary	analog P

We discuss (in Chapter 7) a hierarchy of numbers between rationals and reals in terms of information theory. We prove that networks that utilize as weights numbers carrying different amount of information result in different computational power. Thus, we establish an infinite hierarchy of networks classified by their allowed weights, between the rationals and the general real case.

Chapter 4

Networks with Integer Weights

We establish a correspondence between networks in which weights are restricted to take integer values and the simplest class of machines, namely, finite automata. The classical 1943 result of McCulloch and Pitts ([MP43]) (see also Kleene's work [Kle56]) shows how to implement logic gates by threshold networks, and therefore how to simulate finite automata by such nets. For us, however, neurons allow for analog values ($[0,1]$) rather than the discrete 0-1 McCulloch and Pitts neurons. Thus, potentially our networks are more powerful. We show that when we restrict our networks to allow for integer weights only, the neurons may assume only binary activations, and the networks become computationally equivalent to those studied by McCulloch and Pitts. The material of the chapter is fairly straightforward, and essentially well-known, but it is needed for completeness.

4.1 Finite Automata - Preliminaries

Recall that a *finite automaton* (FA) is a machine that consists of finite set of states. It moves from state to state after each input symbol is received. Formally, a FA is defined as a 5-tuple ([HU79]) $M = (Q, \Sigma, \delta, q_0, F)$ where Q is a finite set of *states*, Σ is a finite set of *input* symbols, $q_0 \in Q$ is the initial state, $F \subseteq Q$ is the set of *accepting* states, and $\delta : Q \times \Sigma \mapsto Q$ is the *transition* function. That is, $\delta(q,a)$ is a state, for any state q and input symbol a, interpreted as the "next-state". We assume that $\delta(q,a) = q$ for all $q \in F$.

We extend the transition function δ to be defined over a state and a string (rather than a single symbol) by means of the formula (inductively on the length of ω) $\delta(q, \omega a) = \delta(\delta(q,\omega), a)$, for a symbol $a \in \Sigma$ and a string of 0 or more symbols $\omega \in \Sigma^*$. We also define for completeness $\delta(q, \epsilon) = q$ for the empty string ϵ. A string x is said to be *accepted* by a finite automaton M if $\delta(q_0, x) \in F$. The *language* $L(M) \subseteq \Sigma^*$ *accepted* by M is the set of all accepted strings. A language is *regular*

if it is accepted by some finite automaton. Note that the decision of acceptance or rejection of a string is made immediately after it is read.

In our model of computation, i.e. neural networks, decisions are indicated by a special output signal. To establish the desired equivalence we need to define an *offline finite automaton*. This is an automaton that may continue computing after reading in the input string, thus providing the decision not immediately after reading the input but after some delay. A decision is reached when the computation arrives to either an accepting or a rejecting state.

Formally, we denote an offline finite automaton as a 5-tuple $(Q, \Sigma, \delta_f, q_0, F_f)$. The transition function δ_f maps $Q \times (\Sigma \bigcup \$)$ into Q, where '\$' is a special symbol ($\$ \notin \Sigma$) denoting that the complete input string has been read into the machine. Here, $F_f \subseteq Q$ is the set of accepting states, and the transition function satisfies that $\delta_f(q, a) = q$ for all $a \in (\Sigma \bigcup \$)$ and $q \in F_f$. The objects Q, Σ, q_0 are defined as in the finite automaton case. We extend the transition function δ_f as above to be defined on a state and a string of the type $x\$^*$, where $x \in \Sigma^*$ and $\$^*$ is a string of 0 or more appearances of the '\$' sign. We define a function $\tilde{\delta}_f$ which maps $Q \times \Sigma^*$ (without the \$ sign) into 2^Q (the set of subsets of Q) by

$$\tilde{\delta}_f(q, x) = \{\delta_f(q, x\$^*)\}$$

for any $x \in \Sigma^*$. A string $x \in \Sigma^*$ is said to be *accepted* by an offline finite automaton M if $\tilde{\delta}_f(q_0, x) \bigcap F_f \neq \phi$.

Lemma 4.1.1 The class L_f of languages accepted by offline finite automata is exactly the class of regular ones.

Proof. Any finite automaton can be seen as an offline automaton by letting $\delta_f(q, \$) = q$ for all $q \in Q$. Thus, the regular languages are included in L_f. To prove the other inclusion, we show next that given an offline finite automaton $M = (Q, \Sigma, \delta_f, q_0, F_f)$ accepting a language l, there is a finite automaton M' that accepts the same language.

Define $F = \{q \in Q \mid \tilde{\delta}_f(q, \$^*) \bigcap F_f \neq \phi\}$, and let δ be the map δ_f restricted to $Q \times \Sigma$. Then, the machine $M = (Q, \Sigma, \delta, q_0, F_f)$ accepts l. ∎

4.2 Integer Networks and Regular Languages

Theorem 2 *The languages accepted by integer networks are exactly the regular ones.*

Proof. We establish a correspondence between integer networks and offline finite automata.

$\Rightarrow$

Given a formal integer network $\mathcal{N}_i$ that consists of N neurons and accepts the language l, we define an offline automaton $M = (Q, \{0,1\}, \delta_f, q_0, F_f)$ as follows:

1. We identify the input $(V(t), D(t))$ to $\mathcal{N}_i$ with the values $\{0, 1, \$\}$ using the following encoding: $\zeta[(1,0)] = 0$, $\zeta[(1,1)] = 1$ and $\zeta[(0,0)] = \$$ (the case $(0,1)$ is invalid).

2. Let $Q = \{0,1\}^N$ and $q_0 = 0^N$. (Note that as the network starts from an initial state 0^N and utilizes only integer weights, its neurons may assume binary values only.) That is, we identify each state of M with the activations of all the neurons.

3. Assume w.l.o.g. that $O_v(t), O_d(t)$ are the first and second neurons in the state encoding, respectively. We denote by $q[i]$ the ith coordinate of state q. Then, $F_f = \{q \in Q | \ \ q[1] = 1, q[2] = 1\}$.

4. The transition function δ_f is defined by the update equation $q^+ = \sigma(Aq + Bu + c)$.

It is easy to verify that $L(M) = l$.

$\Leftarrow$

Given an offline finite automaton $M = (Q, \Sigma, \delta_f, q_0, F_f)$ that accepts a language $l \in \{0,1\}^*$, assume w.l.o.g. that there is no transition into the initial state q_0. We define a formal integer network $\mathcal{N}$ and the simulation of M as follows:

The input letters from $\{0, 1, \$\}$ are translated via the function ζ^{-1}. The number of neurons is $N = 3|Q| + 2$. Of these, $3|Q|$ are indexed by $j = 0, \ldots, (|Q| - 1)$, and a pair (k, l) which may assume the values $(0,0)$, $(1,0)$, or $(1,1)$. Each x_{jkl} may assume a binary value only, where x_{jkl} is 1 if and only if the current state of the machine M is q_j and its last input was $\zeta[(k,l)]$. The construction will be so that at each moment exactly one of these neurons has the value 1 and all the rest have the value 0.

Before we show the update equations of the neurons x_{jkl}, we introduce some notation: Introduce $|Q|$ binary variables p_j $(j = 0, \ldots, (|Q| - 1))$, each of them having the value 1 if the state of the

machine M is q_j. These variables can be updated by the previous activation values of the neurons x_{jkl} as follows:

$$\begin{aligned} p_0^+ &= 1 - \sum x_{jkl} \\ p_j^+ &= \sum a_{jkl}\, x_{jkl} \end{aligned}$$

where the sum is over all $j = 0, \ldots, (|Q| - 1)$, and pairs (k, l) as before, and the constants a_{jkl} are such that $a_{jkl} = 1$ when $\delta_f(q_j, \zeta[(k, l)]) = q_i$ and $a_{jkl} = 0$ otherwise.

Now we are ready to precisely state the update equation of the neurons x_{jkl}:

$$\begin{aligned} x_{j11}^+ &= \sigma(p_j + V + D - 2) \\ x_{j10}^+ &= \sigma(p_j + 2V - D - 2) \\ x_{j00}^+ &= \sigma(p_j - V - D)\ . \end{aligned}$$

Two additional neurons are the validation and data output: Define $F_R = \{q \in Q \mid \tilde{\delta}_f(q, \$^*) \bigcap F_f = \phi\}$. The validation neuron updates by $x_v^+ = \sigma(\sum_{j=0}^{|Q|-1} a_j\, x_{j00})$, where a_j is 1 when $q_j \in (F_f \bigcup F_R)$ and is 0 otherwise. The data neuron updates by a similar equation where $a_j = 1$ when $q_j \in F_f$ and $a_j = 0$ otherwise.

It is easy to verify that M and $\mathcal{N}$ accept the same language. Note that all weights of $\mathcal{N}$ are integers. ∎

Chapter 5

Networks with Rational Weights

As discussed in Chapter 3, our neural network model when restricted to integer weights, is not more powerful than a finite automaton. Here, we discuss the same model, but let the weights be rational numbers. The rational numbers we consider are simple, small and do not require much precision. For example, one could verify that the result we show in section 5.1 does not require more complicated rational weights than those that can be written as $\frac{n_1}{n_2}$, for $n_1, n_2 \in \mathbb{N}$, $n_2 \in \{1, \ldots, 4\}$ and $n_1 \in \{-4, \ldots 8\}$.

Related work that asserted universality of a finite network—similar to the recurrent network model—was done by Pollack [Pol87]. His model consisted of a finite number of neurons with two different activation functions, identity and threshold. The machine was *high-order* (also called Sigma-Pi), that is, the inputs of each neuron were combined using multiplications as opposed to just linear combinations. Pollack conjectured that high-order connections are necessary in order to achieve universality. The validity of this conjecture would imply that our recurrent network model is less powerful than standard models of computation. This conjecture was for a time rather widely accepted by the neural network community. In particular, it was the basic motivation for using high-order models (i.e. ([CSSM89], [Elm90], [GMC+92], [Pol90], [WZ89]). In this chapter we refute this conjecture.

We prove in this chapter that one can simulate all (multi-tape) Turing Machines by nets, using *only* first-order (i.e., linear) connections and rational weights. Furthermore, this simulation can be done in linear time. In particular, it is possible to give a net made up of about 1,000 processors which computes a universal partial-recursive function. Non-deterministic Turing Machines can be simulated by non-deterministic rational nets, also in linear time. Later in this chapter, we prove that the simulation of a Turing machine by a neural network can in fact be done in *real time* rather than linear time.

The simulation result has many consequences regarding the *decidability*, or more generally the complexity, of questions about recursive nets of the type we consider. For instance, determining if a given neuron ever assumes the value "0" is effectively undecidable (as the halting problem can be reduced to it); on the other hand, the problem is believed to be decidable if a linear activation is used (halting in that case is equivalent to a fact that is widely conjectured to follow from classical results due to Skolem and others on rational functions; see [BR88], page 75), and is also decidable in the pure threshold case (there are only finitely many states). As our function σ is in a sense a combination of thresholds and linear functions, this gap in decidability is perhaps remarkable. Another consequence of our results is that the problem of determining if a dynamical system

$$x^+ = \sigma(Ax + c)$$

(with A and c rational) ever reaches an equilibrium point, from a given (rational) initial state, is effectively undecidable. Given the linear-time simulation bound, it is of course also possible to transfer NP-completeness results into the same questions for nets having rational coefficients.

The remainder of this chapter is organized as follows. In section 5.1, we prove the simulation of a TM by a neural network with a linear slow down in the computation. Starting with section 5.2 and through the end of this chapter, we construct a network that simulates a given TM in real time. This section includes the construction of a universal network with 886 neurons. The proof of the real-time simulation is more complicated than the proof of the linear-time result, but it is independent of the use of the language NEL and is self-contained.

5.1 Rational Networks Simulate Turing Machines in Linear Time

It is an easy exercise to verify that a network with rational weights can be simulated by a Turing machine with a polynomial time slow down. Here, we show the other side of the simulation; we prove that networks can simulate Turing machines. Furthermore, we construct a simulation that does not require more than linear time slow-down in the computation.

Theorem 3 *Let $\phi : \{0,1\}^* \to \{0,1\}^*$ be any recursively computable partial function. Then, there exists a formal network $\mathcal{N}$ which computes ϕ. Furthermore, if a Turing Machine $\mathcal{M}$ (of one*

input tape and several working tapes) computes $\phi(\omega)$ *in time* $T(|\omega|)$*, then* $\mathcal{N}$ *computes* ϕ *in time* $O\left(T(|\omega|) + |\omega|\right)$. ∎

Proof. As a departure point, we pick single-tape Turing Machines with binary alphabets. As is well-known, by storing separately the parts of the tape to the left and to the right of the head, we may equivalently study push-down automata with two binary stacks. This machine consists of a finite control together with two binary stacks. At each step, the stacks are read via Top and Isempty operations (the same definition as in Chapter 2: Top reads the top element of the stack and Isempty verifies if the stack is empty), the finite control considers its state and the readings from the stacks and computes two functions: the next state and the operations on each stack. (The operations on the stacks are Push(0), Push(1), or Pop: Push adds the top element, while Pop removes it from the stack.) At the end of the step, the stack operations are executed, and the control changes its state according to the transition function.

To show the existence of a network that simulates a two stack machine in linear time, we exhibit a program with this property. Given the linear time equivalence of programs in NEL and networks (see Chapter 2), theorem 3 will be proved.

Consider a program with one binary input channel and one binary output channel, as follows:

Program Turing-machine (input,output);
Var s_1, s_2: Stack;
Begin
  While (¬ **Eoc** (input)) **do**
    s_1 = **Push** (**Read** (input), s_1);
  Repeat
    %Simulate one step on the two stack machine
    state = **Next-state** (state, **Top** (s_1), s_1, **Top** (s_2), s_2)
    s_1 = **Stack-Op**$_1$ (state, **Top** (s_1), **Isempty** (s_1), **Top** (s_2), **Isempty** (s_2), s_1)
    s_2 = **Stack-Op**$_2$ (state, **Top** (s_1), **Isempty** (s_1), **Top** (s_2), **Isempty** (s_2), s_2)
    %Next-state, Stack-Op$_1$ and Stack-Op$_2$ are procedures that require constant
    %computation time.
  Until (halting state)

Open (output)
While ($\neg$ **Isempty** (s_1)) **do**
Begin
Write (output, **Top** (s_1));
s_1 =**Pop** (s_1)
End;
End.

∎

5.2 Simulation of Turing Machines in Real Time

Theorem 3 states a Turing machine simulation by a neural network, while preserving the computation time, up to a linear slow-down. Here, we prove the following stronger result.

Theorem 4 *Let M be a Turing machine that computes ϕ in time T. Then, there is a network $\mathcal{N}$ that computes ϕ in time $T + O(|w|)$ for any input ω.*

The result is proved not just for standard Turing machines, but also for multi-tape versions. The proof is self-contained, without use of the language NEL.

It will be convenient to have a version of Theorem 4 that does not involve inputs but rather an encoding of the initial data into the initial state. (This would be analogous, for Turing machines, to an encoding of the input into an input tape rather than having it arrive as a stream of external input.)

For processor net without inputs, we may think of the dynamics map $\mathcal{F}$ as a map $\mathbb{Q}^N \to \mathbb{Q}^N$. In that case, we denote by $\mathcal{F}^k$ the k-th iterate of $\mathcal{F}$. For a state $\xi \in \mathbb{Q}^N$, we let $\xi^j := \mathcal{F}^j(\xi)$. We now state that if $\phi : \{0,1\}^* \to \{0,1\}^*$ is a recursively computable partial function, then there exists a processor net $\mathcal{N}$ without inputs, and an encoding of data into the initial state of $\mathcal{N}$, such that: $\phi(\omega)$ is undefined if and only if the second processor has activation value always equal to zero, and it is defined if this value ever becomes equal to one, in which case the first processor has an encoding of the result.

Given $\omega = a_1 \cdots a_k \in \{0,1\}^*$, we define the encoding function

$$\delta[a_1 \cdots a_k] := \sum_{i=1}^{k} \frac{2a_i + 1}{4^i}. \tag{5.1}$$

(Note that the empty sequence gets mapped into 0.)

Theorem 5 *Let M be a Turing machine computing $\phi : \{0,1\}^* \to \{0,1\}^*$ in time T. Then there exists a processor net $\mathcal{N}$ without inputs so that the following properties hold: For each $\omega \in \{0,1\}^*$, consider the initial state*

$$\xi(\omega) \ := \ (\delta[\omega], 0, \ldots, 0) \in \mathbb{Q}^N.$$

Then, if $\phi(\omega)$ is undefined, the second coordinate $\xi(\omega)_2^j$ of the state after j steps is identically equal to zero, for all j. If instead $\phi(\omega)$ is defined, then

$$\xi(\omega)_2^j \ = \ 0 \ , \ j = 0, \ldots, T-1, \xi(\omega)_2^T \ = \ 1 \ ,$$

and $\xi(\omega)_1^T = \delta[\phi(\omega)]$.

In the appendix of this chapter, we show how to add inputs and outputs to Theorem 5, thus obtaining Theorem 4 as a corollary of Theorem 5.

To prove Theorem 5, we construct the network in three phases. After a general discussion (in section 5.3), we show (in section 5.4) how to simulate a Turing machine with a "two-level" neural network. In section 5.5, we modify the construction so that in one of the levels, the neurons differ from the standard neurons: they compute linear combinations of their input with no sigma function applied to the combinations. Finally, in section 5.6, we show how to modify the last network into a standard one with one level only.

5.3 General Construction Of The Simulation

As a departure point, we pick p'-tape Turing Machines with binary alphabets. We may equivalently study push-down automata with $p = 2p'$ binary stacks. We choose to represent the values in the stacks as fractions with denominators which are powers of four. An algebraic formalization is as follows.

5.3.1 P-Stack Machines

Denote by $\mathcal{C}$ the "Cantor 4-set" consisting of all those rational numbers q which can be written in the form

$$q = \sum_{i=1}^{k} \frac{a_i}{4^i}$$

with $0 \leq k < \infty$ and each $a_i = 1$ or 3. (When $k = 0$, we interpret this sum as $q = 0$.) Elements of $\mathcal{C}$ are precisely those of the form $\delta[\omega]$, where δ is as in Equation 5.1.

The instantaneous description of a p-stack machine, with a control unit of n states, can be represented by a $(p+1)$-tuple

$$(s, \delta[\omega_1], \delta[\omega_2], \ldots, \delta[\omega_p]),$$

where s is the state of the control unit, and the stacks store the words ω_i $(i = 1, \ldots, p)$, respectively. (Later, in the simulation by a net, the state s will be represented in unary as a vector of the form $(0, 0, \ldots, 0, 1, 0, \ldots, 0)$.)

For any $q \in \mathcal{C}$, we write

$$\zeta[q] := \begin{cases} 0 & \text{if } q \leq \frac{1}{2} \\ 1 & \text{if } q > \frac{1}{2} \end{cases},$$

and:

$$\tau[q] := \begin{cases} 0 & \text{if } q = 0 \\ 1 & \text{if } q \neq 0 \end{cases}.$$

We think of $\zeta[\cdot]$ as the "top of stack," as in terms of the base-4 expansion, $\zeta[q] = 0$ when $a_1 = 1$ (or $q = 0$), and $\zeta[q] = 1$ when $a_1 = 3$. We interpret $\tau[\cdot]$ as the "nonempty stack" operator. Nonempty is the logical negation of the stack operation Isempty described in Chapter 2. We use it in this chapter to simplify the discussion. It can never happen that $\zeta[q] = 1$ while $\tau[q] = 0$; hence the pair $(\zeta[q], \tau[q])$ can have only three possible values in $\{0, 1\}^2$.

Definition 5.3.1 A *p-stack machine* $\mathcal{M}$ is specified by a $(p+4)$-tuple

$$(S, s_I, s_H, \theta_0, \theta_1, \theta_2, \ldots, \theta_p),$$

where S is a finite set, s_I and s_H are elements of S called the *initial* and *halting states*, respectively,

and the θ_i's are maps as follows:

$$\begin{aligned}\theta_0 &: S \times \{0,1\}^{2p} \to S \\ \theta_i &: S \times \{0,1\}^{2p} \to \{(1,0,0),(\frac{1}{4},0,\frac{1}{4}),(\frac{1}{4},0,\frac{3}{4}),(4,-2,-1)\} \quad \text{for } i = 1,\ldots p \ .\end{aligned}$$

(The function θ_0 computes the next state, while the functions θ_i compute the next stack operations of stack$_i$, respectively. The actions depend only on the state of the control unit and the symbol being read from each stack. The elements in the range

$$(1,0,0),(\frac{1}{4},0,\frac{1}{4}),(\frac{1}{4},0,\frac{3}{4}),(4,-2,-1)$$

of the θ_i should be interpreted as "no operation", "push0", "push1", and "pop", respectively.)

The set $\mathcal{X} := S \times \mathcal{C}^p$ is called the *instantaneous description set* of $\mathcal{M}$, and the map

$$\mathrm{P} : \mathcal{X} \to \mathcal{X}$$

defined by

$$\begin{aligned}\mathrm{P}(s,q_1,\ldots,q_p) := [\ & \theta_0(s,\zeta[q_1],\ldots,\zeta[q_p],\tau[q_1],\ldots,\tau[q_p]), \\ & \theta_1^T(s,\zeta[q_1],\ldots,\zeta[q_p],\tau[q_1],\ldots,\tau[q_p])\cdot(q_1,\zeta[q_1],1), \\ & \vdots \\ & \theta_p^T(s,\zeta[q_1],\ldots,\zeta[q_p],\tau[q_1],\ldots,\tau[q_p])\cdot(q_p,\zeta[q_p],1)]\end{aligned}$$

where the dot "$\cdot$" indicates inner product, is the *complete dynamics map* of $\mathcal{M}$. As part of the definition, it is assumed that the maps θ_i, $(i = 1,\ldots p)$ are such that $\theta_1(s,\zeta[q_1],\ldots,\zeta[q_p],0,\tau[q_2],\ldots\tau[q_p])$, $\theta_2(s,\zeta[q_1],\ldots,\zeta[q_p],\tau[q_1],0,\tau[q_3],\ldots\tau[q_p])\ldots \neq (4,-2,-1)$ for all $s,q_1,\ldots q_p$ (that is, one does not attempt to pop an empty stack).

Let $\omega \in \{0,1\}^*$ be arbitrary. If there exist a positive integer k, so that starting from the initial state, s_I, with $\delta[\omega]$ on the first stack and empty other stacks, the machine reaches after k steps the

halting state s_H, that is,

$$\mathrm{P}^k(s_I, \delta[\omega], 0, \ldots 0) \;=\; (s_H, \delta[\omega_1], \delta[\omega_2], \ldots, \delta[\omega_p])$$

for some k, then the machine $\mathcal{M}$ is said to *halt on the input* ω. If ω is like this, let k be the least possible number such that

$$\mathrm{P}^k(s_I, \delta[\omega], 0 \ldots 0)$$

has the above form. Then the machine $\mathcal{M}$ is said to *output the string* ω_1, and we let $\phi_{\mathcal{M}}(\omega) := \omega_1$. This defines a partial map

$$\phi_{\mathcal{M}} : \{0,1\}^* \to \{0,1\}^*,$$

the *i/o map* of $\mathcal{M}$. □

Save for the algebraic notation, the partial recursive functions $\phi : \{0,1\}^* \to \{0,1\}^*$ are exactly the same as the maps $\phi_{\mathcal{M}} : \mathcal{C} \to \mathcal{C}$ of p-stack machines as defined here; it is only necessary to identify words in $\{0,1\}^*$ and elements of $\mathcal{C}$ via the above encoding map δ. Our proof will then be based on simulating p-stack machines by processor nets.

5.4 Network with Two Levels: Construction

Assume that a p-stack machine $\mathcal{M}$ is given. Without loss of generality, we assume that the initial state s_I, differs from the halting state s_H (otherwise the function computed is the identity, which can be easily implemented by a net), and we assume that $S := \{0, \ldots, s\}$, with $s_I = 0$ and $s_H = 1$.

We build the net in two stages.

• **Stage 1:** As an intermediate step in the construction, we shall show how to simulate $\mathcal{M}$ with a certain dynamical system over $\mathbb{Q}^{s+p}$. Writing a vector in $\mathbb{Q}^{s+p}$ as

$$(x_1, \ldots, x_s, q_1, \ldots, q_p)\,,$$

the first s components will be used to encode the state of the control unit, with $0 \in S$ corresponding to the zero vector $x_1 = \cdots = x_s = 0$, and $i \in S$, $i \neq 0$ corresponding to the ith canonical vector

$$e_i = (0, \ldots, 0, 1, 0, \ldots, 0)$$

(the "1" is in the ith position). For convenience, we also use the notation $e_0 := 0 \in \mathbb{Q}^s$. The q_i's will encode the contents of the stacks. For notational ease, we substitute $\zeta[t_i]$ and $\tau[t_i]$ by a_i and b_i, respectively. Formally, define

$$\beta_{ij} \ : \ \{0,1\}^{2p} \to \{0,1\},$$

for $\quad i \in \{1,\ldots,s\}\,, \quad j \in \{0,\ldots,s\}$ and

$$\gamma_{ij}^{k} \ : \ \{0,1\}^{2p} \to \{0,1\},$$

for $\quad i = 1,\ldots,p\,, \quad j \in \{0,\ldots,s\}\,, \quad k = 1,2,3,4 \quad$ as follows:

$$\beta_{ij}(a_1,a_2,\ldots,a_p,b_1,b_2,\ldots b_p) = 1 \iff \theta_0(j,a_1,a_2,\ldots,a_p,b_1,b_2,\ldots b_p) = i$$

(intuitively: there is a transition from state j of the control part to state i iff the readings from the stacks are: top of stack$_k$ is a_k, and the nonemptyness test on stack$_k$ gives b_k),

$$\gamma_{ij}^{1}(a_1,a_2,\ldots,a_p,b_1,b_2,\ldots b_p) = 1 \iff \theta_i(j,a_1,a_2,\ldots,a_p,b_1,b_2,\ldots b_p) = (1,0,0)$$

(if the control is in state j and the stack readings are $a_1,a_2,\ldots,a_p,b_1,b_2,\ldots b_p$, then the stack i will not be changed),

$$\gamma_{ij}^{2}(a_1,a_2,\ldots,a_p,b_1,b_2,\ldots b_p) = 1 \iff \theta_i(j,a_1,a_2,\ldots,a_p,b_1,b_2,\ldots b_p) = (\frac{1}{4},0,\frac{1}{4})$$

(if the control is in state j and the stack readings are $a_1,a_2,\ldots,a_p,b_1,b_2,\ldots b_p$, then the operation *Push0* will occur on stack i),

$$\gamma_{ij}^{3}(a_1,a_2,\ldots,a_p,b_1,b_2,\ldots b_p) = 1 \iff \theta_i(j,a_1,a_2,\ldots,a_p,b_1,b_2,\ldots b_p) = (\frac{1}{4},0,\frac{3}{4})$$

(if the control is in state j and the stack readings are $a_1,a_2,\ldots,a_p,b_1,b_2,\ldots b_p$, then the operation *Push1* will occur on stack i),

$$\gamma_{ij}^{4}(a_1,a_2,\ldots,a_p,b_1,b_2,\ldots b_p) = 1 \iff \theta_i(j,a_1,a_2,\ldots,a_p,b_1,b_2,\ldots b_p) = (4,-2,-1)$$

(if the control is in state j and the stack readings are $a_1,a_2,\ldots,a_p,b_1,b_2,\ldots b_p$, then the operation *Pop* will occur on stack i).

Let $\widetilde{\mathcal{P}}$ be the map $\mathbb{Q}^{s+p} \to \mathbb{Q}^{s+p}$:

$$(x_1,\ldots,x_s,q_1,\ldots,q_p) \mapsto (x_1^+,\ldots,x_s^+,q_1^+,\ldots,q_p^+)$$

where, using the notation $x_0 := 1 - \sum_{j=1}^{s} x_j$:

$$x_i^+ := \sum_{j=0}^{s} \beta_{ij}(a_1, \ldots, a_p, b_1, \ldots b_p)\, x_j \tag{5.2}$$

for $i = 1, \ldots, s$ and

$$\begin{aligned} q_i^+ \quad := \sigma \quad & \left(\sum_{j=0}^{s} \gamma_{ij}^1(a_1, \ldots, a_p, b_1, \ldots b_p) x_j \right) q_i + \\ & \left(\sum_{j=0}^{s} \gamma_{ij}^2(a_1, \ldots, a_p, b_1, \ldots b_p) x_j \right) (\frac{1}{4} q_i + \frac{1}{4}) + \\ & \left(\sum_{j=0}^{s} \gamma_{ij}^3(a_1, \ldots, a_p, b_1, \ldots b_p) x_j \right) (\frac{1}{4} q_i + \frac{3}{4}) + \\ & \left(\sum_{j=0}^{s} \gamma_{ij}^4(a_1, \ldots, a_p, b_1, \ldots b_p) x_j \right) (4q_i - 2\zeta[q_i] - 1) \end{aligned} \tag{5.3}$$

for $i = 1, \ldots, p$.

Let $\pi : \mathcal{X} = S \times \mathcal{C}^p \to \mathbb{Q}^{s+p}$ be defined by

$$\pi(i, q_1, \ldots, q_p) := (e_i, q_1, \ldots, q_p).$$

It follows immediately from the construction that

$$\widetilde{\mathcal{P}}(\pi(i, q_1, \ldots, q_p)) = \pi(\mathrm{P}(i, q_1, \ldots, q_p))$$

for all $(i, q_1, \ldots, q_p) \in \mathcal{X}$.

Applied inductively, the above implies that

$$\widetilde{\mathcal{P}}^k(e_0, \delta[\omega], 0, \ldots, 0) = \pi(\mathrm{P}^k(0, \delta[\omega], 0, \ldots, 0))$$

for all k, so $\phi(\omega)$ is defined if and only if for some k it holds that $\widetilde{\mathcal{P}}^k(e_0, \delta[\omega], 0, \ldots, 0)$ has the form

$$(e_1, q_1, \ldots, q_p)$$

(recall that for the original machine, $s_I = 0$ and $s_H = 1$, which map respectively to $e_0 = 0$ and e_1 in the first s coordinates of the corresponding vector in $\mathbb{Q}^{s+p}$). If such a state is reached, then q_1 is in $\mathcal{C}$ and its value is $\delta[\phi(\omega)]$.

• **Stage 2:** The second stage of the construction simulates the dynamics $\widetilde{\mathcal{P}}$ by a net. We first need an easy technical fact.

Lemma 5.4.1 Let $t \in \mathbb{N}$. For each function $\beta : \{0,1\}^t \to \{0,1\}$ there exist vectors

$$v_1, v_2, \ldots, v_{2^t} \in \mathbb{Z}^{t+2}$$

and scalars

$$c_1, c_2, \ldots, c_{2^t} \in \mathbb{Z}$$

such that, for each $d_1, d_2, \ldots, d_t, x \in \{0,1\}$ and each $q \in [0,1]$,

$$\beta(d_1, d_2, \ldots, d_t)x = \sum_{i=1}^{2^t} c_i \sigma(v_i \cdot \mu)$$

and

$$\beta(d_1, d_2, \ldots, d_t)xq = \sigma\left(q + \sum_{i=1}^{2^t} c_i \sigma(v_i \cdot \mu) - 1\right),$$

where we denote $\mu = (1, d_1, d_2, \ldots, d_t, x)$ and "$\cdot$" = dot product in $\mathbb{Z}^{t+2}$.

Proof. Write β as a polynomial

$$\beta(d_1, d_2, \ldots, d_t) = c_1 + c_2 d_1 + \ldots c_{t+1} d_t + c_{t+2} d_1 d_2 + \ldots c_{2^t} d_1 d_2 \cdots d_t \,,$$

expand the product $\beta(d_1, d_2, \ldots, d_t)x$, and use that for any sequence $l_1, \ldots, l_k$ of elements in $\{0,1\}$, one has

$$l_1 \cdots l_k = \sigma(l_1 + \cdots + l_k - k + 1).$$

Using that $x = \sigma(x)$, this gives that

$$\beta(d_1, d_2, \ldots, d_t)x =$$

$$c_1\sigma(x) + c_2\sigma(d_1 + x - 1) + \cdots + c_{2^t}\sigma(d_1 + d_2 + \cdots + d_t + x - t) =$$

$$\sum_{i=1}^{2^t} c_i \sigma(v_i \cdot \mu)$$

for suitable c_i's and v_i's. On the other hand, for each $\tau \in \{0,1\}$ and each $q \in [0,1]$ it holds that $\tau q = \sigma(q + \tau - 1)$ (just check separately for $\tau = 0, 1$), so substituting the above formula with $\tau = \beta(d_1, d_2, \ldots, d_t)x$ gives the desired result. ∎

In the case where $t = 2p$ and the arguments are the top and nonempty functions of the stacks, the arguments are dependent and there is a need of only 3^p terms in the summation, rather than 2^{2p}.

Example 5.4.2 If $p = 2$,

$$(d_1, d_2, d_3, d_4) = (a_1, a_2, b_1, b_2) \ \ (\equiv (\zeta[q_1], \zeta[q_2], \tau[q_1], \tau[q_2])) \ \ ,$$

then the information of the values

$$(1, a_1, a_2, b_1, b_2, a_1a_2, a_1b_2, a_2b_1, b_1b_2)$$

includes the information

$$(a_1b_1, a_2b_2, a_1a_2b_1, a_1a_2b_2, a_1b_1b_2, a_2b_1b_2, a_1a_2b_1b_2) \ .$$

□

We can write the dynamics of the stack q_i as the sum of four components:

$$q_i \ = \ \sum_{j=1}^{4} q_{ij} \ \ , \tag{5.4}$$

where each q_{ij} is the jth term (row) in Equation (5.3). That is, q_{i1} may differ from 0 only if the last operation on stack i was "no-operation." Similarly, the components q_{i2}, q_{i3}, q_{i4} may differ from 0 only if the last operations of the ith stack were push0, push1, or pop, respectively.

Using Lemma 5.4.1, we can write

$$q_{ij} = \sigma \left(\text{next-q}_{ij} + \sum_{k=0}^{s} \gamma_{ik}^{j} x_k - 1 \right) \ \ ,$$

where

$$\text{next-q}_{ij} = \begin{cases} q_i & \text{if } \ j = 1 \\ \frac{1}{4}q_i + \frac{1}{4} & \text{if } \ j = 2 \\ \frac{1}{4}q_i + \frac{3}{4} & \text{if } \ j = 3 \\ 4q_i - 2\zeta[q_i] - 1 & \text{if } \ j = 4 \ \ . \end{cases}$$

Similarly, the top of stack i ($t_i = \zeta[q_i]$) can be expressed as

$$\begin{aligned} t_i \ &= \ \sum_{j=1}^{4} t_{ij} \ \ , \\ t_{ij} \ &= \ \sigma \left(4 \left[\text{next-q}_{ij} + \sum_{k=0}^{s} \gamma_{ik}^{j} x_k - 1 \right] - 2 \right) \ \ , \end{aligned} \tag{5.5}$$

and the nonempty function of stack i ($\tau[q_i]$) as

$$\begin{aligned} e_i &= \sum_{j=1}^{4} e_{ij} \quad , \\ e_{ij} &= \sigma\left(4\left[\text{next-q}_{ij} + \sum_{k=0}^{s} \gamma_{ik}^{j} x_k - 1\right]\right) \quad . \end{aligned} \tag{5.6}$$

Here, t_{ij} is the "top" of the element q_{ij}, and e_{ij} is the nonempty test of the same element. As three out of the four stack elements $\{q_{i1}, q_{i2}, q_{i3}, q_{i4}\}$ of each stack $i = 1, \ldots, p$ are 0, and the fourth has the value of the stack i, it is also the case that three out of four elements of t_i (and e_i) are 0, and the fourth one holds the correct value of the top (nonempty predicate) of the relevant stack.

We construct a network in which the stacks and their readings are not kept explicitly in values q_i, t_i, e_i, but implicitly only via q_{ij}, t_{ij}, e_{ij} , $j = 1, \ldots, 4$, $i = 1, \ldots, p$. By Lemma 5.4.1, all update equations of

$$\begin{aligned} & x_k, \quad k = 1, \ldots, s \qquad \text{(states)} \\ & q_{ij} \qquad i = 1,2 \qquad j = 1,2,3,4 \, , \\ & t_{ij} \qquad i = 1,2 \qquad j = 1,2,3,4 \, , \\ & e_{ij} \qquad i = 1,2 \qquad j = 1,2,3,4 \, . \end{aligned}$$

can be written as

$$\sigma\,(\ \text{lin. comb. of}\ \sigma(\ \text{lin. comb. of tops and nonempty}\ ))\quad ,$$

that is, as what is usually called a "feedforward neural net with one hidden layer."

The main layer consists of the elements q_{ij}, t_{ij}, e_{ij}. In the hidden layer, we compute all elements $\sigma(\cdots)$ required by the lemma to compute the functions β and γ. We showed that $3^p s + 2$ terms of this kind are required. We refer to these terms as "configuration detectors" as they provide the needed combinations of states and stack readings. These terms are all that are required to compute x_k^+. We also keep in the hidden layer the values of q_i, t_i to compute next-q_{ij}.

The result is that $\widetilde{\mathcal{P}}$ can be written as a composition

$$\widetilde{\mathcal{P}} \;=\; F_1 \circ F_2$$

of two "saturated-affine" maps, i.e. maps of the form $\vec{\sigma}(Ax + c)$: $F_1 : \mathbb{Q}^\nu \to \mathbb{Q}^\eta, F_2 : \mathbb{Q}^\eta \to \mathbb{Q}^\nu$, for $\nu = 3^p s + 2p + 2$ and $\eta = s + 12p$.

In summary:

- The main layer consists of:
  1. s neurons x_k, $k = 1, \ldots, s$ that represent the state of the system unarily.
  2. For each stack i, $i = 1, \ldots, p$ we have
     (a) four neurons $q^1_{ij} \equiv q_{ij}, j = 1,2,3,4$,
     (b) four neurons $t^1_{ij} \equiv t_{ij}, j = 1,2,3,4$,
     (c) four neurons $e^1_{ij} \equiv e_{ij}, j = 1,2,3,4$.
- The hidden layer consists of:
  1. $3^p s + 2$ neurons for configuration detecting. (The additional two are for the case of s_0.)
  2. For each stack i, $i = 1, \ldots p$ we have
     (a) a neuron $q^2_i \equiv q_i$,
     (b) a neuron $t^2_i \equiv t_i$.

5.4.1 Universal Network

The number of neurons required to simulate a TM consisting of s states and p stacks is:

$$\underbrace{s + 12p}_{\text{main layer}} \quad + \quad \underbrace{3^p s + 2 + 2p}_{\text{hidden layer}} \ .$$

To approximate the number of processors in a universal processor net, we should calculate the number s discussed above, which is the number of states in the control unit of a two stack universal Turing Machine. Minsky proved the existence of a universal Turing Machine having one tape with 4 letters and 7 control states, [Min67]. Shannon showed in [Sha56] how to control the number of letters and states in a Turing Machine. Following his construction, we obtain a 2-letter 63-state 1-tape Turing Machine. However, we are interested in a two-stack machine rather than one tape. Similar arguments to the ones made by Shannon, but for two stacks, leads us to $s = 84$. Applying the formula $3^p s + s + 14p + 2$, we conclude that there is a universal net with 870 processors. We will see in the appendix of this chapter that to allow for input and output to the network, we need an extra 16 neurons, thus having 886 in a universal machine. (This estimate is very conservative. It

would certainly be interesting to have a better bound. The use of multi-tape Turing Machines may reduce the bound. Furthermore, it is quite possible that with some care in the construction one may be able to drastically reduce this estimate. One useful tool here may be the result in [ADO91] applied to the control unit—here we used a very inefficient simulation.)

5.5 Removing the Sigmoid From the Main Level

Here, we show how to construct an equivalent network to the above, in which neurons in the main level compute linear combinations only. In the following construction, we introduce a set of stack "noisy-elements" $\{\tilde{q}_{i1}, \tilde{q}_{i2}, \tilde{q}_{i3}, \tilde{q}_{i4}\}$ for each stack $i = 1, \ldots p$. These may assume not only values in $[0, 1]$, but also negative values. Negative values of the stacks are interpreted as the value 0, while positive values are the true values of the stack. As in last section, only one of these four elements may assume a non-negative value at each time. The Noisy-top and Noisy-nonempty functions applied to the noisy-elements of the stacks may also produce values outside of the range $[0, 1]$.

To manage with only one level of σ functions, we need to choose a number representation that enforces large enough gaps between valid values of the stacks. We choose base $b = 10p^2$ (p is the number of stacks). Denote $c = 2p + 1$, $b = 10p^2$, $\epsilon_1 = (10p^2 - 1)$, $\epsilon_0 = (10p^2 - 4p - 1)$. The reading functions noisy-top and noisy-nonempty corresponding to the of the jth noisy-element of stack i, are defined as:

$$\text{N-top } (\tilde{q}_{ij}) := c(b\tilde{q}_{ij} - (\epsilon_1 - 1)) \tag{5.7}$$

$$\text{N-nonempty } (\tilde{q}_{ij}) := b\tilde{q}_{ij} - (\epsilon_0 - 1)\,. \tag{5.8}$$

The ranges of values of these functions are

$$\text{N-top } (\tilde{q}_{ij}) \in \begin{cases} [2p+1, 4p+2] & \text{when the top is "1"} \\ [-8p^2 - 2p + 1, -8p^2 + 2] & \text{when the top is "0"} \\ [-\infty, -20p^3 - 10p^2 + 4p + 2] & \text{for an empty stack} \end{cases},$$

$$\text{N-nonempty } (\tilde{q}_{ij}) \in \begin{cases} [4p+1, 4p+2] & \text{when the top is "1"} \\ [1, 2] & \text{when the top is "0"} \\ [-\infty, -10p^2 + 4p + 2] & \text{for an empty stack} \end{cases}.$$

We denote $\tilde{t}_{ij} =$ N-top $(\tilde{q}_{ij})$ and $\tilde{e}_{ij} =$ N-nonempty $(\tilde{q}_{ij})$ for $i = 1, \ldots, p$, $j = 1 \ldots, 4$. Observe that all these ranges are included in

$$R = [-\infty, -8p^2 + 2] \cup [1, 4p + 2] \ ,$$

Note that:

Property: For all $p \geq 2$, any negative value of the functions N-top and N-nonempty has an absolute value of at least $(2p - 1)$ times any positive value of them.

The large negative numbers operate as inhibitors. We will see later how this property assists in constructing the network. As for the possibility of maintaining negative values in stack elements rather than 0, Equation 5.4 is not valid any more. That is, the elements $\tilde{q}_{ij}$, $j = 1, \ldots, 4$ can not be combined linearly to provide the real value of the stack q_i. This is also the case with the expression in Equations 5.7, and 5.8, and thus Equations 5.5 and 5.6 are no longer valid.

Next, we prove that using the noisy-top and noisy-nonempty functions (rather than the binary top and nonempty functions), one may still compute the next value of the stack noisy-elements with one hidden layer only. We consider a function $\beta : R^{8p} \rightarrow \{0, 1\}$ which is sign-invariant. That is, if for all $i = 1, \ldots, p$ and $j = 1, \ldots, 4$

$$\text{sign}\,(t_{ij}) = \text{sign}\,(t'_{ij}) \ \text{ and } \text{sign}\,(e'_{ij}) = \text{sign}\,(e'_{ij})$$

then

$$\beta(t_{11} \ldots, e_{tr}) = \beta(t'_{11} \ldots, e'_{tr}) \ .$$

Here we prove that any such function $\beta(\cdot)$ can be written as a finite sum of the type

$$\sum c_i \sigma(\text{linear combination of the } t_{ij}\text{'s and } e_{ij}\text{'s}). \qquad (\star)$$

That is, the stack noisy-elements are computed correctly using one hidden layer only.

To prove $(\star)$, we state the following (more general) Lemma:

Lemma 5.5.1 For each $t, r \in \mathbb{N}$, let R_t be the range

$$[-\infty, -2t^2 + 2] \cup [1, 2t + 2] \ ,$$

and let

$$S_{r,t} = \{d \mid d = (d_1^{(1)}, \ldots, d_1^{(r)}, d_2^{(1)}, \ldots, d_2^{(r)}, \ldots, d_t^{(1)}, \ldots, d_t^{(r)}) \in R_t^{rt}\ , \quad \text{and}$$

$$\forall i = 1, \ldots, t \ \text{ at most one of } \ d_i^1, \ldots, d_i^r \ \text{ is positive } \}\ .$$

We denote by I the set of multi-indices $(i_1, \ldots, i_t)$, with each $i_j \in \{0, 1, \ldots, r\}$. For each function $\beta : S_{r,t} \to \{0, 1\}$ that is sign invariant, there exist vectors

$$\{v_i \in \mathbb{Z}^{t+2},\ i \in I\}$$

and scalars

$$\{c_i \in \mathbb{Z},\ i \in I\}$$

such that for each $(d_1^{(1)}, \ldots, d_t^{(r)}) \in S_{r,t}$ and any $x \in \{0, 1\}$, we can write

$$\beta(d_1^{(1)}, \ldots, d_t^{(r)})x = \sum_{i \in I} c_i \sigma(v_i \cdot \mu_i)\ ,$$

where

$$\mu_i = \mu_{(i_1, \ldots, i_t)} = (1, d_1^{(i_1)}, d_2^{(i_2)}, \ldots d_t^{(i_t)}, x), \quad \text{and where we are defining} \quad d_i^0 = 0\ . \tag{5.9}$$

Note that $|I| = (r+1)^t$. The operator "$\cdot$" is the dot product in $\mathbb{Z}^{t+2}$. □

Proof. As β is sign-invariant, we can write β—when acting on $S_{r,t}$ as a polynomial

$$\begin{aligned} \beta(d_1^{(1)}, d_1^{(2)}, \ldots, d_t^{(r)}) \quad = \quad & c_1 + c_2\sigma(d_1^{(1)}) + c_3\sigma(d_1^{(2)}) + \cdots + c_{rt+1}\sigma(d_t^{(r)}) + \\ & c_{rt+2}\sigma(d_1^{(1)})\sigma(d_2^{(1)}) + \ldots + c_{(r+1)^t}\sigma(d_1^{(r)})\sigma(d_2^{(r)})\cdots\sigma(d_t^{(r)})\ . \end{aligned}$$

(Note that no term with more than t elements of the type $\sigma(d_j^{(i)})$ appears, as most $\sigma(d_j^{(i)}) = 0$, by definition of $S_{r,t}$. Observe that for any sequence $l_1, \ldots, l_k$ of $(k \leq t)$ elements in R_t and $x \in \{0, 1\}$, one has

$$\sigma(l_1)\cdots\sigma(l_k)x = \sigma(l_1 + \cdots + l_k + k(2t+2)(x-1)).$$

This is due to two facts:

1. The sum of k, $k \leq t$ elements of R_t is non-positive when at least one of the elements is negative. This stems from the property that any negative value in this range is at least $(t-1)$ times larger than any positive value there.

2. Each l_i is bounded by $(2t+2)$.

Expand the product $\beta(d_1^1, \ldots, d_t^r)x$, using the above observation and the fact $x = \sigma(x)$. This gives that

$$\begin{aligned}
&\beta(d_1^{(1)}, \ldots, d_t^{(r)})x = \\
&\quad c_1\sigma(x) + c_2\sigma\left(d_1^{(1)} + (2t+2)(x-1)\right) + \cdots + c_{(r+1)^t}\sigma\left(d_1^{(r)} + \ldots + d_t^{(r)} + t(2t+2)(x-1)\right) \\
&\quad = \sum_{i \in I} c_i\sigma(v_i \cdot \mu_i) \ ,
\end{aligned}$$

for suitable c_i's and v_i's, where μ_i is defined as in 5.9. ∎

Remark 5.5.2 Note that in the case where the arguments are the functions N-top and N-nonempty, the arguments are dependent and not all $(r+1)^t$ terms are needed.

Network Description

The network consists of two levels. The main level consists of both s state neurons that are updated as beforehand, and stack noisy-neurons accompanied by stack noisy-readings neurons: $\tilde{q}_{ij}^1, \tilde{t}_{ij}^1, \tilde{e}_{ij}^1$, $i = 1, \ldots, p$ $j = 1, \ldots, 4$ — representing respectively stack noisy-elements, noisy-top elements, and noisy-nonempty elements. Using the same notations for the functions γ_{ij} as in Equation 5.3, the update equations are

$$\tilde{q}_{ij}^{1+} = \text{next-}\tilde{q}_{ij} + \sum_{k=0}^{s} \gamma_{ik}^j x_k - 1 \ , \tag{5.10}$$

$$\tilde{t}_{ij}^{1+} = (2p+1)\left[10p^2(\text{next-}\tilde{q}_{ij} + \sum_{k=0}^{s} \gamma_{ik}^j x_k - 1) - (10p^2 - 2)\right] \ , \tag{5.11}$$

$$\tilde{e}_{ij}^{1+} = 10p^2(\text{next-}\tilde{q}_{ij} + \sum_{k=0}^{s} \gamma_{ik}^j x_k - 1) - (10p^2 - 4p - 2) \ , \tag{5.12}$$

where

$$\text{next-}\tilde{q}_{ij} = \begin{cases} q_i & \text{if } j = 1 \\ \frac{1}{10p^2} q_i + \frac{10p^2-4p-1}{10p^2} & \text{if } j = 2 \\ \frac{1}{10p^2} q_i + \frac{10p^2-1}{10p^2} & \text{if } j = 3 \\ 10p^2 - 4pt_i - (10p^2 - 4p - 1) & \text{if } j = 4 \end{cases} ,$$

and q_i and t_i are the exact values of the stacks and top elements. Using Lemma 5.5.1, all the expressions of the type γx can be written as linear combinations of terms like $\sigma($ linear combinations of $\tilde{t}^1_{ij}, \tilde{e}^1_{ij})$.

The hidden layer consists of both up to $(5^{2p}s + 2)$ configuration detectors neurons (as proved in Lemma 5.5.1) and the stack and top neurons:

$$q^2_{ij}, t^2_{ij}, \quad i = 1, \ldots, p \quad j = 1, \ldots, 4 \ ,$$

which are updated by the equations

$$\begin{aligned} q^{2+}_{ij} &:= \sigma(\tilde{q}^1_{ij}) && [q_i = \textstyle\sum_{j=1}^4 q^2_{ij}] \ , \\ t^{2+}_{ij} &:= \sigma(\tilde{t}^1_{ij}) \quad i = 1, \ldots, p \quad j = 1, \ldots, 4 && [t_i = \textstyle\sum_{j=1}^4 t^2_{ij}] \ . \end{aligned}$$

5.6 One Level Network Simulates TM

Consider the above network. Remove the main level and leave the hidden level only, while letting each neuron there compute the information that it received beforehand from a neuron at the main level. This can be written as a standard network.

Appendix: Inputs and Outputs

We now explain how to deduce Theorem 4 from Theorem 5. In order to do that, we first show how to modify a net with no inputs into one which, given the input $u_\omega(\cdot)$, produces the encoding $\delta[\omega]$ as a state coordinate and after that emulates the original net. Later we show how the output is decoded. We adapt the input-output convention described in Chapter 3; there are two input lines: $D = u_1$ that carries the data, and $V = u_2$ that validates the data line.

Assume we are given a net with no inputs

$$x^+ = \sigma(Ax + c) \tag{5.13}$$

as in the conclusion of Theorem 5. Suppose that we have already found a net

$$y^+ = \sigma(Fy + gu_1 + hu_2) \tag{5.14}$$

(consisting of 5 processors) so that, if $u_1(\cdot) = D_\omega(\cdot)$ and $u_2(\cdot) = V_\omega(\cdot)$, then with $y(0) = 0$ we have

$$y_4(\cdot) = \underbrace{0\cdots0}_{|\omega|+1}\delta[\omega]00\cdots \quad \text{and} \quad y_5(\cdot) = \underbrace{0\cdots0}_{|\omega|+2}11\cdots,$$

that is,

$$y_4(t) = \begin{cases} \delta[\omega] & \text{if } t = |\omega| + 2 \\ 0 & \text{otherwise,} \end{cases} \quad \text{and} \quad y_5(t) = \begin{cases} 0 & \text{if } t \leq |\omega| + 2 \\ 1 & \text{otherwise.} \end{cases}$$

Once this is done, modify the original net (5.13) as follows. The new state consists of the pair (x, y), with y evolving according to (5.14) and the equations for x modified in this manner (using A_i to denote the ith row of A and c_i for the ith entry of c):

$$\begin{aligned} x_1^+ &= \sigma(A_1x + c_1y_5 + y_4) \\ x_i^+ &= \sigma(A_ix + c_iy_5)\ , \quad i = 2, \ldots, n\ . \end{aligned}$$

Then, starting at the initial state $y = x = 0$, clearly $x_1(t) = 0$ for $t = 0, \ldots, |\omega|+2$ and $x_1(|\omega|+3) = \delta[\omega]$, while, for $i > 1$, $x_i(t) = 0$ for $t = 0, \ldots, |\omega| + 3$.

After time $|\omega| + 3$, as $y_5 \equiv 1$ and $u_1 = u_2 \equiv 0$, the equations for x evolve as in the original net, so $x(t)$ in the new net equals $x(t - |\omega| - 3)$ in the original one for $t \geq |\omega| + 3$.

The system (5.14) can be constructed as follows:

$$\begin{aligned} y_1^+ &= \sigma(\frac{1}{4}y_1 + \frac{1}{2}u_1 + \frac{1}{4} + u_2 - 1) \\ y_2^+ &= \sigma(u_2) \\ y_3^+ &= \sigma(y_2 - u_2) \\ y_4^+ &= \sigma(y_1 + y_2 - u_2 - 1) \\ y_5^+ &= \sigma(y_3 + y_5) \end{aligned}$$

This completes the proof of the encoding part. For the decoding process of producing the output signal o, it will be sufficient to show how to build a net (of dimension 10 and with two inputs) such that, starting at the zero state and if the input sequences are i_1 and i_2, where $i_1(k) = \delta[w]$ for some k and $i_2(t) = 0$ for $t < k$, $i_2(k) = 1$ ($i_1(t) \in [0,1]$ for $t \neq k$, $i_2(t) \in [0,1]$ for $t > k$), then for processors z_9, z_{10} it holds that

$$z_9 = \begin{cases} 1 & \text{if } k+4 \leq t \leq k+3+|\omega| \\ 0 & \text{otherwise ,} \end{cases}$$

and

$$z_{10} = \begin{cases} \omega_{t-k-3} & \text{if } k+4 \leq t \leq k+3+|\omega| \\ 0 & \text{otherwise .} \end{cases}$$

This is easily done with:

$$\begin{aligned} z_1^+ &= \sigma(x_2 + z_1) \\ z_2^+ &= \sigma(z_1) \\ z_3^+ &= \sigma(z_2) \\ z_4^+ &= \sigma(x_1) \\ z_5^+ &= \sigma(z_4 + z_1 - z_2 - 1) \\ z_6^+ &= \sigma(4z_4 + z_1 - 2z_2 - 3) \\ z_7^+ &= \sigma(16z_8 - 8z_7 - 6z_3 + z_6) \\ z_8^+ &= \sigma(4z_8 - 2z_7 - z_3 + z_5) \\ z_9^+ &= \sigma(4z_8) \\ z_{10}^+ &= \sigma(z_7) \,. \end{aligned}$$

In this case the output is $o = (z_{10}, z_9)$.

Remark 5.6.1 If one would also like to achieve a resetting of the whole network after completing the operation, it is possible to add the processor

$$z_{11}^{+} = \sigma(z_9) \,,$$

and to add to each processor that is not identically zero at this point of time,

$$v_i^{+} = \sigma(\ldots - z_{11} + z_9) \,, \; v \in \{x, y, z\} \,,$$

where "..." is the formerly defined operation of the processor.

Chapter 6

Networks with Real Weights

We prove that neural networks with real weights can recognize in polynomial time the same class of languages as those recognized by Turing Machines that consult sparse oracles in polynomial time (the class P/poly); they can recognize all languages, including of course non-computable ones, in exponential time. Furthermore, we show that almost every language requires exponential recognition time. (For simplicity, we give our main results in terms of recognition; it is also possible to provide a more general version regarding the computation of more general functions.)

The proofs of the above results will be consequences of the following equivalence. For functions $T : \mathbb{N} \to \mathbb{N}$ and $S : \mathbb{N} \to \mathbb{N}$, let $\text{NET}_R\,(T)$ be the class of all functions computed by neural networks that may have real weights in *time* $T(n)$ —that is, recognition of strings of length n is in time at most $T(n)$— and let $\text{CIRCUIT}\,(S)$ the class of functions computed by non-uniform families of circuits of *size* $S(n)$ —that is, circuits for input vectors of length n have size at most $S(n)$. We show that if F is so that $F(n) \geq n$, then

$$\text{NET}_R\ (F(n)) \ \subseteq\ \text{CIRCUIT}\ (\text{Poly}(F(n)))$$

and

$$\text{CIRCUIT}\ (F(n)) \ \subseteq\ \text{NET}_R\ (\text{Poly}(F(n)))\,.$$

This equivalence will allow us to make use of results from the theory of (nonuniform) circuit complexity in order to study nets with real weights.

One might ask about using such analog models, or even maybe high order nets, to "solve" NP-hard problems in polynomial time. We introduce a nondeterministic model and show that the equality P = NP in the real nets model is very not likely as it would imply the collapse of the polynomial hierarchy to Σ_2.

The remainder of this chapter will be organized as follows: In Sections 6.1 to 6.3, we show the equivalence between real networks and Boolean circuits. We conclude in Section 6.5 with corollaries. In Section 6.4, we show the equivalence between networks and threshold circuits. As Boolean and threshold circuits are polynomially equivalent, this proof does not add any conceptually new ideas to those in Sections 6.1 to 6.3. Nonetheless, the direct connection and simulation may shed insight when a finer comparison is desired. Furthermore, the proof techniques differ in the two proofs.

6.1 Real Networks And Boolean Circuits

Circuit Families

We briefly recall some of the basic definitions of non-uniform families of circuits. A *Boolean circuit* is a directed acyclic graph. Its nodes of in-degree 0 are called *input nodes*, while the rest are called *gates* and are labeled by one of the Boolean functions AND, OR, or NOT (the first two seen as functions of many variables, the last one as a unary function). One of the nodes, which has no outgoing edges, is designated as the *output* node. The *size* of the circuit is the total number of gates. Adding if necessary extra gates, we assume that nodes are arranged into levels $0, 1, \ldots, d$, where the input nodes are at level zero, the output node is at level d, and each node only has incoming edges from the previous level. The *depth* of the circuit is d, and its *width* is the maximum size of each level. Each gate computes the corresponding Boolean function of the values from the previous level, and the value obtained is considered as an input to be used by the successive level; in this fashion each circuit computes a Boolean function of the inputs.

A *family of circuits* $\mathcal{C}$ is a set of circuits

$$\{c_n, n \in \mathbb{N}\} \ .$$

These have sizes $S_{\mathcal{C}}(n)$, depth $D_{\mathcal{C}}(n)$, and width $W_{\mathcal{C}}(n)$, $n = 1, 2, \ldots$, which are assumed to be monotone nondecreasing functions. If $L \subseteq \{0,1\}^+$, we say that the language L is *computed by the family* $\mathcal{C}$ if the characteristic function of

$$L \bigcap \{0,1\}^n$$

is computed by c_n, for each $n \in \mathbb{N}$.

The qualifier "nonuniform" serves as a reminder that there is no requirement that circuit families be recursively described. It is this lack of classical computability that makes circuits a possible model of resource-bounded "computing," as emphasized in [Par92]. We will show that recurrent neural networks, although more "uniform" in the sense that they have an unchanging physical structure, share exactly the same power.

If L is recognized by the formal net $\mathcal{N}$ in time T, we write $\phi_{\mathcal{N}} = L$ and $T_{\mathcal{N}} = T$. If L is computed by the family of circuits $\mathcal{C}$, we write $\phi_{\mathcal{C}} = L$. We are interested in comparing the functions $T_{\mathcal{N}}$ and $S_{\mathcal{C}}$ for formal nets and circuits so that $\phi_{\mathcal{N}} = \phi_{\mathcal{C}}$.

Statement Of Result

Recall that $\text{NET}_R\,(T(n))$ is the class of languages recognized by formal networks (with real weights) in time $T(n)$ and that $\text{CIRCUIT}\,(S(n))$ is the class of languages recognized by (non-uniform) families of circuits of size $S(n)$.

Theorem 6 *Let F be so that $F(n) \geq n$. Then,* $\text{NET}_R\,(F(n)) \subseteq \text{CIRCUIT}\,(\text{Poly}(F(n)))$, *and* $\text{CIRCUIT}\,(F(n)) \subseteq \text{NET}_R\,(\text{Poly}(F(n)))$. ∎

More precisely, we prove the following two facts. For each function $F(n) \geq n$:

- $\text{CIRCUIT}\,(F(n)) \subseteq \text{NET}_R\,(nF^2(n))$.
- $\text{NET}_R\,(F(n)) \subseteq \text{CIRCUIT}\,(F^3(n))$.

6.2 Circuit Families Are Simulated By Networks

We start by reducing circuit families to networks. The proof will construct a fixed, "universal" net, having roughly $N = 1000$ processors, which, through the setting of a particular real weight which encodes an entire circuit family, can simulate that family.

Theorem 7 *There exists a positive integer N such that the following property holds: For each circuit family $\mathcal{C}$ of size $S_{\mathcal{C}}(n)$ there exists an N-processor formal network $\mathcal{N} = \mathcal{N}(\mathcal{C})$ so that $\phi_{\mathcal{N}} = \phi_{\mathcal{C}}$ and $T_{\mathcal{N}}(n) = O(n\,S^2_{\mathcal{C}}(n))$.*

The proof is provided in the remainder of this section.

6.2.1 The Circuit Encoding

Given a circuit c—with size s, width w, and w_i gates in the ith level—we encode it as a finite sequence over the alphabet $\{1,3,5,7\}$, as follows:

- The encoding of each level i starts with the letter 7. Levels are encoded successively, starting with the bottom level and ending with the top one.
- At each level, gates are encoded successively. The encoding of a gate g consists of three parts—a starting symbol, a 2-digit code for the gate type, and a code to indicate which gate feeds into it:
  - It starts with the letter 1.
  - A two digit sequence $\{53,55,33\}$ denotes the type of the gate, {AND, OR, NOT} respectively.
  - If gate g is in level i, then the input to g is represented as a sequence in $\{3,5\}^{w_{i-1}}$, such that the jth position in the sequence is 5 if and only if the jth gate of the $(i-1)$th level feeds into gate g.

The encoding of a gate g in level i is of length $(w_{i-1}+3)$. The *length* of the encoding of a circuit c is $l(c) \equiv |\mathrm{en}(c)| = O(sw)$.

Example 6.2.1 The circuit c_1 in Figure 6.2.1 is encoded as

$$\mathrm{en}[c_1] = \mathbf{7}\underbrace{153555}_{g_1}\underbrace{155535}_{g_2}\underbrace{133353}_{g_3}\mathbf{7}\underbrace{155555}_{g_4} .$$

For instance, the NOT gate corresponds to the subsequence "133353": It starts with the letter 1, followed by the two digits "33," denoting that the gate is of type NOT, and ends with "353," which indicates that only the second input feeds into the gate. □

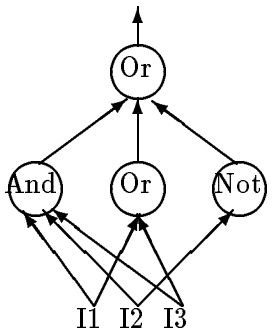

Figure 6.1: Circuit c_1

We encode a non-uniform family of circuits, $\mathcal{C}$, of size $S(n)$ as an infinite sequence

$$e(\mathcal{C}) = 9\ \overline{\text{en}}[c_1]\ 9\ \overline{\text{en}}[c_2]\ 9\ \overline{\text{en}}[c_3]\ \cdots\ , \tag{6.1}$$

where $\overline{\text{en}}[c_i]$ is the encoding of c_i in the reversed order.

Let b be a natural number, and $r = r_1 r_2 \cdots$ a finite or infinite sequence of natural numbers smaller than b. The *interpretation* of the sequence r in base b is the number

$$r|_b \equiv \sum_{i=1}^{\infty} \frac{r_i}{b^i}\ .$$

Generally, two different sequences may result in the same encoding. For instance, both $r = 0999\cdots$ and $r = 1000\cdots$ provide $r|_{10} = 0.1$. However, restricted to the sequences we will consider, the encoding is one-to-one.

We can interpret Formula 6.1 in base 10. We denote this representation of the family of circuits $\mathcal{C}$ as $\hat{\mathcal{C}}$,

$$\hat{\mathcal{C}} = 9\ \overline{\text{en}}[c_1]\ 9\ \overline{\text{en}}[c_2]\ 9\ \overline{\text{en}}[c_3]\ \cdots|_{10}\ . \tag{6.2}$$

Let c_i be the ith circuit in the family. We denote by $\widehat{\text{en}}[c_i]$, the encoding $\text{en}[c_i]$ interpreted in base 10.

Cantor Like Set Encoding

A number which encodes a family of circuits, or one which is a suffix of such an encoding, is a number between 0 and 1. However, not every number in the range $[0, 1]$ can appear in this manner. If the first digit to the right of the decimal point is 1, then the value of the encoding ranges in

$[\frac{1}{10}, \frac{2}{10}]$; if it is 3, the value ranges in $[\frac{3}{10}, \frac{4}{10}]$, and so forth. The number cannot lie in any of the ranges $[\frac{2i}{10}, \frac{2i+1}{10}]$, for $i = 0, 1, 2, 3$. The second digit after the decimal point decides the possible range relative to the currently candidate range. That is, it decides the current range $[\frac{2i+1}{10}, \frac{2i+2}{10}]$ $(i = 0..4)$ just the same way that the first digit did with the range $[0, 1]$. See Figure 6.2.

Figure 6.2: Values of the circuit encoding

The set of possible values is a Cantor set in base 10. Its self-similar structure means that bit (base 10) shifts preserve the "holes."

The advantage of this approach is that, just as when dealing with the rational case, there is never a need to distinguish among two very close numbers in order to read the desired circuit out of the encoding; the circuit can be then retrieved with finite-precision operations employing a finite number of neurons.

6.2.2 A Circuit Retrieval

Lemma 6.2.2 For each (non-uniform) family of circuits $\mathcal{C}$ there exists a processor network $\mathcal{N}_R(\mathcal{C})$ with one input line such that, starting from the zero initial state and given the input signal $u(1) = 1 - 2^{-n}$, $\mathcal{N}_R(\mathcal{C})$ outputs $x_r = \widehat{\text{en}}[c_n]$ after $O(\sum_{i=1}^{n} l(c_i))$.

Proof. We show the existence of a universal network that has the value $\hat{\mathcal{C}}$ as one of its weights. The network retrieves the relevant part of this number. We prove this by providing an NEL program that receives as an input a natural number n and outputs the value $\widehat{\text{en}}[c_n]$ in time $O(\sum_{i=1}^{n} l(c_i))$.

- **Constant:** $\hat{C}$ is a number between $[0,1]$. Interpreted in base 10, it utilizes only the digits $\{1,3,5,7,9\}$.

  Recall that the data structure "stack of n letters" is represented as (Equation 2.4):

  $$l = a_1 a_2 \cdots a_k \hookrightarrow \sum_{i=1}^{k} \frac{2a_i + 1}{(2n)^i} \quad .$$

  A special case for $a_i = \{0,1,2,3,4\}$ is

  $$l_{\{0,1,2,3,4\}} \quad = \quad \sum_{i=0}^{4} \frac{2a_i + 1}{10^i} \, , \tag{6.3}$$

  where the stack digit "0" is represented as 1, "1" as 3, "2" as 5, "3" as 7, and "4" as 9. Thus, a network's weight $\hat{C}$ is equivalent to a NEL's constant of the type "stack over $\{0,1,2,3,4\}$."

- **Input:** An input to the network of the type $u = 1 - 2^{-n}$, where n is a natural number, is equivalent to the NEL's input of type "counter" with the value n. (See Equation 2.6.)

- **Output:** The output of the program is a "stack over $\{0,1,2,3,4\}$," just as with constants. In the network it is a number in $[0,1]$ interpreted as in Equation 6.3.

```
Function Retrieval (value n : counter): Stack of [0...4];
Const  Ĉ: Stack of [0...4];
Var    count: Counter;
       s1, s2: Stack of [0...4];
Begin
   s1 = Ĉ
   Repeat
      If (Top (s1 = 4)) then count = Dec (count)
      s1 = Pop (s1)
   Until (count= 0)
   While (Top (s1) < 4) do
      Begin
         s2 = Push(Top (s1), s2)
         s1 = Pop (s1)
      End
```

Retrieval = s_2

End

Compiling this program into a network, we obtain a universal network architecture. The networks simulating the different circuit families differ only in the one constant encoding the family of circuits: $\hat{\mathcal{C}}$ ∎

We can obtain a more precise lemma:

Lemma 6.2.3 For each (non-uniform) family of circuits $\mathcal{C}$ there exists a 17-processor network $\mathcal{N}_R(\mathcal{C})$ with one input line such that, starting from the zero initial state and given the input signal

$$u(1) = \underbrace{1\,1\cdots 1}_{n}\,0\,0\cdots|_2 = 1-2^{-n}, \quad u(t) = 0 \text{ for } t > 1\ ,$$

$\mathcal{N}_R(\mathcal{C})$ outputs

$$x_r = \underbrace{0\,0\,0\cdots\cdots 0}_{2n+2\sum_{i=1}^{n} l(c_i)+4}\ \widehat{\text{en}}[c_n]\,0\,0\,0\cdots\ .$$

We provide the proof in the appendix of this chapter. The proof demonstrates how useful the language NEL is as a proof technique.

6.2.3 Circuit Simulation By A Network

Let $\omega \in \{0,1\}^n$ be a binary sequence. Denote by $\text{en}[\omega]$ the sequence $\in \{2,4\}^n$ that substitutes $(2\omega_i + 2)$ for each ω_i, and by $\widehat{en}[\omega]$ the interpretation of $\text{en}[\omega]$ in base 9, that is, $\text{en}[\omega]|_9$. We next construct a "universal net" for interpreting circuits.

Lemma 6.2.4 There exists a network $\mathcal{N}_s$, such that for each circuit c and binary sequence ω, starting from the zero initial state and applying the input signal

$$u_1 = \widehat{\text{en}}[c]\,0\,0\cdots \quad u_2 = \widehat{\text{en}}[\omega]\,0\,0\cdots\ ,$$

$\mathcal{N}_s$ outputs

$$x_0 = \underbrace{0\,0\cdots 0}_{T}\,y\,0\,0\cdots \quad x_v = \underbrace{0\,0\cdots 0}_{T}\,1\,0\,0\cdots\ ,$$

where y is the response of circuit c on the input ω, and $T = O(l(c) + |\omega|)$.

Proof. It is easy to verify that, given any circuit, there is a three-tape Turing Machine which can simulate the given circuit in time $O(l(c) + |\omega|)$. This Turing Machine would employ its tapes to store the circuit encoding, the input and output encoding, and the current level's calculation, respectively. Now we can simulate this machine by a net. Indeed, we proved in Chapter 5 that if M is a p-tape Turing Machine with s states which computes in time T a function f on binary input strings, then there exists a rational network $\mathcal{N}$, which consists of

$$9^p s + s + 28p + 2$$

processors, that computes the same function f in time $O(T)$. Closer counting shows that less than 1000 processors suffice. ∎

Remark 6.2.5 If the lemma would only require an estimate of a polynomial number of processors, as opposed to the more precise estimate that we obtain, the proof would have been immediate from the consideration of the *circuit value problem* (CVP). This is the problem of recognizing the set of all pairs $<x, y>$, where $x \in \{0,1\}^+$, and y encodes a circuit with $|x|$ input lines which outputs 1 on input x. It is known that CVP $\in P$ ([BDG90] volume I, pg 110). □

6.2.4 Proof: Circuit Families Are Simulated By Networks

Proof of Theorem 7.

Let $\mathcal{C}$ be a circuit family. We construct the required formal network as a composition of the following three networks:

- An input network, $\mathcal{N}_I$, which receives the input

$$\begin{aligned} u_1 &= \omega\, 0\, 0 \cdots \\ u_2 &= \underbrace{1\, 1 \cdots 1}_{|\omega|}\, 0\, 0 \cdots \ , \end{aligned}$$

and computes $\widehat{\text{en}}[\omega]$ and $u_2|_2$, for each $\omega \in \{0,1\}^+$. This network is trivial to implement.

- A retrieval network, $\mathcal{N}_R(c)$, as described in Lemma 6.2.3, which receives $u_2|_2$ from $\mathcal{N}_I$, and computes $\widehat{\text{en}}[c_{|\omega|}]$. (Note that during the encoding operation, network $\mathcal{N}_I$ produces an output of zero, and $\mathcal{N}_R(c)$ remains in its initial state 0.)

- A simulation network, $\mathcal{N}_S$, as stated in Lemma 6.2.4, which receives $\widehat{\text{en}}[c_{|\omega|}]$ and $\widehat{\text{en}}[\omega|$, and computes

$$x_0 = \underbrace{0\,0\,\cdots\,0}_{T}\ \phi_c(\omega)\,0\,0\,\cdots\quad x_v = \underbrace{0\,0\,\cdots\,0}_{T}\ 1\,0\,0\,\cdots\ .$$

Notice that out of the above three networks, only $\mathcal{N}_R$ depends on the specific family of circuits $\mathcal{C}$. Moreover, all weights can be taken to be rational numbers, except for the one weight that encodes the entire circuit family.

The time complexity involved in the computation of the response of $\mathcal{C}$ to the input ω is dominated by that of retrieving the circuit description. Thus, the complexity is of order

$$T = O\left(\sum_{i=1}^{|\omega|} l(c_i)\right)\ .$$

We remarked that the length of the encoding $l(c_i)$ is of order $O(W_{\mathcal{C}}(i)S_{\mathcal{C}}(i))$, which is itself $O(S^2_{\mathcal{C}}(i))$. Since $S_{\mathcal{C}}(i) \leq S_{\mathcal{C}}(i+1)$ for $i = 1, 2, \ldots$, we achieve the claimed bound $T = O(|\omega|\ S^2_{\mathcal{C}}(|\omega|))$.

Remark 6.2.6 In case of bounded fan-in, the "standard encoding" of circuit c_n is of length $l(c_n) = O(S_{\mathcal{C}}(n)\log(S_{\mathcal{C}}(n)))$. The total running time of the algorithm is then $O(n\,S_{\mathcal{C}}(n)\log(S_{\mathcal{C}}(n)))$. □

6.3 Networks Are Simulated By Circuit Families

We next state the reverse simulation, of nets by nonuniform families of circuits.

Theorem 8 *Let $\mathcal{N}$ be a formal network that computes in time $T : \mathbb{N} \to \mathbb{N}$, where the function T is computable in time $O(T)$. There exists a non-uniform family of circuits $\mathcal{C}(\mathcal{N})$ of size $O(T^3)$, depth $O(Tlog(T))$, and width $O(T^2)$, that accepts the same language as $\mathcal{N}$ does.* ∎

The proof is given in the next two subsections. In the first part, we replace a single formal network by a *family* of formal networks with small rational weights. (This is unrelated to the standard fact for *threshold* gates that weights can be taken to have $n \log n$ bits.) In the second part, we simulate such a family of formal networks by circuits.

6.3.1 Linear Precision Suffices

Define a processor to be a *designated output* processor if its activation value is used as an output of the network (i.e. it is an output processor) and is *not* fed into any other processor. A formal network, for which its two output processors are designated, is called an *output designated network*. Those processors which are not the designated output processors, are called *internal processors*.

For the next result, we introduce the notion of a *q-truncation* net. This is a processor network in which the update equations take the form

$$x_i^+ = q\text{-Truncation } [\sigma(\sum_{j=1}^{N} a_{ij}x_j + \sum_{j=1}^{M} b_{ij}u_j + c_i)] \ ,$$

where q-Truncation means the operation of truncating after q bits.

Lemma 6.3.1 Let $\mathcal{N}$ be an output designated network. If $\mathcal{N}$ computes in time T, there exists a family of $T(n)$-Truncation output designated networks $\mathcal{N}_1(n)$ such that

- For each n, $\mathcal{N}_1(n)$ has the same number of processors and input and output channels as $\mathcal{N}$ does.
- The weights feeding into the internal processors of $\mathcal{N}_1(n)$ are like those of $\mathcal{N}$, but truncated after $O(T(n))$ bits.
- For each designated output processor in $\mathcal{N}$, if this processor computes $x_i^+ = \sigma(f)$, where f is a linear function of processors and inputs, then the respective processor in $\mathcal{N}_1(n)$ computes $\sigma(2\tilde{f} - .5)$, where $\tilde{f}$ is the same as the linear function f but applied instead to the processors of $\mathcal{N}_1(n)$ and with weights truncated at $O(T(n))$ bits.
- The respective output processors of $\mathcal{N}$ and $\mathcal{N}_1(n)$ have the same activation values at all times $t \leq T(n)$.

Proof. We first measure the difference (error) between the activations of the corresponding internal processors of $\mathcal{N}_1(n)$ and $\mathcal{N}$ at time $t \leq T(n)$. This calculation is analogous to that of the chop error in floating point computation, [Atk89].

We use the following notations:

- N is the number of processors, M is the number of input lines, $L \equiv N + M + 1$.
- W' is the largest absolute value of the weights of $\mathcal{N}$, $W \equiv W' + 1$.
- $x_i(t)$ is the value of processor i of network $\mathcal{N}$ at time t.
- $\delta_w \in (0,1)$ and $\delta_p > 0$ are the truncation errors at weights and processors, respectively.
- $\epsilon_t > 0$ is the largest accumulated error at time t in processors of $\mathcal{N}_1(n)$.
- $u \in \{0,1\}^M$ is the input to both $\mathcal{N}$ and $\mathcal{N}_1(n)$. ($u(t) = 0^M$ for $t > n$.)
- a_{ij}, b_{ij}, and c_i are the weights influencing processor i of network $\mathcal{N}$.
- $\tilde{x}_i(t)$, $\tilde{a}_{ij}$, $\tilde{b}_{ij}$, and $\tilde{c}_i$ are the respective activation values of processors, and weights of $\mathcal{N}_1(n)$.

Network $\mathcal{N}_1(n)$ computes at each step

$$\tilde{x}_i^+ \;=\; q\text{-Truncation } [\sigma(\sum_{j=1}^{N} \tilde{a}_{ij}\tilde{x}_j + \sum_{j=1}^{M} \tilde{b}_{ij}u_j + \tilde{c}_i)] \;\; .$$

We assume inductively on t that for all internal processors i, j,

$$\begin{aligned} |\tilde{x}_i(t) - x_i(t)| &\leq \epsilon_t \\ |\tilde{a}_{ij}(t) - a_{ij}(t)| &\leq \delta_w \\ |\tilde{b}_{ij}(t) - b_{ij}(t)| &\leq \delta_w \text{ , and} \\ |\tilde{c}_i(t) - c_i(t)| &\leq \delta_w \text{ .} \end{aligned}$$

Using the global Lipschitz property $|\sigma(a) - \sigma(b)| \leq |a - b|$, it follows that

$$\epsilon_t \;\leq\; N(W' + \delta_w)\epsilon_{t-1} + (N + M + 1)\delta_w + \delta_p \;\leq\; LW\epsilon_{t-1} + L\delta_w + \delta_p \;\; .$$

Therefore,

$$\epsilon_t \;\leq\; \sum_{i=0}^{t-1}(LW)^i(L\delta_w + \delta_p) \;\leq\; (LW)^t(L\delta_w + \delta_p) \;\; .$$

We now analyze the behavior of the output processors. We need to prove that $\sigma(2\tilde{f} - .5) = 0, 1$ when $\sigma(f) = 0, 1$ respectively. That is,

$$f \leq 0 \Longrightarrow \tilde{f} < \frac{1}{4}$$

and

$$f \geq 1 \Longrightarrow \tilde{f} > \frac{3}{4} \ .$$

This happens if $|f - \tilde{f}| < \frac{1}{4}$. Arguing as earlier, the condition $\epsilon_t < \frac{1}{4}$ suffices. This is translated into the requirement

$$(L\delta_w + \delta_p) \leq \frac{1}{4}(LW)^{-t} \ .$$

If both δ_w and δ_p are bounded by $\frac{1}{8}(LW)^{-(t+1)}$, this inequality holds. This happens when the weights and the processor activations are truncated after $O(t\log(LW))$ bits. As L and W are constants, we conclude as desired that a sufficient truncation for a computation of length T is $O(T)$. ∎

6.3.2 The Network Simulation by a Circuit

Lemma 6.3.2 Let $\mathcal{N}_1$ be a family of $T(n)$-Truncation output designated networks, where all networks $\mathcal{N}_1(n)$ consist of N processors and the weights are all rational numbers with $O(T)$ bits. Then, there exists a circuit family $\mathcal{C}$ of size $O(T^3)$, depth $O(T\log(T))$, and width $O(T^2)$, so that c_n accepts the same language as $\mathcal{N}_1(n)$ does on $\{0,1\}^n$.

Proof. We sketch the construction of the circuit c_n which corresponds to the $T(n)$-Truncation output designated net $\mathcal{N}_1(n)$.

The network $\mathcal{N}_1(n)$ has two input lines: data and validation, where the validation line sees n consecutive 1's followed by 0's. We think of the n data bits on the data line which appear simultaneously with the 1's in the validation line, as data input of size n. These n bits are fed simultaneously into c_n via n input nodes.

To simulate the sequential input in $\mathcal{N}_1(n)$, we construct an *input-subcircuit* which preserves the input as it is to be released one bit at a time in later times of the computation. The input subcircuit is of size $nD_{\mathcal{C}}(n)$.

Let

$$p, \ \ p = 1, \ldots, N$$

be a processor of $\mathcal{N}_1(n)$. We associate with each processor p a subcircuit $sc(p)$. Each processor $p \in \mathcal{N}_1(n)$ computes a truncated sum of up to $N+2$ numbers, N of which are multiplications of two T-bit numbers. Hardwiring the weights, we can say that each processor computes a sum of $(TN+2)$ $(2T)$-bit numbers. Using the carry-look-ahead method, [Sav76], the summation can be computed via a subcircuit of depth $O(\log(TN))$, width $O(T^2N)$, and size $O(T^2N)$. (This depth is of the same order as the lower bound when polynomial size is imposed, see [Has87], [Yao85].)

As for the saturation, one gate, p_u, is sufficient for the integer part. As only $O(T)$ bits are preserved, the activation of each processor can be represented in binary by the unit gate, p_u, and the most significant gates

$$p_i,\ i = 1, \ldots O(T)$$

after the operation

$$\text{AND}\,(p_i, \neg(p_u)),\ i = 1, \ldots O(T)\ .$$

Let $sc(p')$ be a subcircuit of largest depth. Pad the other $sc(p)$'s with "demi gates" (e.g. an AND gate of a single input), so that all $sc(p)$'s are of equal depth. The output of circuit $sc(p)$ is called the *activation of* $sc(p)$.

We place the N subcircuits

$$sc(p),\ \ p = 1, \ldots, N$$

to compute in parallel. We call this subcircuit a *layer*. A layer corresponds to one step in the computation of $\mathcal{N}_1(n)$. As $\mathcal{N}_1(n)$ computes in time $T(n)$, $T(n)$ layers are connected sequentially. Each layer i receives the ith input bit from the input-subcircuit, and the N activation values of its preceding layer (except for layer 1, which receives input only). This *main* subarchitecture is of size $O(T^3)$, depth $O(Tlog(T))$, and width $O(T^2)$, where $T = T(n)$.

As $\mathcal{N}_1(n)$ may compute the response to different strings of size n in different times of order $O(T)$, we construct an *output-subcircuit* which forces the response to every string of size n to appear at the top of the circuit.

For each layer $i = 1, \ldots, T$, we apply the AND function to the output of the subcircuits $sc(p1), sc(p2)$, where p_1, p_2 are the output processors of $\mathcal{N}_1(n)$. We transfer these values and apply the OR functions to them. The resulting value is the output of the circuit. When OR is applied at each layer, only $D_{\mathcal{C}}(n)$ gates are needed for this subcircuit.

The resources of the total circuit are dominated by those of the main subarchitecture. ∎

The proof of Theorem 8 follows immediately from Lemma 6.3.1 and Lemma 6.3.2.

6.4 Real Networks Versus Threshold Circuits

A threshold circuit is defined similarly to a Boolean circuit, but the function computed by each node is now a linear threshold function rather than one of the Boolean functions (And, Or, Not). Each gate i computes

$$f_i : \mathbb{B}^{n_i} \mapsto \mathbb{B} ,$$

thus giving rise to the activation updates

$$x_i(t+1) = f_i(x_{i1}, x_{i2}, \ldots, x_{in}) \equiv \mathcal{H}\left(\sum_{j=1}^{n_i} a_{ij} x_{ij}(t) + c_i\right) . \tag{6.4}$$

Here x_{ij} are the activations of the processors feeding into it, and the a_{ij} and c_i are integer constants associated to the gate. Without loss of generality, one may assume that these constants can each be expressed in binary with at most $n_i \log(n_i)$ bits; see [Mur71]. If x_i is on the bottom level, its input is the external input. The function $\mathcal{H}$ is the threshold function

$$\mathcal{H}(z) = \begin{cases} 1 & z \geq 0 \\ 0 & z < 0 \end{cases} . \tag{6.5}$$

The relationships between threshold circuits and Boolean circuits are well studied. (See for example [Par93].) They are known to be polynomial equivalent in size. We provide here an alternative direct relationship between threshold circuits and real networks, without passing through Boolean circuits.

Statement Of Result

Recall that $\text{NET}_R\,(T(n))$ is the class of languages recognized by formal networks (with real weights) in time $T(n)$ and define T-CIRCUIT $(S(n))$ as the class of languages recognized by (non-uniform) families of threshold circuits of size $S(n)$.

Theorem 9 *Let F be so that $F(n) \geq n$. Then,* $\text{NET}_R\,(F(n)) \subseteq$T-CIRCUIT $(\text{Poly}(F(n)))$, *and* T-CIRCUIT $(F(n)) \subseteq \text{NET}_R\,(\text{Poly}(F(n)))$. ∎

More precisely, we prove the following two facts. For each function $F(n) \geq n$:

- T-CIRCUIT $(F(n)) \subseteq$ NET$_R$ $(nF^3(n)\log(F(n)))$.
- NET$_R$ $(F(n)) \subseteq$ T-CIRCUIT $(F^2(n))$.

6.4.1 Families Of Threshold Circuits Are Simulated By Networks

We start by reducing threshold circuit families to networks. The proof will construct a fixed, "universal" net, which, through the setting of a particular real weight which encodes an entire family of threshold circuits, can simulate that family.

Theorem 10 *There exists a positive integer N such that the following property holds: For each family of threshold circuits $\mathcal{C}$ of size $S_{\mathbf{C}}(n)$ there exists an N-processor formal network $\mathcal{N} = \mathcal{N}(\mathcal{C})$ so that $\phi_{\mathcal{N}} = \phi_{\mathcal{C}}$ and $T_{\mathcal{N}}(n) = O(n\, S_{\mathbf{C}}^3(n)\, \log(S_{\mathbf{C}}(n))$.*

The proof is provided in the remainder of this subsection.

The Circuit Encoding

Given a threshold circuit c—with size s, width w, and w_i gates in the ith level—we encode it as a finite sequence over the alphabet $\{1,3,5,7,9\}$, as follows:

- The encoding of each level i starts with the letter 9. Levels are encoded successively, starting with the bottom level and ending with the top one.
- At each level, gates are encoded successively from left to right. The digit 7 separates successive gates of the same level.
- The encoding of each gate includes the encoding of its weights and its bias, from left to right. (The last weight is the bias.)

  The encoding of each weight consists of two parts:

  - its sign ("55" for positive and "5" for negative),

– its absolute value encoded over $\{1,3\}^*$ (rather than $\{0,1\}$).

The jth appearance of either "5" or "55" in a gate encoding starts the encoding of the weight on the connection from the jth gate at the former level to the current gate. If no connection exists between these gates, the letter 1 appears as the absolute value of the weight.

The encoding of a gate g in level i is of length $(w_{i-1}\log(w_{i-1}))$. The *length* of the encoding of a circuit c is $l(c) \equiv |\mathrm{en}(c)| = O(sw^2\log(w))$.

Example 6.4.1 The circuit c_2 in Figure 6.3 is encoded as

$$\mathrm{en}[c_1] = \mathbf{9}\,\underbrace{553\,531\,551\,553}_{g_1}\underbrace{5533\,551\,55311\,53}_{g_2}\,\mathbf{9}\,\underbrace{55331\,55313\,533}_{g_3}\ .$$

Figure 6.3: Circuit c_2

□

We encode a non-uniform family of threshold circuits, $\mathcal{C}$, of size $S(n)$ as an infinite sequence

$$e(\mathcal{C}) = 11\ \overline{\mathrm{en}}[c_1]\ 11\ \overline{\mathrm{en}}[c_2]\ 11\ \overline{\mathrm{en}}[c_3]\ \cdots\ , \tag{6.6}$$

where $\overline{\mathrm{en}}[c_i]$ is the encoding of c_i in the reversed order.

We represent this family $\mathcal{C}$ as $\hat{\mathcal{C}}$, where

$$\hat{\mathcal{C}} = 11\ \overline{\mathrm{en}}[c_1]\ 11\ \overline{\mathrm{en}}[c_2]\ 11\ \overline{\mathrm{en}}[c_3]\ \cdots|_{12}\ . \tag{6.7}$$

¿From here, a similar network to the one developed for simulating Boolean circuits is constructed. (See Subsection 6.2.)

The time complexity to compute the response of $\mathcal{C}$ to the input ω is dominated by that of retrieving the circuit description. Thus, the complexity is of order

$$T = O\left(\sum_{i=1}^{|\omega|} l(c_i)\right) \quad .$$

The length of the encoding $l(c_i)$ is of order $O(W_{\mathcal{C}}^2(i)\log(W_{\mathcal{C}}(i))\,S_{\mathcal{C}}(i))$, which is itself $O(S_{\mathcal{C}}^3(i)\log(S_{\mathcal{C}}(i)))$. Since $S_{\mathcal{C}}(i) \leq S_{\mathcal{C}}(i{+}1)$ for $i = 1, 2, \ldots$, we achieve the claimed bound $T = O(|\omega|\, S_{\mathcal{C}}^3(|\omega|)\log(S_{\mathcal{C}}(|\omega|)))$.

Remark 6.4.2 In case of a fan-in bounded by a constant, the encoding of the threshold circuit c_n is of length $l(c_n) = O(S_{\mathcal{C}}^2(n))$, like in the case of Boolean circuits. □

6.4.2 Networks Are Simulated By Families Of Threshold Circuits

We next state the reverse simulation, of nets by nonuniform families of threshold circuits.

Theorem 11 *Let $\mathcal{N}$ be a formal network that computes in time $T : \mathbb{N} \to \mathbb{N}$. There exists a nonuniform family of threshold circuits $\mathcal{C}(\mathcal{N})$ of size $O(T^2)$, depth $O(T)$, and width $O(T)$, that accepts the same language as $\mathcal{N}$ does.* ■

We start with simulating $\mathcal{N}$ by the family of $T(n)$-Truncation output designated networks $\mathcal{N}_1(n)$ as described in Lemma 6.3.1. Next, we simulate this family of networks of depth $T(n)$ and size $O(T(n))$ via a family of threshold circuits of depth $2T(n)$ and size $O(T^2(n))$.

Assume $\mathcal{N}' \equiv \mathcal{N}_1(n)$ is an m-truncation network for input of size n; $\mathcal{N}'$ has depth $T(n)$ and $m = O(T(n))$. Each gate of $\mathcal{N}'$ computes an addition of N m-bit numbers; then, it applies the σ function to it. Using a technique similar to the one provided in [Par93] pg 156-157, we show how to simulate each σ gate of $\mathcal{N}'$ via a threshold circuit of size $O(m)$ and depth 2. We achieve the simulation in two steps: First we add the N numbers and then we simulate the application of the saturation functions.

Simulating a saturated gate in an m-truncation network by a threshold circuit.

Step 1: Adding N m-bit numbers.

Suppose the numbers are

$$z_1, \ldots z_N \ ,$$

each having m bit representation:

$$z_i = z_{i1} z_{i2} \cdots z_{im} \ .$$

The sum of the N m-bit numbers has $\leq m + \lfloor \log N \rfloor + 1$ bits in the representation. [As the upper bound on the absolute value of the result is $N(2^m - 1)$.] Generally, the sum is

$$\begin{array}{cccccccc} & & & & z_{11} & z_{12} & \cdots & z_{1m} \\ & & & + & & \vdots & & \\ & & & & z_{N1} & z_{N2} & \cdots & z_{Nm} \\ \hline y_{-l} & \cdots & y_{-1} & y_0 & y_1 & y_2 & \cdots & y_m \end{array}$$

As the network is an m-truncation network, we only need to compute $y_0, y_1, \ldots y_m$. We show below how to compute y_k, $k \geq 1$. The circuit for y_0 is very similar.

To compute y_k, we need to consider only z_{ij} for all i and $j \geq k$. Look at the sum:

$$\begin{array}{cccccccc} & & & & & z_{1k} & \cdots & z_{1m} \\ & & & & + & \vdots & & \\ & & & & & z_{Nk} & \cdots & z_{Nm} \\ \hline c_{-l} & \cdots & c_{-1} & c_0 & y_k & & \cdots & y_m \end{array}$$

It is easy to verify that

$$\tilde{z}_k \equiv c_{-l} \cdots c_{-1} \, c_0 \, y_k \cdots y_m = \sum_{i=1}^{N} \sum_{j=k}^{m} (z_{ij} 2^{m-j}) \ .$$

To extract from the sum the y_kth bit, we build the following circuit:

1. **Level 1:** For each possible value i of $c_{-l} \cdots c_{-1} \, c_0$ $(i = 1 \ldots 2^{l+1})$, we have a pair of threshold gates

   $$\tilde{y}_{ki0} = \mathcal{H}(\tilde{z}_k - c_{-l} \cdots c_{-1} \, c_0 \, 1 \, \underbrace{0 \, 0 \cdots 0}_{m-k}) \ , \quad \tilde{y}_{ki1} = \mathcal{H}(-\tilde{z}_k + c_{-l} \cdots c_{-1} \, c_0 \, 1 \, \underbrace{1 \, 1 \cdots 1}_{m-k}) \ .$$

   If $y_k = 0$, exactly one of each pair is active; if $y_k = 1$, one of the pairs has both gates active and the rest one only. Thus, the y_k bit can be computed by counting if more than half of the gates in the first level are active.

2. **Level 2:** It includes one gate only that computes the desired bit:

   $$y_k = \mathcal{H}(\sum_{i=1}^{2^{l+1}} (\tilde{y}_{ki0} + \tilde{y}_{ki1}) - (2^{l+1} + 1)) \ . \tag{6.8}$$

Step 2: Applying the saturated function.

The value of the kth bit is

$$b_k = \begin{cases} y_k & c_0 = 0 \\ 0 & c_0 = 1 \ . \end{cases}$$

First, we have to compute c_0. We allocate 2^l pairs of threshold gates in the first level:

$$\tilde{c}_{ki0} = \mathcal{H}(\tilde{z}_k - c_{-l} \cdots c_{-1} 1 \underbrace{0\,0 \cdots 0}_{m+1-k}) \ , \qquad \tilde{c}_{ki1} = \mathcal{H}(-\tilde{z}_k + c_{-l} \cdots c_{-1} 1 \underbrace{1\,1 \cdots 1}_{m+1-k}) \ .$$

The majority of these gates is the value of c_0:

$$c_0 \equiv \sum_{i=1}^{2^l} (\tilde{c}_{ki0} + \tilde{c}_{ki1}) - 2^l \ .$$

We change Equation 6.8 to compute b_k directly without computing first y_k.

$$b_k = \mathcal{H}(\sum_{i=1}^{2^{l+1}} (\tilde{y}_{ki0} + \tilde{y}_{ki1}) - (2^{l+1} + 1) - c_0) \ . \tag{6.9}$$

The size of the circuit that computes the kth bit is then $O(2^l)$, where $l = \lfloor \log N \rfloor$. We copy this circuit for each of the m bits to simulate one threshold gate. Thus, each σ gate is simulated via a threshold circuit of depth 2 and size $O(m)$. The network itself is hence simulated via Nm copies of those. As $m = O(T)$, and N is considered a constant, the simulating threshold circuit has the size $O(T^2)$, and it doubles the depth of the network $\mathcal{N}'$.

6.5 Corollaries

Let NET-P and NET-EXP be the classes of languages accepted by formal networks in polynomial time and exponential time, respectively. Let CIRCUIT-P and CIRCUIT-EXP be the classes of languages accepted by families of circuits in polynomial and exponential size, respectively.

Corollary 6.5.1 NET-P = CIRCUIT-P and NET-EXP = CIRCUIT-EXP .

The class CIRCUIT-P is often called "P/poly" and coincides with the class of languages recognized by Turing Machine "with advice sequences" in polynomial time. The following corollary states that this class also coincides with the class of languages recognized in polynomial time by Turing Machines that consult oracles, where the oracles are sparse sets. A sparse set S is a set in which

for each length n, the number of words in S of length at most n is bounded by some polynomial function. For instance, any tally set, that is, a subset of 1^*, is an example of a sparse set. The class $P(S)$, for a given sparse set S, is the class of all languages computed by Turing machines in polynomial time and using queries from the oracle S.

From [BDG90], volume I, Theorem 5.5, pg 112, and Corollary 6.5.1, we conclude as follows:

Corollary 6.5.2

$$\text{NET-P} \;=\; \cup_S \text{ sparse } P(S)\,.$$

From [BDG90], volume I, Theorem 5.11, pg 122 (originally, [Mul56]), we conclude as follows:

Corollary 6.5.3 NET-EXP includes all possible binary languages. Furthermore, most Boolean functions require exponential time complexity.

Nondeterministic Neural Networks

The concept of a nondeterministic circuit family is usually defined by means of an extra input, whose role is that of an oracle. Similarly, we define a *nondeterministic network* to be a network having an extra binary input line, the *Guess* line, in addition to the Data and Validation lines. A language L is said to accepted by a nondeterministic formal network $\mathcal{N}$ in time B if

$$L \;=\; \{\omega \,|\, \exists \text{ a guess } \gamma, \phi_{\mathcal{N}}(\omega,\gamma) = 1, T_{\mathcal{N}}(\omega,\gamma) \leq B(|\omega|)\}\,.$$

It is easy to see that Corollary (6.5.1), stated for the deterministic case, holds for the nondeterministic case as well. That is, if we define NET-NP to be the class of languages accepted by nondeterministic formal networks in polynomial time, and CIRCUIT-NP to be the class of languages accepted by nondeterministic non-uniform families of circuits of polynomial size, then:

Corollary 6.5.4 NET-NP $=$ CIRCUIT-NP . □

Since NP$\subseteq$ NET-NP (one may simulate a nondeterministic Turing Machine by a nondeterministic network with rational weights), the equality NET-NP $=$ NET-P implies NP$\subseteq$ CIRCUIT-P $=$ P/poly. Thus, from [KL82] we conclude:

Theorem 12 *If* NET-NP = NET-P *then the polynomial hierarchy collapses to* Σ_2. ∎

The above result says that a theory of computation similar to that which arises in the classical case of Turing machine computation is also possible for our model of analog computation. In particular, even though the two models have very different power, the question of knowing if the verification of solutions to problems is really easier than finding solutions, at the core of modern computational complexity, has a precise corresponding version in our setup, and its solution will be closely related to that of the classical case. Of course, it follows from this that it is quite likely that NET-NP is strictly more powerful than NET-P .

6.6 Appendix

Lemma 6.6.1 For each (non-uniform) family of circuits $\mathcal{C}$ there exists a 17-processor network $\mathcal{N}_R(\mathcal{C})$ with one input line such that, starting from the zero initial state and given the input signal

$$u(1) = \underbrace{1\,1\,\cdots 1}_{n}\;0\,0\,\cdots|_2 = 1 - 2^{-n}, \quad u(t) = 0 \text{ for } t > 1\ ,$$

$\mathcal{N}_R(\mathcal{C})$ outputs

$$x_r = \underbrace{0\,0\,0\;\cdots\cdots 0}_{2n+2\sum_{i=1}^{n} l(c_i)+4}\;\widehat{\text{en}}[c_n]\,0\,0\,0\,\cdots\ .$$

Proof. Let $\Sigma = \{1,3,5,7,9\}$. Denote by $\mathcal{C}_{10}$ the "Cantor 10-set," which consists of all those real numbers q which admit an expansion of the form

$$q = \sum_{i=1}^{\infty} \frac{\omega_i}{10^i} \tag{6.10}$$

with each $\omega_i \in \Sigma$. Let $\Lambda : \mathbb{R} \to [0,1]$ be the function

$$\Lambda[x] := \begin{cases} 0 & \text{if } x < 0 \\ 10x - \lfloor 10x \rfloor & \text{if } 0 \le x \le 1 \\ 1 & \text{if } x > 1\ . \end{cases} \tag{6.11}$$

Let $\Xi : \mathbb{R} \to [0,1]$ be the function

$$\Xi[x] := \begin{cases} 0 & \text{if } x < 0 \\ 2\lfloor 5x \rfloor & \text{if } 0 \le x \le 1 \\ 1 & \text{if } x > 1\ . \end{cases} \tag{6.12}$$

Note that, for each

$$q = \sum_{i=1}^{\infty} \omega_i/10^i \ \in\ \mathcal{C}_{10}\ ,$$

we may think of $\Xi[q]$ as the "select left" operation, since

$$\Xi[q] = \omega_1\ ,$$

and of $\Lambda[q]$ as the "shift left" operation, since

$$\Lambda[q] = \sum_{i=1}^{\infty} \omega_{i+1}/10^i \ \in\ \mathcal{C}_{10}\ .$$

For each $i \geq 0$, $q \in \mathcal{C}_9$,

$$\Xi[\Lambda^i[q]] = \omega_{i+1} \ .$$

The following procedure summarizes the task to be performed by the network constructed below, which in turn satisfies the requirements of the lemma.

Procedure Retrieval($\hat{C}, n$)
Variables counter, y, z
Begin
counter $\leftarrow 0$, y $\leftarrow 0$, z $\leftarrow \hat{C}$,
While counter $< n$
Parbegin
$z \leftarrow \Lambda[z]$
if $\Xi[z] = 9$ then increment counter
Parend,
While $\Xi[z] < 9$
Parbegin
$z \leftarrow \Lambda[z]$
$y \leftarrow \frac{1}{10}(y + \Xi[z])$
Parend,
Return(y)
End

The functions Λ and Ξ can not be programmed within the neural network model due to their discontinuity. However, we can program the functions $\tilde{\Lambda}, \tilde{\Xi}$, which coincide with Λ, Ξ respectively on $\mathcal{C}_{10}$:

$$\tilde{\Lambda}[q] = \sum_{j=1}^{9} (-1)^{j-1} \sigma(10q - j) \ , \tag{6.13}$$

and

$$\tilde{\Xi}[q] = 1 + 2 \sum_{j=1}^{4} \sigma(10q - 2j) \ . \tag{6.14}$$

The retrieval procedure is, then, achieved by the following network:

$$x_i^+ \ = \ \sigma(10x_{10} - i - 1) \quad i = 0, \ldots, 8$$

$$
\begin{aligned}
x_9^+ &= \sigma(2u) \\
x_{10}^+ &= \sigma(\hat{C}x_9 + x_0 - x_1 + x_2 - x_3 + x_4 - x_5 + x_6 - x_7 + x_8) \\
x_{11}^+ &= \sigma(\frac{1}{10}x_{12} + \frac{2}{10}(x_1 + x_3 + x_5 + x_7 + \frac{x_{17}}{2}) - 2x_{13}) \\
x_{12}^+ &= \sigma(x_{11}) \\
x_{13}^+ &= \sigma(u + x_{14} + x_{15}) \\
x_{14}^+ &= \sigma(2x_{13} + x_7 - 2) \\
x_{15}^+ &= \sigma(x_{13} - x_7) \\
x_{16}^+ &= \sigma(x_{12} + x_7 - 1) \\
x_{17}^+ &= \sigma(x_{12})\,.
\end{aligned}
$$

If the input u arrives at time 1, then $x_{10}(2k+3) = \tilde{\Lambda}^k[\hat{C}]$ (because of Equation 6.13). Processors x_{13}, x_{14}, x_{15} serve to implement the counter, and processor x_{16} is the output processor. This network satisfies the requirements of the lemma. ∎

Chapter 7

Kolmogorov Weights: Between P and P/poly

In previous chapters, we showed that the computational power of neural networks depends on the type of numbers utilized as weights. When the networks compute in polynomial time, the computational power of these networks happens to coincide with the classes P and P/poly for networks with rational and real weights, respectively.

It may be argued that any net with real weights that is computationally feasible to implement must admit a "short" description of its real-valued weights. It is therefore interesting to have characterizations of the accepted languages in terms of the amount of information and resources required to specify these reals. We next show that this issue is closely linked to a natural question regarding nonuniform complexity classes, namely, the possibility of bounding the amount of advice allowed for machines in the class.

Thus we set bounds on the resource-bounded Kolmogorov complexity of the reals used as weights in neural nets, and then prove that such bounds correspond precisely to the amount of advice allowed to nonuniform classes lying between P and P/poly, as studied previously in [BHM92]. It is known that P/poly and some of its subclasses can be characterized by polynomial time computation using tally oracles. This motivates us to compare various real-weight neural models in terms of the Kolmogorov complexity of these tally oracles. Using such Kolmogorov complexity arguments, we prove that there exists a proper hierarchy of complexity classes defined by neural nets whose weights have increasing Kolmogorov complexity.

7.1 Statement of Results

We define different classes of computable numbers by considering different time constraints and amounts of information in their construction. Our definition of Kolmogorov complexity of infinite

sequences is a time-bounded analog of that in [Kob81]. We denote by $w_{1:k}$ the word consisting of the first k symbols of w.

Definition 7.1.1 Fix a universal Turing Machine U. Let f be a nondecreasing function, g a time-constructible function, and $\alpha \in \{0,1\}^\infty$. We say that $\alpha \in K[f(n), g(n)]$ if there exists $\beta \in \{0,1\}^\infty$ such that, for every n, the universal machine U outputs $\alpha_{1:n}$ in time $g(n)$, when given $\beta_{1:f(n)}$ and n as inputs. If no condition is imposed on the running time, we say $\alpha \in K[f(n)]$.

Observe that here the length of the output is provided for free to the universal machine; so our definition corresponds to the complexity measure more commonly called "Kolmogorov complexity relative to the length." The reason is that we want simple numbers (e.g. rationals) to have extremely low complexity (e.g. constant), and the information contained in the length of a string could be higher. However, the definitions are equivalent (modulo constants) for complexities at least logarithmic.

Generally, $K[\mathcal{F}, \mathcal{G}]$ is the set of all infinite binary sequences taken from $K[f, g]$ where $f \in \mathcal{F}$ and $g \in \mathcal{G}$. For example, a sequence is in $K[log, poly]$ if its prefixes are computable from logarithmically long prefixes of some other sequence in polynomial time.

In what follows, we denote by $\{0,1\}^\#$ the set of both finite and infinite binary strings.

Define a function

$$\delta_4 : \{0,1\}^\# \to [0,1]$$

by the formula

$$\delta_4(\epsilon) = 0 \qquad \delta_4(\alpha) = \sum_{i=1}^{|\alpha|} \frac{2\alpha_i + 1}{4^i} \quad .$$

Here ϵ is the empty string; $|\alpha|$ is the length of the string α, and can be either a finite value or ∞; α_i is the ith bit of the string α. The map δ_4 is injective on $\{0,1\}^\#$, and its image is the set:

$$\left\{ \sum_{i=1}^{n} \frac{\beta_i}{4^i} \mid \beta_i \in \{1,3\} \right\}_{n \in \mathbb{N} \cup \{0\} \cup \{\infty\}} .$$

The inverse map δ_4^{-1} is well defined there. Let Δ_4 be the range of this function when restricted to the domain of infinite strings. That is,

$$\Delta_4 \equiv \left\{ \sum_{i=1}^{\infty} \frac{\beta_i}{4^i} \mid \quad \beta_i \in \{1,3\} \right\} .$$

Thus, δ_4 can be used to define the Kolmogorov complexity of numbers in Δ_4: A number $\omega \in \Delta_4$ is said to be in $K[f(n), g(n)]$ iff $\delta^{-1}(\omega) \in K[f(n), g(n)]$.

The main contribution here is to show that the Kolmogorov complexity of the weights of nets relates to a structural notion: the amount of advice for nonuniform classes. Important consequences follow; for instance, we can prove the "hierarchy" theorem stated below. We say that a class $\mathcal{F}$ of functions is *closed under* $O(.)$ if for every f, g, if $g \in O(f)$ and $f \in \mathcal{F}$, then $g \in \mathcal{F}$.

Theorem 13 *Let $\mathcal{F}, \mathcal{G}$ be two function classes closed under $O(.)$, for which there exists some $s \in \mathcal{G}, s = o(n)$, such that the following property holds: $\forall p \in poly$, $\forall r \in \mathcal{F}, r(p(n)) = o(s(n))$. Let $\mathcal{N}_{K[\mathcal{F},poly]}$ be the set of networks that compute in polynomial time, and each of which uses weights from $K[\mathcal{F}, poly] \bigcup \mathbb{Q}$. Let $\mathcal{L}(\mathcal{N}_{K[\mathcal{F},poly]})$ be the class of languages accepted by $\mathcal{N}_{K[\mathcal{F},poly]}$, and similarly for $\mathcal{G}$. Then:*

$$\mathcal{L}(\mathcal{N}_{K[\mathcal{F},poly]}) \neq \mathcal{L}(\mathcal{N}_{K[\mathcal{G},poly]}) .$$

∎

In Section 7.2, we establish an equivalence between oracle TMs and networks. We then introduce in Section 7.3 a hierarchy in these sets of oracle TMs, thus concluding Theorem 13.

7.2 Equivalence of TMs with Tally Oracles and NNs

Definition 7.2.1 Let $S \subseteq \{0,1\}^{\#}$. The set S is *closed under mixing* if for any finite number $k \in \mathbb{N}$ and for any k strings from S,

$$\alpha^1 = \alpha_1^1 \alpha_2^1 \alpha_3^1 \cdots ,$$

$$\alpha^2 = \alpha_1^2 \alpha_2^2 \alpha_3^2 \cdots ,$$

$$\cdots ,$$

$$\alpha^k = \alpha_1^k \alpha_2^k \alpha_3^k \cdots$$

the shuffled string

$$\alpha_1^1 \alpha_1^2 \alpha_1^3 \cdots \alpha_1^k \, \alpha_2^1 \alpha_2^2 \alpha_2^3 \cdots \alpha_2^k \, \alpha_3^1 \alpha_3^2 \cdots$$

is again an element of S.

We recall the definition of the characteristic string of a set. If S is a set of strings in Σ^*, $\chi_S \in \{0,1\}^\infty$ is the characteristic sequence of S, defined in the standard way: the i^{th} bit of the sequence is 1 if and only if the i^{th} word of Σ^* in the lexicographic order is in S. We choose Σ as the smallest alphabet containing all the symbols occurring in words of S, so that for a tally set T, χ_T denotes the characteristic sequence of T relative to $\{0\}^*$.

Definition 7.2.2 Let $T \subseteq \{0,1\}^\star$. We define the *characteristic number* of T as

$$T_4 = \delta_4(\chi_T) \in \Delta_4 \ ,$$

where χ_T is the characteristic string of T.

The main theorem of this section is as follows:

Theorem 14 *Let $S \subseteq \{0,1\}^\infty$ be closed under mixing and let $\mathcal{T}$ be the class of tally sets*

$$\mathcal{T} = \{\ T\ :\ \chi_T \in S\ \}\ .$$

Time in the following models is polynomially related:

1. *Oracle Turing Machines that consult oracles in $\mathcal{T}$.*

2. *Neural networks that have all weights in the set $\delta_4(S) \bigcup \mathbb{Q}$.*

∎

Some interesting special cases arise when considering various natural bounds for the Kolmogorov complexity:

- $S = K[n, \text{poly}]$, that is, arbitrary strings. The class of languages accepted in this case is P/poly; this is the main result of Chapter 6.

- $S = K[1, \mathrm{poly}]$, that is, the sets of strings computable in polynomial time. The class of languages accepted in this class is P, as proved in Chapter 5.

- $S = K[\log, \mathrm{poly}]$. In this case, the class of languages accepted is Full-P/log, described by Balcázar, Hermo and Mayordomo [BHM92]. This is the class of TMs that receive logarithmically long advices, where each advice aimed at words of length n is appropriate for all shorter words as well. More formally, the class Full-P/log consists of those sets A for which there is some $B_A \in \mathrm{P}$ and $h_A \in \mathrm{LOG}$ (LOG is the class of functions f defined on the naturals, such that for some constant c, $f(n) \leq c \log n$ for all $n \in \mathbb{N}$) such that:

$$\forall n \, \exists \omega_n \, (|\omega_n| \leq h_A(n)) \, \forall x \, (|x| \leq n)$$
$$x \in A \iff < x, \omega_n > \in B_A \; .$$

  This class was shown in [BHM92] to have several interesting features. In particular, Theorem 17 there states that Full-P/log coincides with the class of languages recognized in polynomial time by TMs that consult tally sets, having characteristic sequences in $K[\log, \mathrm{poly}]$. The equivalence there implies this observation.

The next two subsections prove Theorem 14.

7.2.1 Proof: $1 \subseteq 2$

Next, we denote by Δ_4^* the range $\delta_4(\{0,1\}^*)$.

Definition 7.2.3 An *oracle neural network* (ONN) is a network $\mathcal{N}$ with three additional special "oracle neurons" Q, A, W — called the query, answer, and wait neurons — and a particular *oracle number* Y. The above take their values in the following sets:

$$Q \in \Delta_4^*, \;\; A \in \{0,1\}, \;\; W \in \{0,1\}, \;\; Y \in \Delta_4 \; .$$

- The network operates regularly as long as $W = 0$. When $W = 1$, the activations in the network $\mathcal{N}$ are not being changed.

- The network can set W to 1 but cannot reset it.

- When $W = 1$, the three oracle neurons change as follows:

$$A^+ = (\delta_4^{-1}(Y))_{\mathrm{lex}(Q)}$$

$$Q^+ = 0$$

$$W^+ = 0$$

where $\mathrm{lex}(Q)$ is the lexicographic index of Q in Δ_4^*. Other neurons of $\mathcal{N}$ do not change.

Setting $W = 1$ is like invoking a subroutine for solving a membership query. Y can be thought of as the characteristic number of an oracle set Y', and the subroutine tests whether $\delta_4^{-1}(Q) \in Y'$. The model assumes that this oracle subroutine answers in unit time.

Lemma 7.2.4 Let T be a tally set. Time in the following models is polynomially related.

- Oracle TM that consults the tally set T.
- Oracle NN with all weights in $\mathbb{Q}$ and oracle number T_4.

The proof of this lemma is completely analogous to the proof of the main result in Chapter 5, and is not included here. The use of a base-4 Cantor set is critical in being able to implement the desired subroutines, in particular, conditional statements.

Lemma 7.2.5 For each number $T_4 \in \Delta_4$, there exists a network with two inputs — denoted by u_1, u_2 — and two outputs (y_1 and y_2) which when started from the zero initial value, and given the input signals

$$\begin{aligned} u_1 &= [\sum_{i=1}^{n} (\frac{1}{4})^i] \ 0 \ 0 \ 0 \ \cdots \\ u_2 &= 1 \ 0 \ 0 \ 0 \ \cdots , \end{aligned}$$

outputs

$$\begin{aligned} y_1 &= \underbrace{0\,0\,\cdots 0}_{v} \ b \ 0 \ 0 \ \cdots \\ y_2 &= \underbrace{0\,0\,\cdots 0}_{v} \ 1 \ 0 \ 0 \ \cdots , \end{aligned}$$

where b is the truth value of $\delta_4^{-1}(u_1(1)) \in T$, for the tally set T that has T_4 as its characteristic number; and $v = O(n)$.

Proof. We use T_4 as one of the weights of the network. Notice that

$$\begin{aligned} u_1(1) &= .\underbrace{11\cdots1}_{n} \quad \text{in base 4 ,} \\ T_4 &= .3133113\cdots \quad \text{in base 4 ,} \end{aligned}$$

and the nth digit of T_4 in the base 4 expansion labels the decision of whether $\delta_4^{-1}(\sum_{i=1}^{n}(\frac{1}{4})^i) \in T$.

The following NEL program computes the required value $\delta_4^{-1}(u_1(1)) \in T$.

Function Query: Boolean

Var

c : counter;

s : stack of Boolean;

Begin

Read (u_1, c);

$s := T_4$;

For $i = 1$ to $(c-1)$ **do**

Pop (T_4) ;

Query := **Top** (T_4);

End;

By Theorem 1, the constants of the network associated with this program are rationals except for the constant T_4. ∎

Using the above two lemmas, we can prove the inclusion $1 \subseteq 2$ as follows: let M be an OTM that uses a tally set T as an oracle, where $\chi_T \in S$. We construct a network $\mathcal{N}$ that accepts the same language and has all its weights in $\delta_4(S) \bigcup \mathbb{Q}$. The network $\mathcal{N}$ consists of two subnetworks: $\mathcal{N}_1$ is an oracle network that consults the oracle number T_4, and $\mathcal{N}_2$ is the retrieval network of T_4 as described in Lemma 7.2.5. (They are coupled with a few neurons, this should be clear.) By Lemma 7.2.4, $\mathcal{N}_1$ has only rational weights, and by Lemma 7.2.5, $\mathcal{N}_2$ has both rational weights and the weight $T_4 \in \delta_4(S)$. The network $\mathcal{N}_1$ simulates M in linear time (Chapter 5), while $\mathcal{N}_2$ has a total

computation time bounded by $O(\sum |\text{queries}|)$ — which is bounded by the computation time of M. Thus, given an OTM with an oracle in S, there is a corresponding neural network whose weights are either rationals or in the set $\delta_4(S)$, that computes the same language with no more than linear slowdown in the computation. ∎

7.2.2 Proof: $2 \subseteq 1$

Assume that we are given a network $\mathcal{N}$ with weights in $\delta_4(S) \bigcup \mathbb{Q}$. The network has a fixed number $k' \in \mathbb{N}$ of weights, which can be written in base four expansion as:

$$\begin{aligned} \omega^1 &= .\omega_1^1\omega_2^1\omega_3^1\cdots, \\ \omega^2 &= .\omega_1^2\omega_2^2\omega_3^2\cdots, \\ &\cdots, \\ \omega^{k'} &= .\omega_1^{k'}\omega_2^{k'}\omega_3^{k'}\cdots. \end{aligned}$$

Assume w.l.o.g. that the first k of them are in $\delta_4(S)$, that is, $\omega_j^i \in \{1,3\}$ for such weights i. (The weights $\omega^{k+1}, \ldots, \omega^{k'}$ are rationals.) As S is closed under mixing, the string

$$\alpha = \frac{\omega_1^1-1}{2}\frac{\omega_1^2-1}{2}\frac{\omega_1^3-1}{2}\cdots\frac{\omega_1^k-1}{2}\frac{\omega_2^1-1}{2}\frac{\omega_2^2-1}{2}\frac{\omega_2^3-1}{2}\cdots\frac{\omega_2^k-1}{2}\frac{\omega_3^1-1}{2}\frac{\omega_3^2-1}{2}\frac{\omega_3^3-1}{2}\cdots$$

is again an element of S.

We show the existence of an oracle TM M that consults a tally set with characteristic string $\chi_T = \alpha$, and simulates the network $\mathcal{N}$ while keeping the polynomial time constraint. The machine M performs the following operations:

1. M receives the input string x.
2. M computes the running time $B(|x|)$ of $\mathcal{N}$.
3. For a certain constant C, M executes:

   For $i = 1$ to $kCB(|x|)$

   query the ith word of α.

After these operations, M has the weights of $\mathcal{N}$ up to a precision $CB(|x|)$. Here, C is a constant such that this precision suffices. The existence of such a C was proved in Chapter 6. (The $(k' - k)$ rational weights are encoded in the machine M.)

4. Finally, M simulates $\mathcal{N}$ step by step in polynomial time.

■

7.3 Hierarchy of TMs That Consult Tally Oracles

All we are left in order to prove the proper hierarchy of computational classes associated to networks, is to define a hierarchy of subsets of Δ_4, and prove that oracle TMs which consult the associated tally sets result in computationally different classes.

We define a partial order of function classes. Let $\mathcal{F}, \mathcal{G}$ be function classes. We say that $\mathcal{F} \prec \mathcal{G}$ if there is some $s \in \mathcal{G}$ so that $s = o(n)$ and, for every polynomial p and every $r \in \mathcal{F}$, $r(p(n)) = o(s(n))$.

Note that this partial order defines a proper hierarchy. For instance, one may consider the function classes $\theta_i = \{q_1, \ldots, q_i\}$, where $q_i = \log^{(i)}$ is defined inductively by $q_1 = \log$ and $q_i = \log(q_{i-1})$ for $i > 1$.

For the next theorem, we denote by $\mathrm{P}(T_{\mathcal{F}})$ the class of TMs that compute in polynomial time, where each TM consults a tally set T such that $\chi_T \in K[f, \mathrm{poly}]$ and $f \in \mathcal{F}$. We also denote by $L(\mathrm{P}(T_{\mathcal{F}}))$ the class of languages computed by these TMs.

Theorem 15 *Let $\mathcal{F}, \mathcal{G}$ be function classes, such that $\mathcal{F} \prec \mathcal{G}$. Then, $\mathcal{L}(P(T_{\mathcal{F}}))$ is properly included in $\mathcal{L}(P(T_{\mathcal{G}}))$.*

Proof. We define a set $\mathcal{A} \in \mathcal{L}(\mathrm{P}(T_{\mathcal{G}}))$ but not in $\mathcal{L}(\mathrm{P}(T_{\mathcal{F}}))$. Let $s(n)$ be as in the theorem. Choose an infinite sequence $\gamma \notin K[n/2]$. For each n define the string β_n as

$$\beta_n = \gamma_{1:s(n)/2} \cdot 0^{n - s(n)/2} ,$$

if $n \geq s(n)/2$, and $\beta_n = 0^n$ otherwise.

Let $\mathcal{A}$ be the tally set with characteristic string $\beta_1\beta_2\beta_3\ldots$ Given $\gamma_{1:s(n)/2}$ it is easy to build $\chi_{\mathcal{A}_{1:n}}$, so that

$$\chi_{\mathcal{A}} \in K[s(n)/2 + c, q(n)] \subseteq K[s(n), q(n)] \,,$$

for some constant c and polynomial q. Hence, $\mathcal{A} \in \mathcal{L}(\mathrm{P}(T_{\mathcal{G}}))$.

However, $\mathcal{A} \notin \mathcal{L}(\mathrm{P}(T_{\mathcal{F}}))$. Assume otherwise, then there is some machine that prints $\gamma_{1:s(n)/2}$ in time $p_1(n)$, querying at most the first $p_1(n)$ elements of a tally set T, with $\chi_T \in K[r(n), p_2(n)]$, p_1, p_2 polynomials, and $r \in \mathcal{G}$. Then $\gamma_{1:s(n)/2}$ is obtained from the first $r(p_1(n)) + O(1) < s(n)/4$ bits of χ_T, in fact in time $O(p_2(p_1(n)))$. This contradicts the choice of γ. ∎

Theorem 14 establishes a connection between networks with weights from some set and oracle Turing machines that consult related tally sets. Theorem 15 displays a the hierarchy of oracle Turing machines whose their oracles belong to different Kolmogorov complexity classes. To prove the hierarchy in networks in terms of Kolmogorov complexity of their weights—that is, to prove Theorem 13—it suffices to show that Theorem 14 is applicable to Kolmogorov classes. That is, that the classes of the form $K[\mathcal{F}, \mathrm{poly}]$ for $\mathcal{F}$ which is closed under $O(.)$, are closed under mixing.

Lemma 7.3.1 Let $\mathcal{F}$ be a function class that is closed under $O(.)$. Then, $K[\mathcal{F}, \mathrm{poly}]$ is closed under mixing.

Proof. Let $k \in \mathbb{N}$ and $\alpha^1, \ldots, \alpha^k \in K[\mathcal{F}, \mathrm{poly}]$. That is, for all $i = 1, \ldots k$, $\alpha^i_{1:n}$ is computed in $g_i(n) \in \mathrm{P}$ time using the input $\beta^i_{1:f_i(n)}$. Thus, the shuffled string of the α's ($\tilde{\alpha}$) can be computed in time $g = O(n) + \sum_{i=1}^{k} g_i(\frac{n}{k})$ from an input which is equal to the prefix length $f = \sum_{i=1}^{k} f_i(\frac{n}{k})$ of the shuffled string of the β's ($\tilde{\beta}$). It is easy to verify that as both $\mathcal{F}$ and P are closed under $O(.)$, $g \in \mathrm{P}$ and $f \in \mathcal{F}$. Thus, $\tilde{\alpha} \in K[\mathcal{F}, \mathrm{poly}]$. ∎

Chapter 8

Equivalence of Different Dynamical Systems

The material in this chapter could be seen as justifying a "Church's thesis" equivalence for analog computing. We show that a large class of different networks and dynamical systems has no more computational power than our neural (first-order) model with real weights. Analogously to Church's thesis of computability (see e.g. [Yas71] p.98), our results suggest the following Thesis of Time-bounded Analog Computing: "Any reasonable analog computer will have no more power (up to polynomial time) than first-order recurrent networks."

In previous chapters, we studied the computational power of the neural network model (see Equation 1.2) presented in Chapter 1. As our model is highly homogeneous and extremely simple, one may suspect that it is weaker than other possible more complex models. For example, in many applications of neural networks to language recognition, each neuron is allowed to compute inside its sigma function a polynomial combination of its input values (see Equation 1.1) rather than affine combinations only. Furthermore, in both applications and biologically plausible models, the activation function is usually more complicated than the saturated-linear function used in our model; for instance, one encounters the classical sigmoid $\frac{1}{1+e^{-x}}$ or other activations.

We show that if one allows multiplications in addition to only linear operations in each neuron, that is, if one considers instead what are often called *high order* (or sigma-pi) neural nets, the computational power does not increase. Even further, and perhaps more surprising, no increase in computational power (up to polynomial time) can be achieved by letting the activation function be not necessarily the simple saturated linear one, but any function which satisfies certain reasonable assumptions. Also, no increase results even if the activation functions are not necessarily identical in the different processors.

We remark also that the neural network models that we study have a weak property of "robustness" to noise and to implementation error, in the sense that small enough changes in the network

would not affect the computation for finite times. This robustness includes changes in the precise form of the activation function, in the weights of the network, and even an error in the update. In classical models of (digital) computation, this type of robustness can not even be properly defined.

8.1 Generalized Networks: Definition

We consider dynamical systems –which we will call *generalized processor networks*– with far less restrictive structure than the recurrent neural network model which was described in Chapter 1. We show that these networks are not more powerful, up to polynomial time slowdown, than the previously considered model.

Let N, M, p be natural numbers. A *generalized processor network* is a dynamical system D that consists of N processors $x_1, x_2, \ldots x_N$, and receives its input $u_1(t), u_2(t), \ldots u_M(t)$ via M input lines. A subset of the N processors, say $x_{i1}, \ldots, x_{ip}$, is the set of output processors of the system, used to communicate the output of the system to the environment. In vector form, a generalized processor network D updates its processors via the dynamic equation

$$x^+ \;=\; f(x, u)\,,$$

where x is the current state of the network (a vector), u is an external input (also possibly a vector), and f is a composition of functions:

$$f = \psi \circ \pi\,,$$

where

$$\pi :\;\; \mathbb{R}^{N+M} \mapsto \mathbb{R}^N$$

is some vector polynomial in $N + M$ variables with real coefficients, and

$$\psi \;:\; \mathbb{R}^N \mapsto \mathbb{R}^N$$

is any vector function which has a bounded range and is locally Lipschitz. (Thus, the composite function $f = \psi \circ \pi$ again satisfies the same properties.)

We also assume, as part of the definition of generalized processor network, that, at least for binary inputs of the type considered in the definition of "formal networks," given in Chapter 3, D outputs "soft" binary information. That is, there exist two constants α, β, satisfying $\alpha < \beta$ and

called the *decision thresholds*, so that each output neuron of D outputs a stream of numbers each of which is either smaller than α or larger than β. We interpret the outputs of each output neuron y as a binary value:

$$\text{binary}(y) = \begin{cases} 0 & \text{if } y \leq \alpha \\ 1 & \text{if } y \geq \beta \,. \end{cases}$$

In the usual model we studied earlier, the values are always binary, but we allow more generality to show that even if one allows more general analog values, no increase in computational power is attained, at least up to polynomial time.

Remark 8.1.1 The above assumptions imply that, for each $\rho > 0$, there exists a constant C, such that, for all x and $\tilde{x}$ satisfying that

$$|x - \tilde{x}| < \rho \quad \text{and} \quad x \in \text{Range}\,(\psi)$$

(the absolute value sign indicates Euclidean norm), the following property holds:

$$|\psi(x, u) - \psi(\tilde{x}, u)| \leq C|x - \tilde{x}|$$

for any binary vector u. A similar property holds for f. □

Let $T : \mathbb{N} \mapsto \mathbb{N}$ be a function from integers into integers. We say that a generalized processor network D *computes in time* T if for every input of size $n \in \mathbb{N}$, D completes its output in no more than $T(n)$ steps.

A neural network is a special case of a generalized processor network, in which all coordinates of the function ψ compute the same piecewise linear function σ, and the polynomial π is a first order polynomial, that is, an affine function.

8.2 Generalized Networks with Bounded Precision

Let D be a generalized processor network

$$D \,:\, x^+ \,=\, \psi(\pi(x, u))$$

as above. Let Q be a positive integer. The Q-*truncation* of D, denoted

$$Q\text{-Truncation } (D) \ ,$$

is the network with dynamics defined by

$$x^+ \ = \ Q\text{-Truncation } [\psi(\pi(x,u))] \ ,$$

where "Q-Truncation" represents the operation of truncating after Q bits. The Q-*chop* of D is the network with dynamics defined by

$$x^+ \ = \ Q\text{-Chop } [\psi(\pi(x,u))] \ \equiv \ Q\text{-Truncation } [\psi(\tilde{\pi}_Q(x,u))] \ ,$$

where $\tilde{\pi}_Q$ is the same polynomial π but with coefficients truncated after Q bits.

The next observations insure that round-off errors due to truncation or chopping are not too large.

Lemma 8.2.1 Assume D computes in time T, with decision thresholds α, β. Then, there is a constant c such that the function

$$q(n) = cT(n)$$

satisfies the following property. For each positive integer n, let $Q = q(n)$. Then, Q-Truncation(D) computes the same function as D on inputs of length at most n, with decision thresholds

$$\alpha' = \alpha + \frac{\beta - \alpha}{3} \ \text{ and } \ \beta' = \beta - \frac{\beta - \alpha}{3} \ .$$

Proof. Let D be a generalized processor network satisfying the above conditions, and let $\tilde{D} = Q$-truncation(D), with Q still to be decided upon. Let δ be the error due to truncating after Q bits, that is, $\delta = c_1 2^{-Q}$ for some constant c_1. Finally, let ϵ_t be the largest accumulated error in all the processors by time t. The following estimates are obtained using the Lipschitz property of f:

$$\begin{aligned} \epsilon_o &= 0 \\ \epsilon_1 &= \delta \\ \epsilon_t &= \delta \sum_{i=o}^{t-1} C^i \ = \delta \frac{C^t - 1}{C - 1} \ , \end{aligned}$$

where C is the Lipschitz constant of f for $\rho = 1$ (c.f. Remark 8.1.1). To bound the error with the amount $\gamma = \frac{\beta-\alpha}{3}$, we require

$$\epsilon_t \leq \gamma \ .$$

That is,

$$\delta \leq \frac{\gamma(C-1)}{C^t - 1} \leq \tilde{C}^{-t} \ ,$$

for some constant $\tilde{C}$. This requirement is met when δ is the truncation error corresponding to

$$Q = (\log_2((\frac{1}{c_1}\tilde{C}))T \ ,$$

so we can take $q(n) = \log_2(\frac{1}{c_1}\tilde{C})T(n)$. ∎

As a corollary of Lemma 6.3.1, and using an argument exactly as in the proof of Lemma 8.2.1, we conclude:

Lemma 8.2.2 Lemma 8.2.1 holds for the Q-chop network as well.

8.3 Equivalence of Neural and Generalized Networks

Definition 8.3.1 Given a vector function $f = \psi \circ \pi$ as above, we say that f is *approximable in time* $A_f(n)$, if there is a Turing Machine M that computes $T(n)$-Truncation(f) in time $A_f(n)$ on each input having total bit size n.

Example 8.3.2 If $f = \psi \circ \pi$, ψ is approximable, and π has rational coefficients, then f is approximable. (As π is approximable at this case.) □

Lemma 8.3.3 Let $L(T)$ be the class of languages recognized by generalized processor networks in time T, for which the function f is approximable in time A_f, and the function T is computable in time $M(n)$. Then, $L(T)$ is included in the class of languages recognized by Turing Machines in time $O(M(n) + T(n)A_f(T(n)))$.

Proof. Assume given a generalized processor network D satisfying the above assumptions. A Turing Machine which approximates it can be built as follows. The machine receives an input string of length n. As a first step, it computes the function $T(n)$, and it estimates the required precision $Q = q(n)$ as in the previous Lemma. Finally, it simulates the generalized processor network step by step, forgetting all information but the first Q required bits. This Turing Machine computes the required function in the stated time. ∎

Corollary 8.3.4 Let D be a network which computes in polynomial time T, and so that f is approximable is polynomial time. Then the language recognized by D is in P.

Definition 8.3.5 Given a vector function $f = \psi \circ \pi$ as above, we say that f is *non-uniformly $F(n)$-approximable in time $A_f(n)$*, if there is a Turing Machine M that computes $T(n)$-Chop(f) using an advice function (c.f. [BDG90], volume I, pg 99-115) in K[$F(n)$, poly($T(n)$)].

Example 8.3.6 Assume a generalized processor network D that computes in time T. A polynomial π with general real coefficients is non-uniformly $T(n)$-computable: For each input of size n, the machine receives the first $O(T(n))$ bits of each coefficient as an advice sequence, and then computes the polynomial. □

From the above results, we may conclude as follows:

Theorem 16 *Let D be a generalized processor network which computes via a function $f = \psi \circ \pi$. Assume ψ is non-uniformly $F(n)$-approximable in polynomial time. Then there exists a neural network $\mathcal{N}_D$ which recognizes the same language as D and which does so with at most a polynomial time slowdown. Furthermore, if ψ is $F(n)$-approximable in polynomial time and π involves rational coefficients only, the weights of $\mathcal{N}_D$ are rational numbers as well.*

Corollary 8.3.7 Adding flexibility to the neural network model does not add power to the model, except for a possible polynomial time speed up. This flexibility includes:

- Using a high order polynomial π rather than an affine function.
- Using other ψ functions rather than the saturation we used earlier, including the possibility of having different functions in different neurons.
- Allowing for the output to be "soft binary" rather than pure binary.

Note that networks with high order polynomials have appeared especially in the language recognition literature (see e.g. [GMC+92] and references there). We emphasize the relationship between these models: Let N_1 be neural network (of any order), which recognizes a language L in polynomial time. Then there is a first order network N_2 which recognizes the same language L in polynomial time.

Remark 8.3.8 The networks that we consider are mildly "robust to noise and to implementation error" in the sense that small enough perturbations in weights or (formulated in a suitable sense) in the sigmoid activation function do not affect the computation, as long as "soft binary" outputs are considered. Given any time T, there is some ϵ_T so that an error of ϵ_T would not affect the computation up to a time T. This is an easy consequence of the continuous dependence of the output on all the data. (A detailed proof involves defining precisely "perturbations of the activation function; we omit the details.) □

Chapter 9

Space Constraints

This chapter discusses rational-weight neural nets on which a bound is set on the precision available for the computations. It should be observed that any simulation of a neural net computation, e.g. by implementing a simulation program on a more or less standard computer, will be subject to such bounds. Indeed, efficient implementations of the arithmetic require dedicated hardware, able to handle "reals" of a limited precision seldom larger than 64 bits (and quite frequently smaller). When larger precision is necessary, for instance to process longer inputs, one must resort to a software implementation of real arithmetic (sometimes provided by the compiler), and even in this case a physical limitation on the length of the mantissa of each state of a network under simulation is imposed by the amount of available memory. It is thus important to know the computational consequences of these limitations.

This very same observation suggests that some connection can be established between the space requirements needed to solve a problem and the precision required on the states of the neural networks that solve them. In Section 9.1 we consider rational networks consisting of neurons whose precision increases as a function of the input length. In Section 9.2 we consider neurons of constant precision and allow for a growing network. *(This is the only place in this thesis where we consider growing, not constant-size, networks.)*

9.1 Space Classes

Definition 9.1.1 A rational neural net *works within precision* $S(n)$ if and only if all the weights, and all the rational values of the states of the neurons through a computation on an input of length n, can be represented in binary within $O(S(n))$ bits.

We observe here the following:

Theorem 17 *Let $S(n) \geq n$ be a space-constructible function. Then the following are equivalent:*

1. *the set L is accepted by a Turing machine within space $O(S(n))$;*

2. *the set L is accepted by a neural net within precision $O(S(n))$.*

The proof is omitted, as it is similar to the proof of Theorem 4 in Chapter 5, We should point out, however, that the proof relies on a preliminary phase during which the input is completely loaded into the state of a specific neuron, before proceeding to the actual computation. This is the reason why we need the condition $S(n) \geq n$, since the precision needed for that neuron will be at least linear. Actually, the proof of Theorem 18 below can be used as well to prove this theorem, taking into account that the restrictions imposed there become trivial for at least linear space.

It is quite interesting as well to see what happens under *sublinear* precision bounds. The point is that the input convention we have described for neural nets makes available each input symbol only once; moreover, it is available for only a single step, since the next iteration brings a new symbol in.

Thus, under sublinear bounds, we may expect some sort of equivalence to restricted variants of Turing machines, such as on-line machines or a still more restricted model called here lr-machines. An *on-line machine* is a Turing machine whose input head can either move to the right, or stay still, but cannot backtrack to the left. We define *lr-machines* as the subclass of on-line machines that must move their input head to the right one symbol per each step, and cannot backtrack nor even stay at a symbol for more than one step. However, they are allowed to continue working without further reading after exhausting the input. This last period of work uses only the information gathered in the worktapes during the reading. Clearly this restricted model is equivalent to the standard model for at least linear space bounds.

For sublinear $S(n)$, the next result provides one necessary and one sufficient characterization. It is a simple extension of our previous results, and we only provide an outline of its proof.

Theorem 18 *Let $S(n)$ be any space-constructible function.*

1. *If a set L is accepted by an lr-machine within space $O(S(n))$, then L is accepted by a neural net within precision $O(S(n))$.*

2. *If a set L is accepted by a neural net within precision $O(S(n))$, then L is accepted by an on-line machine within space $O(S(n))$.*

Note that, unlike the previous and next theorems, we don't have to impose any lower bound on $S(n)$ here. The first part essentially corresponds to proving that the intermediate step of loading the input into a single neuron state, as done in Chapter 5, is not necessary; we may use instead the real time simulation from the same chapter.

The second part consists of a straightforward simulation of the computation of the neural net. The state of each of the fixed number of neurons is kept in a worktape , where it fits due to the precision bound. Since the network receives its input in real time, there is never the need of backtracking the input head during the simulation. Observe however that the simulating machine is not an lr-machine since each step of the net requires a nontrivial number of Turing machine steps due to the arithmetic operations to be performed.

Off-line space-bounded machines can be proven equivalent to precision-bounded neural nets under a different input convention.

Definition 9.1.2 A neural net *with cyclic input* receives the input w through two input lines as follows: the data line brings in the bits of the input w repeatedly, w^∞, while the validation line brings in $(10^{|w|-1})^\infty$. □

So, the data line brings in $wwwww\ldots$ and the validation one, instead of marking the end of the whole input, marks the beginning of each cycle. This (admittedly somewhat artificial) input convention gives:

Theorem 19 *Let $S(n) \geq \log n$ be a space constructible function. Then the following are equivalent:*

1. *the set L is accepted by an off-line Turing machine within space $O(S(n))$;*

2. the set L is accepted by a neural net with cyclic input within precision $O(S(n))$.

Here we only sketch the proof.

Proof. $1 \Rightarrow 2$) The network $\mathcal{N}$ simulating the Turing machine M is built conceptually out of two subnetworks: In a manner similar to that of Chapter 5 we construct a constant size subnet that receives as input the bit currently scanned by the input-tape head of M and the state of M, and returns a new state and the direction to move the input-tape head, right or left. Another neuron keeps a rational that, interpreted as an integer value, indicates the current position of the input-tape head. The value is incremented or decremented depending on the direction of movement. Then another subnet, triggered by the 1 that marks the beginning of each cycle, counts up to the position of the input-tape head to detect the input symbol necessary for the simulation of the next step. With some precomputation time, it is possible to do the counting in real time using only logarithmic precision.

$2 \Rightarrow 1$) For the backward implication, use the same simulation as for the on-line case. When reaching the right end of the input, stop the simulation, reset the input tape head, and resume it; when the simulating machine is reading the first symbol of the input, it simulates a 1 on the input validation line. ∎

The fact that time-bounded rational nets correspond modulo polynomial-time simulations to time-bounded Turing machines (Chapter 5), taken together with Theorem 17 here, allows us to close this section by pointing out a remark on the "linear precision suffices" lemma of Chapter 6. There it is proved that for a neural net running in time $T(n)$, the net obtained truncating all states to $O(T(n))$ bits is equivalent to it. That proof is valid for real weights; but if we consider its restriction to the simpler rational case, then we can see an interesting intuitive analogy. Through the equivalences with the Turing model, we see that this result corresponds in some sense to the basic theorems relating time-bounded and space-bounded classes, and in particular to the by now elementary result that everything done in time $T(n)$ is done in space $T(n)$ as well. The "linear precision lemma", restricted to the rational case, would be essentially the neural net analog of this result.

9.2 Fixed Precision

A rational network *works within a constant precision* if it works within a precision $S(n)$ and $S(n)$ is a constant.

Clearly, a finite net that works within a constant precision is essentially a finite automaton. To gain a recursive power with neurons of constant precision, we have to allow the network to grow as a function of the computation time.

Ideally, the number of neurons required to simulate a Turing machine that operates in space S is $\Omega(S)$. We prove that indeed this lower bound is achievable. Furthermore, we sketch the construction of a network which, using this number of neurons, simulates a Turing machine in linear time.

Theorem 20 *Let M be a Turing machine that computes the function ϕ in space S and time T. Then, for each positive integer p, there exists a network $\mathcal{N}^p[\frac{cS}{p}]$ (c is a constant independent of p) that computes ϕ in time $O(T)$, consists of $\frac{cs}{p}$ neurons, and works within a constant precision p.*

Proof. Assume that M is a z-stack machine. We show how to simulate a stack of M. This, combined with the proof of Theorem 4 in Chapter 5, proves the required conclusion.

We start with a few definitions: A p-block is a sequence of length p over the alphabet $\{0,1,3\}$. The interpretation of a block $b = b_1 b_2 \cdots b_p$ in base 4 ($b|_4$) is the value $\sum_{i=1}^{p} \frac{b_i}{4^i}$.

Given a sequence $\alpha = a_1 \ldots a_l \in \{1,3\}^*$, the *$p$-block family* of the sequence α, $\mathcal{F}[\alpha]$, is the infinite sequence of blocks $\{\beta_i\}_{i=1}^{\infty}$ defined as follows:

$$\begin{aligned}
\beta_i &= a_{(i-1)p+1} \cdots a_{ip} && i = 1, 2, \ldots \lfloor \tfrac{l}{p} \rfloor \\
\beta_i &= a_{(i-1)p+1} \cdots a_l \underbrace{0 \cdots 0}_{p-l+p\lfloor \frac{l}{p} \rfloor} && i = \lceil \tfrac{l}{p} \rceil \\
\beta_i &= \underbrace{0 \cdots 0}_{p} && i > \lceil \tfrac{l}{p} \rceil \quad .
\end{aligned}$$

Similarly, we define the *p-mirror-block family* $\mathcal{F}^m$ of the sequence $\alpha = a_1 \ldots a_l \in \{1,3\}^*$ to be

$$\mathcal{F}^m[\alpha] = \{\hat{\beta}_i \mid \hat{\beta}_i = \text{ reverse } [\beta_i],\ \beta_i \in \mathcal{F}\}_{i=1}^{\infty}$$

where the reverse operation is defined intuitively as: reverse$[a_1 a_2 \cdots a_p] = a_p \cdots a_2 a_1$.

Given a binary stack s, we identify it with a $\{1,3\}$-sequence α_s in the top to bottom order (as was described in Chapter 2). Let $\mathcal{F}[\alpha_s]$ and $\mathcal{F}^m[\alpha_s]$ be the p-block family and p mirror-block family of s, respectively. We associate two sequences of neurons with s:

$$\begin{aligned} Q_i &= \{q_i \mid q_i = \beta_i|_4,\ \beta_i \in \mathcal{F}[\alpha_s]\} \\ \hat{Q}_i &= \{\hat{q}_i \mid q_i = \hat{\beta}_i|_4,\ \beta_i \in \mathcal{F}^m[\alpha_s]\}\ . \end{aligned}$$

Note that only $2\lceil\frac{|s|}{p}\rceil$ neurons of $Q_i \bigcup \hat{Q}_i$ are not identically 0.

Each neuron q_i and $\hat{q}_i$ has three associated readout neurons:

$$\begin{aligned} T[q] &= \begin{cases} 0 & q \leq \frac{1}{2} \\ 1 & q > \frac{1}{2} \end{cases} \\ E[q] &= \begin{cases} 0 & q = 0 \\ 1 & q \neq 0 \end{cases} \\ H[E[q],T[q]] &= \begin{cases} 0 & E[q] = 0 \\ \frac{1}{4} & E[q] = 1, T[q] = 0 \\ \frac{3}{4} & T[q] = 1\ . \end{cases} \end{aligned}$$

The top element of s is given by $T[q_1]$. The stack s is empty when $E[q_1] = 0$. We show how using the blocks and mirror blocks, we simulate the stack operations push and pop in a constant number of steps each:

- **Push(I)**, $I \in \{\frac{1}{4}, \frac{3}{4}\}$:

$$\begin{aligned} q_1 &= \frac{1}{4}(q_1 - \frac{H[\hat{q}_1]}{4^{p-1}}) + I \\ q_i &= \frac{1}{4}(q_i - \frac{H[\hat{q}_i]}{4^{p-1}}) + H[\hat{q}_{i-1}] \quad i > 1 \\ \hat{q}_1 &= 4(\hat{q}_1 - H[\hat{q}_1] + \frac{I}{4^p}) \\ \hat{q}_i &= 4(\hat{q}_i - H[\hat{q}_i] + \frac{H[\hat{q}_{i-1}]}{4^p})\ . \end{aligned}$$

- **Pop**:

$$\begin{aligned} q_i &= 4(q_i - H[q_i] + \frac{H[q_{i+1}]}{4^p}) \\ \hat{q}_i &= \frac{1}{4}(\hat{q}_i - \frac{H[q_i]}{4^{p-1}}) + H[q_{i+1}]\ . \end{aligned}$$

That is, keeping the value of a stack both in a chain of p-blocks and in a chain of their mirror images, the stack operations are simulated in a constant number of steps each. This proves the theorem. ∎

Chapter 10

Networks With General σ and Finite Automata

It has been known at least since the work of McCulloch and Pitts ([MP43]) in 1943 that networks consisting of threshold neurons can simulate finite automata. In Chapter 4 we proved a similar result for our model of networks (i.e., those employing saturated neurons) even when only *integer* weights are allowed. Here we suggest that almost every "reasonable" kind of neuron will result in a finite automata simulation. Note that in this chapter we do not use exactly the model of networks developed in Chapters 1 and 3, in the sense that both activation functions and input-output conventions become more flexible.

Let ϱ be any function that satisfies the following property:

$$\star \text{ Both } \lim_{x\to\infty} = t_+ \text{ and } \lim_{x\to-\infty} = t_- \text{ exist and } t_+ \neq t_- .$$

We show that any network of ϱ-neurons of the type

$$x_i(t+1) = \varrho\left(\sum_{j=1}^{N} a_{ij}x_j(t) + \sum_{j=1}^{M} b_{ij}u_j(t) + c_i\right) \quad , \quad i = 1, \ldots, N \tag{10.1}$$

can simulate a finite automaton.

10.1 Simulation

Here, we use the general definition of finite automaton with no initial state and with sequential output. As a mathematical object, an *automaton* is a quintuple

$$M \;=\; (S, U, Y, f, h)$$

consisting of sets S, U, and Y (called respectively the state, input, and output spaces), as well as two functions

$$f \;:\; S \times U \to S\,, \quad h \;:\; S \to Y$$

(called the next-state and the output maps, respectively). A *finite automaton* is one for which each of the sets S, U, and Y is finite.

We start by introducing the notion of simulation. In general, given an automaton $M = (S, U, Y, f, h)$, the map f can be extended by induction to arbitrary input sequences. That is, for any sequence $u_1, \ldots, u_k$ of values in U,

$$f_*(s, u_1, \ldots, u_k)$$

is defined as the iterated composition $f(f(\ldots f(f(s, u_1), u_2), \ldots, u_{k-1}), u_k)$. Suppose now given two automata $M = (S, U, Y, f, h)$ and $\overline{M} = (\overline{S}, U, Y, \overline{f}, \overline{h})$ which have the same input and output sets. The automaton $\overline{M}$ *simulates* M if there exist two maps

$$\text{ENC} : S \to \overline{S} \quad \text{and} \quad \text{DEC} : \overline{S} \to S\,,$$

called the *encoding* and *decoding* maps respectively, such that, for each $s \in S$ and each sequence $\omega = u_1, \ldots, u_k$ of elements of U,

$$f_*(s, \omega) = \text{DEC}\,[\overline{f}_*(\text{ENC}\,[s], \omega)]\,, \quad h(s) = \overline{h}(\text{ENC}\,[s])\,.$$

Assume that for some integer m the input value set U consists of the vectors $e_1, \ldots, e_m$ in $\mathbb{R}^m$, where e_i is the ith canonical basis vector, that is, the vector having a 1 in the ith position and zero in all other entries. Similarly, suppose that Y consists of the vectors $e_1, \ldots, e_p$ in $\mathbb{R}^p$. (The assumption that U and Y are of this special "unary" form is not very restrictive, as one may always encode inputs and outputs in this fashion.) Note that networks having such inputs and outputs are not of the "formal" type described in Chapter 3. The input validation information in such networks is the sum of the different input lines (this sum has the value 1 as long as some input is present). Output validation information is computed analogously.

10.2 Main Result

The following interpolation fact holds for any function ϱ which satisfies property $\star$:

Lemma 10.2.1 Let ϱ be a function with the property $\star$. Then, there exist constants $\omega_i, b_i, c_i \in \mathbb{R}$, $i = 1,2,3$ so that the function

$$f(y) = \omega_0 + \sum_{i=1}^{3} \omega_i \varrho(b_i y + c_i)$$

satisfies $f(-1) = f(0) = 0$ and $f(1) = 1$.

Before proving the above lemma, we show how the theorem follows from it:

Theorem 21 *Every finite automaton can be simulated by a neural net with activation function ϱ.*

Proof. (*Of The Theorem*)
Assume that the states of the finite automaton M to be simulated are $\{\xi_1, \ldots, \xi_s\}$. A neural network that simulates M has $N = 3sm$ neurons and is built as follows. Denote the coordinates of the state vector $x \in \mathbb{R}^N$ by x_{ijk}, $i = 1, \ldots s$, $j = 1, \ldots, m$, $k = 1,2,3$.

Consider the Boolean variables g_{rv} for $r = 1, \ldots, s$, $v = 1, \ldots, m$, that indicate if the current state of M is r and the last input read was v. We will express these variables as

$$g_{rv} \equiv \omega_{rv} + \sum_{k=1}^{3} \omega_{rvk}\, x_{rvk} \ , \tag{10.2}$$

where the weights $\omega_{rv}, \omega_{rvk}$, will be found later. We write (g_{rv}) for a matrix indexed by r and v. In terms of these quantities, the update equations for $r = 1, \ldots, s$, $v = 1, \ldots, m$, $k = 1,2,3$, can be expressed as follows:

$$x^+_{rvk} \ = \ \varrho\left(b_{rvk}(\sum_{j=1}^{m} \sum_{i \in S_{rv}} g_{ij} + u_v - 1) + c_{rvk} \right) \ , \tag{10.3}$$

where

$$S_{rv} \ := \ \{l \,|\, f(q_l, e_v) = q_r\}$$

and where b_{rvk} as well as c_{rvk} will also be specified below. Finally, for each $l = 1, \ldots, p$, the lth coordinate of the output is defined as

$$y_l \ = \ \varrho\left(c \sum_{j=1}^{m} \sum_{i \in T_l} g_{ij} \right)$$

where $T_l := \{i | h_l(q_i) = 1\}$ for the coordinate h_l of h, and c is any constant so that $\varrho(0) \neq \varrho(c)$.

The proof that this is indeed a simulation is as follows:

1. We first prove inductively on the steps of the algorithm that the expressions (g_{rv}) are always of the type E_{ij}, where E_{ij} denotes the binary matrix that its ijth entry has the value 1 and all the rest have the value 0. Furthermore, except for the starting time, E_{rv} indicates that the simulated finite automata is in state r and it last read input was v.

   - We start the network with an initial state $x_0 \in \mathbb{R}^N$ so that the starting (g_{ij}) has the form E_{r1}, where q_r is the corresponding state of the original automaton. It is easy to verify that such x's are possible, since in Equation 10.2 the different equations are uncoupled for different r and v, and not all weights ω_{rvk} for $k = 1, 2, 3$ can vanish, otherwise in lemma 10.2, the function f (which is used as g_{rv}) is constant.
   - Assume (g_{ij}) has indeed the Boolean values as stated at some time t, that is, only one of the g_{rv} has the value 1 and the rest are in 0, and the associated state of the finite automata is q_r and the last read input is u_v. Then, the expression

     $$y_{rv} = (\sum_{j=1}^{m} \sum_{i \in S_{rv}} g_{ij} + u_v - 1)$$

     can only take the values -1, 0, or 1. The value 1 can only be achieved for this sum if both $u_v = 1$ and there is some $i \in S_{rv}$ so that $g_{ij} = 1$, that is, if the current state of the original machine is q_i and $f(q_i, e_v) = q_r$.

     By Lemma 10.2.1, there exist values $\omega_{rvk}, b_{rvk}, c_{rvk}$ $(k = 1 \ldots 3)$ so that $f(y_{rv})$ (which is by definition here the value of g_{rv} at time $t + 1$) assumes the value 1 only if y_{rv} was 1, and is 0 for the other two cases. This proves the correctness in terms of the expressions g_{ij}. The vectors x_{rvk} take the values $\varrho(b_{rvk} y_{rv} + c_{rvk})$, as described by Equation 10.3.

2. The encoding and decoding functions are defined as follows: The encoding map ENC $[q_r]$ maps q_r into any fixed vector x so that Equation 10.2 gives $(g_{ij}) = E_{r1}$. The decoding map DEC $[x]$ maps those vectors x that result in $(g_{ij}) = E_{rv}$ $(r = 1 \ldots s, \ \ v = 1 \ldots m)$ into q_r, and is arbitrary on all other elements of $\mathbb{R}^N$.

∎

In the next section, we prove the lemma.

10.3 Proof of the Above Lemma

Proof. We prove more than required, namely, we show that for each choice of three numbers $r_{-1}, r_0, r_1 \in \mathbb{R}$ there exist $\omega_0, \omega_i, b_i, c_i \in \mathbb{R},\ i = 1,2,3$ so that, denoting

$$f(y) = \omega_0 + \sum_{i=1}^{3} \omega_i \varrho(b_i y + c_i) \ ,$$

it holds that $f(-1) = r_{-1}$, $f(0) = r_0$, and $f(1) = r_1$. To prove this, it suffices to show that there are b_i, c_i $(i = 1,2,3)$, so that the matrix

$$\Sigma_{bc} = \begin{pmatrix} \varrho[b_1(-1)+c_1] & \varrho[b_2(-1)+c_2] & \varrho[b_3(-1)+c_3] \\ \varrho[b_1(0)+c_1] & \varrho[b_2(0)+c_2] & \varrho[b_3(0)+c_3] \\ \varrho[b_1(1)+c_1] & \varrho[b_2(1)+c_2] & \varrho[b_3(1)+c_3] \end{pmatrix}$$

is non-singular. Hence, for all $R = \text{COL}\,(r_{-1}, r_0, r_1)$ there exists a vector $A = \text{COL}\,(\omega_1, \omega_2, \omega_3)$ so that

$$\Sigma A = R \ ,$$

and this solves our problem with $\omega_0 = 0$. We will prove this latter property for a certain function $\tilde{\varrho}$ of the form $a\varrho(x) + b$, and this will imply the result for ϱ.

Let m_i, $i = 1,2,3$ be maps on the real numbers so that $(m_i(-1), m_i(0), m_i(1)) = e_i$ ($e_i \in \mathbb{R}^3$ is the ith canonical vector). Let $U = \{-1, 0, 1\}$. We say that k ϱ-neurons g_j *linearly interpolate[U] the map* m_i if there exist constants $\omega_1^i, \ldots, \omega_k^i$ that $f_i(u) = \sum_{j=1}^{k} \omega_j^i g_j(u)$ and

$$f_i(u) = m_i(u)$$

for all $u \in U$. These neurons are said to ϵ-*approximate[U]* m_i if

$$|f_i(u) - m_i(u)| < \epsilon$$

for all $u \in U$.

Proposition 10.3.1 There are three $\mathcal{H}$-neurons (i.e., threshold neurons, also called before Heaviside neurons) that interpolate$[U]$ the maps m_i, $i = 1,2,3$.

Proof. Let

$$\begin{aligned} h_1(x) &= \mathcal{H}(x+\frac{1}{2}) \\ h_2(x) &= \mathcal{H}(x-\frac{1}{2}) \\ h_3(x) &= \mathcal{H}(-x-\frac{1}{2})\,. \end{aligned}$$

The interpolation is by: $m_1 = h_2$ $(w_1^1 = 0, w_2^1 = 1, w_3^1 = 0)$, $m_2 = h_1 - h_2$ $(w_1^2 = 1, w_2^2 = -1, w_3^2 = 0)$, and $m_3 = h_3$ $(w_1^3 = 0, w_2^3 = 0, w_3^3 = 1)$. ∎

Proposition 10.3.2 For all $\epsilon > 0$ there are three ϱ-neurons that ϵ-interpolate$[U]$ the maps m_i, $i = 1, 2, 3$.

Proof. First note that we can impose $t_+ = 1$, and $t_- = 0$ on ϱ without restricting the affine span of the neurons. This is possible by defining the function

$$\tilde{\varrho} = \frac{\varrho(x) - t_-}{t^+ - t_-}\,.$$

Without loss of generality, we assume $t^+ = 1$ and $t_- = 0$ from now on. So, for each $\epsilon > 0$ there is some $\eta > 0$ such that for all y, $|y - \frac{1}{2}| > \eta$

$$|\varrho(y-\frac{1}{2}) - \mathcal{H}(y-\frac{1}{2})| < \epsilon\,.$$

In particular $\forall y, |y - \frac{1}{2}| > \frac{1}{4}$ we can choose $\lambda > 4\eta$, and using that $\mathcal{H}(\lambda(y - \frac{1}{2})) = \mathcal{H}(y - \frac{1}{2})$,

$$|\varrho(\lambda(y-\frac{1}{2})) - \mathcal{H}(y-\frac{1}{2})| < \epsilon\,.$$

A similar argument can be applied to $(y + \frac{1}{2})$ and $(-y - \frac{1}{2})$. We conclude that for all $\epsilon > 0$ there exists some λ so that

$$\begin{aligned} g_1(x) &= \varrho(\lambda(x+\frac{1}{2})) \\ g_2(x) &= \varrho(\lambda(x-\frac{1}{2})) \\ g_3(x) &= \varrho(\lambda(-x-\frac{1}{2})) \end{aligned}$$

satisfy $|g_i(x) - h_i(x)| < \frac{\epsilon}{3}$ for all $u \in U$. The result is now clear. ∎

Now, define the three matrices:

$$\begin{aligned} \Sigma &= [g_j(u_i)]_{ij} \\ M &= [m_j(u_i)]_{ij} \\ W &= [\omega_j(u_i)]_{ij} \end{aligned}$$

From Proposition 10.3.2 we conclude that for all $\epsilon > 0$ there are g_1, g_2, g_3 so that

$$\left| \begin{pmatrix} \varrho[b_1(-1)+c_1] & \varrho[b_2(-1)+c_2] & \varrho[b_3(-1)+c_3] \\ \varrho[b_1(0)+c_1] & \varrho[b_2(0)+c_2] & \varrho[b_3(0)+c_3] \\ \varrho[b_1(1)+c_1] & \varrho[b_2(1)+c_2] & \varrho[b_3(1)+c_3] \end{pmatrix} \begin{pmatrix} 0 & 1 & 0 \\ 0 & -1 & 1 \\ 1 & 0 & 0 \end{pmatrix} - \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \right| < \epsilon$$

Choosing ϵ so that Σ is non-singular, the lemma results. ∎

Chapter 11

Parallel Time Classes

Parallel models have in principle more power than sequential *ones. Many Parallel models exist, and not all of them are equivalent. Our parallel models are taken from the so-called Second Machine Class [VEB90]. This class captures a very frequently observed features of parallelism, characterized by the Parallel Computation Thesis: time on these models corresponds, modulo polynomial overheads, to space on First Class models. Prominent members of the Second Machine Class are the alternating Turing machines and the Vector Machines ([PS76], see also [BDG90]).

Neural nets are considered a very appropriate model of parallel computation, due to the fact that the net result embodies the activity of a large number of neurons (hence the name "Parallel Distributed Processing"). It was proven in Chapter 8 that second-order nets can be simulated with a polynomial overhead in time by first-order nets. That is, allowing neurons that compute *polynomials* does not increase the computational power of nets (up to polynomial time). In this chapter we show that second class power is obtained by nets with rational weights if they can use *rational* functions (i.e. division) and *bitwise AND*, and obey an exponential precision bound.

We also consider the case of nets with real weights: the use of division and bitwise AND in this case provides exactly the power of nonuniform parallel computation, so that time corresponds to nonuniform (bounded fan-in) circuit depth; in particular, any arbitrary set can be decided in linear time by nets with real weights, provided that division and bitwise AND are available. (This corresponds to writing arbitrary boolean functions as sum of minterms in linear depth.) So, essentially real weights add the characteristic of nonuniformity to both the sequential (compare with Chapters 5, 6) and the parallel models (here).

---

*The sequential models can be defined in a completely standard way by time-bounded or space-bounded multitape Turing machines, possibly nondeterministic, e.g. classes like P, PSPACE, or NP. Relativizations of these classes are also used; the oracle machine model used for defining them is standard. All these classes are invariant under changes of the machine model, provided that it stays within the so-called First Machine Class [VEB90]: they simulate and are simulated by multitape Turing machines within a polynomial time overhead and a linear space overhead.

The extended nets we consider in this chapter have processors with either an update equation of the form

$$x_i(t+1) = \sigma\left(\frac{P_i(x_1(t),\ldots,x_N(t))}{Q_i(x_1(t),\ldots,x_N(t))}\right) ,$$

where P_i and Q_i are polynomials, or of the form

$$x_i(t+1) = x_{j_1}(t) \wedge \ldots \wedge x_{j_k}(t) ,$$

where $\wedge$ the denotes the bitwise AND of binary representations (note that adding σ does not make any difference in this case). When computing an update involving rational functions, it is assumed that the denominator does not vanish (otherwise the computation is undefined).

The rest of this chapter is organized as follows: In section 11.1, we prove that, with the addition of these operations, nets with rational weights are equivalent to parallel models. In section 11.2 we sketch the proof that the same networks with real weights are also equivalent to the same parallel models, but in their non-uniform versions.

11.1 Extended Nets with Rational Weights

We say that a net *works within precision* $p(n)$ if the binary expansion of all weights, and of any state appearing during the computation on an input of length n, is identically zero after the first $p(n)$ digits. Let NNTIME(t,p) be the set of languages accepted by nets with rational weights, division and bitwise AND in time $O(t(n))$ and precision $O(p(n))$ simultaneously.

Intuitively, the extra power we get by using division can be demonstrated by the following example. By repeated multiplication a net can build in time $O(t)$ rationals as small as 2^{-2^t}. To recover the first bit of these numbers, a net without division can only multiply at each step by some (constant) weight, and thus needs $2^{\Omega(t)}$ steps. However, a single division can turn this digit into the most significant one.

We use this power of division, and bitwise AND, to simulate a model of unbounded parallelism introduced by Pratt and Stockmeyer, that of vector machines ([PS76], see also [BDG90, KR90]).

Vector machines are machines that can implement boolean operations and left and right shifts on their potentially infinite registers; these capabilities give them the power of parallel machines. More precisely, a *vector machine* is a processor together with a fixed number of vector registers V_1,

$V_2, \ldots, V_r$, each containing bit vectors. These bit vectors are ultimately constant sequences of bits written from right to left, and infinite to the left. The length of a vector register is the length of its nonconstant part. Vectors that are ultimately 0 and ultimately 1 represent non-negative and negative integers respectively. The input is given to the machine in register V_1, and the output is in V_1 when the machine halts. Programs for the vector machines can contain the following instructions, assumed to have unit cost:

- $V_i := x$: Load the constant x into V_i.
- $V_i :=$ **not** V_i: Bitwise negate all of V_i.
- $V_i := V_j \wedge V_k$: Bitwise AND V_j and V_k.
- $V_i := V_i \uparrow V_j$: If V_j contains a positive number, shift V_i to the left by V_j positions; new positions are filled with zeros. Otherwise, do nothing.
- $V_i := V_i \downarrow V_j$: If V_j contains a positive number, shift V_i to the right by V_j positions; rightmost bits are discarded. Otherwise do nothing.
- if $V_i = \mathbf{0}$ go to *label.*
- *accept, reject.*

To make vector machines equivalent in power to other Second Class models, we have to impose the following restriction: no register is ever shifted by more than $2^{O(t(n))}$ positions in a single shift instruction, where $t(n)$ is the machine's running time (see [BDG90]). In other words, the arguments V_j in the shift instructions always consist of no more than $O(t(n))$ bits. We call machines with this property *restricted.* Let VECTOR – TIME(t) be the class of languages accepted by restricted vector machines in time $O(t(n))$.

We now show that, up to polynomial time, the classes VECTOR – TIME(t) and NNTIME($t, 2^t$) are equal. The intuition behind this result is that the restriction on the allowed shifts in vector machines can be compared to the restriction on precision in nets.

The equality will be proved in two parts.

Theorem 22 *For any $t(n) \geq n$,* VECTOR – TIME(t) $\subseteq$ NNTIME($t^{O(1)}, 2^{O(t)}$). ∎

Proof. Let M be a restricted vector machine running in time $t(n)$. For a given n, let s be the minimum power of two such that the length of M's registers is always less than s, during the computation on an input of length n. It is easy to prove that $s = 2^{O(t(n))}$ (the restriction on the shifts is necessary here).

To simulate M by means of a neural net, we encode the contents of each register V_i of M as the activation value of a net processor v_i. More precisely, if V_i contains the vector $\ldots 000 b_\ell b_{\ell-1} \ldots b_2 b_1$, then $v_i = 0.\underbrace{000\ldots000 b_\ell b_{\ell-1}\ldots b_2 b_1}_{s}000\ldots$, and if V_i contains $\ldots 111 b_\ell b_{\ell-1}\ldots b_2 b_1$, then $v_i = 0.\underbrace{111\ldots111 b_\ell b_{\ell-1}\ldots b_2 b_1}_{s}000\ldots$ Note that $0 \leq v_i < 1$, and that $v_i \geq 1/2$ if and only if $V_i < 0$.

Initially, the net reads the input and stores it as the state of v_1, as described in Chapter 5. For the actual computation, we divide the proof in two parts: First, we show that the effect of each vector instruction can be simulated by rational functions and bitwise ANDs in time polylogarithmic in $s = 2^{O(t(n))}$, i.e., polynomial in $t(n)$. Then, we show that these sequences of operations, as well as the finite control of the vector machine, can be programmed in a neural net.

We simulate each vector instruction as follows:

- $V_i := x$: The constant x is built into the network as a weight, and this instruction sets $v_i := x$.
- $V_i :=$ **not** V_i: Build the rational 2^{-s} as described below for the shift instructions. Then set $v_i := 1 - 2^{-s} - v_i$.
- $V_i := V_j \wedge V_k$: This is simulated by a bitwise AND of v_j and v_k.
- $V_i := V_i \uparrow V_j$. This is simulated as:

  if $v_j < 1/2$ (i.e., $V_j \geq 0$) **then begin**
  build $y = 2^{-2^s v_j}$;
  $v_i := v_i / y$
  end

  Testing condition "$v_j < 1/2$" is equivalent to knowing whether $\sigma(4(v_j - 1/2))$ is 0 or 1. To compute y we use the algorithm given in the figure, which works in time $O(\log |V_j|)$. Because M is restricted, $|V_j|$ is $2^{O(t(n))}$ and thus the computation takes time $O(t(n))$.

```
/* compute y = 2^{-x}, where x = x_{s-1}...x_0 is given as a real x' = 0.x_{s-1}...x_0 */
p := 1/2;
for i := 1 to log s do
        p := p^2;
        /* p = 2^{-s} here; recall that s is a power of 2 */
y := 1; z := 1/2;
while x'/p ≥ 1 do begin                    /* digits left in x' */
        /* ∃i  (p = 2^{-s+i} ∧ z = 2^{-2^i} ∧ y = 2^{-(x_{i-1}...x_0)}) */
        if (x' ∧ p)/p = 1 then              /* x_i = 1 */
                y := y * z;
        p := 2 * p;
        z := z^2
end
```

Figure 11.1: Computing 2^{-x}

- $V_i := V_i \downarrow V_j$. Similar to left shift using product times 2^{-v_j} instead of division. Bitwise AND the result with the value $1 - 2^{-s}$ to remove the overflow to the right of the s designated bits.

- if $V_i = \mathbf{0}$ go to *label:* Compute 2^{-s} as above and then test whether $\sigma(v_i/2^{-s}) = 0$. Note that $\sigma(v_i/2^{-s}) = 1$ for all possible contents of V_i except for 0.

- *accept, reject:* To simulate these instructions, the net sets to 1 the output validation line and to 1 or 0, respectively, the output data line.

It remains to show that sequencing all these instructions can be hardwired into a network. Here we only provide an example: a subnet that implements the computation of $p = 2^{-s}$ following the first loop of the algorithm in the figure.

This network is triggered by the input a; it outputs its data via the neuron p and validates the output via the neuron v.

- The binary input a is 0 except for once. When 1, it triggers the network described below.

- The output validation neuron v is set to 1, $\log(s)$ steps after a triggers.

- The output data neuron p contains the value 2^{-s} when $v = 1$.

The internal neuron c counts the time. We assume that some neuron ℓ in the rest of the net provides the reciprocal of s, the maximum length that a register can have. For example, if $s = 32$, ℓ contains binary 0.00001 (recall that s is a power of two).

The update equations of the processors are:

$p^+ := \sigma(\ a/2 + (1-a)p^2\ )$

/* $a = 1$ resets p to $1/2$, $a = 0$ squares it */

$c^+ := \sigma(\ a \cdot \ell + (1-a) \cdot 2c\ )$

/* $a = 1$ resets counter to $1/s$ */

$v^+ := \sigma(\ 4c - 1\ )$

/* $v^+ = 0$ for $c \leq 1/4$, $v^+ = 1$ for $c \geq 1/2$ */

∎

Theorem 23 *For any* $t(n) \geq n$, $\mathrm{NNTIME}(t, 2^t) \subseteq \mathrm{VECTOR-TIME}(t^{O(1)})$.

Proof. Consider a net running in time $t(n)$ and within precision $2^{t(n)}$. To simulate the net by a vector machine, we keep the state of each processor in a vector register of length $2^{t(n)}$. Recall that addition, product, division, and bitwise AND of m-bit numbers can be computed in parallel machines in time $(\log m)^{O(1)}$ and $m^{O(1)}$ memory (see for example [KR90]). Thus, updating the state of each processor at each simulated step needs $t(n)^{O(1)}$ time and $2^{O(t(n))}$ memory on the vector machine. ∎

In fact, the simulations show that amount of memory in vector machines is polynomially related to net precision. The theorems were stated for at least linear running time, as the networks need linear time to read the input. However, the simulations work even for sublinear running times $t(n) \geq \log n$, if we adopt an alternative convention that the input is given to the net as the initial state of one of the processors, as in theorem 5 of Chapter 5. Then we can characterize NC, the class of sets accepted by Second Class machines in polylog time and polynomial space, as $\mathrm{NNTIME}(\log^{O(1)} n, n^{O(1)})$.

11.2 Extended Nets with Real Weights

In section 11.1 we have considered nets whose processors can compute rational functions and bitwise ANDs on their inputs, and shown that time in these nets is equivalent to time on parallel machines. If we allow real instead of rational weights, their power changes accordingly: we obtain nonuniform parallel time, or, equivalently, nonuniform circuit depth. For example, one can obtain the following analog of the fact that nonuniform circuits of bounded fan-in and linear depth can decide any set.

Theorem 24 *Every language is decided in linear time by a net with real weights whose processors compute rational functions and bitwise ANDs.*

Proof. The net contains a real weight whose binary expansion is the characteristic sequence of the language to decide. On an input that has lexicographical number i, the net computes the real $x = 2^{-i}$ using multiplication; it can do this as the input is entering. When the input ends, the net ANDs x with the real encoding the set, and divides the result by x, thus determining whether the input is in the set or not. ∎

Note that, in fact, the net has the answer two steps after the input has been read.

Chapter 12

The Complexity of Language Recognition by Neural Networks

One of the current applications of recurrent neural networks is as language acceptors. That is, given a string, the network's output is used to decide whether the string is in the language or not. A numerical (gradient-descent) technique is used to "infer" an accepting network from a set of training examples. Much effort has been directed towards practical implementations of this application. Several models of acceptors have been developed. See for instance [CSSM89, Elm90, GMC+92, Pol90, WZ89]. However, all of this has been done heuristically; some languages were found to be "learned", others were not. One of the main difficulties is that of deciding a priori on the proper size of the network to be used.

In this chapter we present some work of a heuristic and experimental nature which is complementary to the previous one on computational power of neural networks. This research represents an attempt to measure the "neural complexity" of languages, that is, to quantify the size of a recurrent neural network needed for acceptance of a given language.

We look at several different second order neural network models–with sigmoid, linear, threshold, or saturated functions–and sketch relationships between the number of neurons required in each of the models. We provide a technique for estimating the number of neurons necessary to recognize a regular language using a second order neural network with the linear activation function. We see that, roughly, the number in the linear activation model is an upper bound for the saturated and sigmoid models, and is a far better predictor than the minimal automaton size that had been used as an upper bound so far. Moreover, this bound is easy to compute, using techniques from the theory of rational power series in noncommuting variables.

We wish to emphasize once more that this chapter is mostly heuristic and experimental.

12.0.1 The Model in This Chapter

The second order recurrent neural network model that we consider in this chapter consists of N neurons in a fully connected architecture. The input I consists of M binary lines. Input is provided to the network one letter at a time, i.e. at time t letter $I(t)$ is input. Each letter i $(i = 1, \ldots, M)$ is presented in unary, as a vector which consists of $(M-1)$ 0's and a single 1 in the ith position.

Each neuron computes its next state as a combination of other neurons and the external input, using the following dynamics, as in [GMC+92]:

$$x_k(t+1) = \sigma\left(\sum_{i=1}^{N}\sum_{j=1}^{M} a_{ij}^k I_j(t) x_i(t)\right) \ . \tag{12.1}$$

Here σ can be one of the following functions:

- Linear $\mathcal{L}(x) := x$
- The classical Sigmoid
$$\sigma_s(x) = \frac{1}{1+e^{-x}}$$
- Heaviside (also called threshold)
$$\mathcal{H}(x) := \begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{if } x \geq 0 \ . \end{cases}$$

The *response* of the network to the string $\omega = \omega_1, \omega_2, \ldots \omega_l \in \Sigma_M^*$, represented as the input sequence $I(1), I(2), \ldots, I(l)$, is the value $x_1(l+1)$. The network *accepts* or *recognizes* a language $L \subseteq \Sigma_M^*$ if for every string $\omega \in \Sigma_M^*$

$$x_1(l+1) \geq \tau \Longleftrightarrow \omega \in L \ , \qquad x_1(l+1) < \tau \Longleftrightarrow \omega \notin L \ ,$$

where τ is a fixed real number, and a fixed initial state is assumed. That is, the language accepted consists of all the input strings to which the network responds with a value larger than τ. As in this chapter we discuss only second order neural networks, we forgo the word "second" and call them just "neural networks".

Our ultimate interest is in sigmoidal networks. Networks using the classical sigmoid were reported as relatively easy to train (see e.g.[GMC+92]) using gradient descent techniques, and

hence have been popular in implementations. (Another reason to focus on the sigmoid activation function stems from its biological plausibility of describing the graded response of a neuron.) As part of this work, we study linear and saturated networks as well.

We first develop an upper bound on the size of linear networks. This bound is not larger than the minimum non-deterministic FSA recognizing the language. Then, we develop a tighter upper bound that is based on the saturated network, and heuristically is appropriate for the sigmoid function. We see through a series of computer simulations that this bound is sometimes fairly accurate.

The remainder of this chapter is organized as follows: Section 12.1 deals with space complexity in linear-activation networks. We base the discussion on the theory of formal power series. This approach identifies languages with power series, and associates with each of these series a special kind of matrix, called a *Hankel matrix*. The rank of each of these Hankel matrices provides an upper bound on the space complexity of regular languages when using linear-activation networks. In Section 12.2, we provide an algorithm to estimate the complexity of regular languages in linear-activation networks. Section 12.3 compares different models of activation functions in terms of space complexity, thus suggesting that the space bound from Section 12.2 can be used to roughly bound the saturated and sigmoidal activation networks as well. We check experimentally this bound by working out several well-known examples.

12.1 Space Complexity in Linear Networks

Let Σ be an alphabet. A *language* L over Σ will be thought of as a function $f_L : \Sigma^* \to \{0,1\}$. We interpret $f_L(w)$ as the truth value of $w \in L$. A *fuzzy language* over Σ^* is a function $f_F : \Sigma^* \to \mathbb{R}$, where $\mathbb{R}$ denotes the real numbers. The value $f_F(\omega)$ denotes the degree of ambiguity, or degree of membership of the string ω in the language.

A *linear-activation* neural network $\mathcal{N}$ is a network with $\sigma = identity$. The response of the network $\mathcal{N}$ is a function $f_{\mathcal{N}} : \Sigma^* \to \mathbb{R}$, which is a fuzzy language. The language L *accepted by a neural network* $\mathcal{N}$ *with respect to* a threshold $\gamma \in \mathbb{R}$ is defined by: $w \in L \iff f_{\mathcal{N}}(\omega) \geq \gamma$.

The *l-complexity* of a language L is the size of the smallest linear-activation neural network accepting L with respect to some threshold $\gamma \in \mathbb{R}$.

12.1.1 Preliminaries

To estimate the l-complexity of regular languages, we discuss the notion of power series and Hankel matrices.

A *power series* r over Σ^* is a formal infinite sum of the form

$$r = \sum_{w \in \Sigma^*} C_w w\,, \quad \text{where each } C_w \in \mathbb{R}\ .$$

We can view a power series as an ambiguous, or fuzzy language: $\Sigma^* \to \mathbb{R}$. An *unambiguous* series is one for which $C_w \in \{0,1\}$ for all $w \in \Sigma^*$.

Similarly to the definition of the language accepted by a neural network, we say that a pair (r,q), where r is a power series over Σ^* and $q \in \mathbb{R}$, *defines* the language $L = L(r,q)$ by $w \in L \iff C_w \geq q$. We also say that a power series r which belongs to some pair defining L, is *related to the language* L.

Conversely, the *characteristic* power series of a language L is the series $r_c(L)$, with $C_w = f_L(w)$. This series is unambiguous. Note that $L(r_c(L), \frac{1}{2}) \equiv L$ for every language L.

From now on, we will assume for simplicity that we have a binary alphabet, $I = \{0,1\}^*$, i.e. $M = 2$ in Equation 12.1.

Definition 12.1.1 [SS78] The Hankel matrix, H_r, of the power series r is the infinite matrix whose rows and columns are indexed by the strings over $\{0,1\}^*$ —listed in the lexicographic order—and is defined by

$$H_r(u,v) = C_{uv}\ .$$

Example 12.1.2 The Hankel matrix $H_r(L)$ of the characteristic power series of $L = 1^*$ is given in Figure 12.1.

The columns are indexed by: λ, 0, 1, 00, 01, 10, 11, 000, 001 ... and the rows are indexed similarly. □

	λ	0	1	00	01	10	11	$\cdots$
λ	1	0	1	0	0	0	1	
0	0	0	0	0	0	0	0	
1	1	0	1	0	0	0	1	
00	0	0	0	0	0	0	0	
01	0	0	0	0	0	0	0	
$\vdots$								$\ddots$

Figure 12.1: A Hankel Matrix of the language 1*

If r is a power series related to a language L, than the matrix H_r is *associated* with L. Note that there are infinitely many Hankel matrices associated with L.

Definition 12.1.3 The *H-complexity* of a language L is

$$\text{H-Complexity(L)} = \min \{\text{rank}(H_r) | L = L(r, q) \text{ for some } q \in \mathbb{R}\} .$$

12.1.2 The Space Complexity Theorem

Theorem 25 *For every language L, l-complexity(L) = H-complexity(L). If L is regular, these numbers are finite.*

The proof consists of three parts: First we recall the equivalence between regular languages and a subset of power series. Then, we recall the relationships between the rank of the Hankel matrix of a power series and representations of this series. In the third part, we identify linear networks with these representations. We end up by combining the three parts into the required result.

Lemma 12.1.4 The regular languages over an alphabet Σ are the languages associated with the unambiguous series of $\mathbb{R}^{rat} \ll \Sigma^* \gg$, the set of rational series. (The set of rational series is the smallest rationally closed subset of the power series over Σ^* which contains all polynomials.)

Proof. We first cite Theorem (5.2) in [SS78]: If $r \in \mathbb{R}^{rat} \ll \Sigma^* \gg$ is unambiguous, then $L(r,q)$ is regular for every $q \in \mathbb{R}$. (Of course, the only interesting case is $q = 1$, or equivalently any $q \in (0,1]$. See also Theorem 2.3 in [Son75] for another proof.)

Conversely, theorem (5.1) in [SS78] states: Let L be a regular language. Then, $r_c(L)$ is in $\mathbb{R}^{rat} \ll \Sigma^* \gg$. ∎

Given a word $\omega = u_1 \cdots u_n$, we denote by $\tilde{\omega}$ its transpose, $u_n \cdots u_1$.

Lemma 12.1.5 [SS78] Let r be a power series over Σ^*. Then, $r \in \mathbb{R}^{rat} \ll \Sigma^* \gg$ iff the rank of H_r is finite. If H_r has finite rank N, then there exists a monoid representation

$$R : \Sigma^* \to \mathbb{R}^{N \times N}$$

and vectors $a \in \mathbb{R}^{1 \times N}$, $b \in \mathbb{R}^{N \times 1}$ such that

$$r = \sum_{w \in \Sigma^*} aR(\tilde{w})bw \ , \tag{12.2}$$

Moreover, if also

$$r = \sum_{w \in \Sigma^*} a'R'(\tilde{w})b'w$$

with $a' \in \mathbb{R}^{1 \times m}$, $b' \in \mathbb{R}^{m \times 1}$, $R'(\tilde{w}) \in \mathbb{R}^{m \times m}$, then $m \geq N$. Conversely, if there is any such representation then the rank of H_r is finite. □

That is, let L be a language and r be a power series related to L. The power series r admits a representation (12.2) of size equal to the rank of H_r, or none if the rank is infinite. The smallest dimension of a representation of a power series related to a language L, is equal to the H-complexity of the language. For regular languages, this is finite, by Lemma 12.1.4.

Lemma 12.1.6 The response of a linear-activation network to a string $\omega \in \Sigma^*$ can be written in the form

$$aR(\tilde{w})b \ ,$$

for some representation of the same size as the network. Conversely, any power series admitting a representation is a response of some linear-activation network of the same size.

Proof. We describe a linear-activation network of size N as follows: Each of the $M = 2$ possible input letters, I_j, is associated with a weight matrix A_j of size $N \times N$. At time t, the network receives an input letter, represented in unary by $I(t)$, and it changes its state according to the corresponding associated weight matrices:

$$x(t+1) = \sum_{i=1}^{2} I_i(t) A_i x(t) \ .$$

That is, if $I(t) = (1,0)$ or $I(t) = (0,1)$, then $x(t+1) = A_1 x(t)$ or $x(t+1) = A_2 x(t)$ respectively.

Given such a network, its output is designated by (without loss of generality) the first neuron. Let a be the row vector of size N, $a = (1,0,0,\ldots,0)$, corresponding to this output. Let the column vector b of size N represent the initial state of the neural network. Assume that the input to the system is $w = u_1, u_2, \ldots, u_k$, where each u_l is one of the 2 letters and its associated matrix is A_{i_l}. Then, the response of the system to the input string w is

$$a A_{i_k} \ldots A_{i_2} A_{i_1} b \ .$$

In general, we denote $R(w) = A_{i_1} \ldots A_{i_{(k-1)}} A_{i_k}$. We conclude that the responses of this network can be represented as in the statement of the lemma.

Conversely, given the representation (12.2) for a power series, we can choose a basis in the state-space so that $a = (1,0,0,\ldots,0)$, and a linear-activation network results. Hence, response of linear-activation networks and power series are equivalent. ∎

This completes the proof of Theorem 25. We are therefore motivated to estimate H-complexities.

12.2 Bounding The H-complexity

We define

$$\mathcal{H}\text{-complexity}(L) = \text{rank}\,(H_{r_c(L)}) \ .$$

By definition, for each language L, $\mathcal{H}$-complexity$(L) \geq$ H-complexity $= l$-complexity(L). That is, $\mathcal{H}$-complexity provides an upper bound on the l-complexity.

Given a regular language L represented as a regular expression, one can construct a nondeterministic finite automaton (NFA) (with "ϵ-moves") accepting it. The NFA outputs 1 for the

strings in L, and 0 for those that are not in L. An NFA can be thought of as a special case of a linear-activation network, for which the activations of the neurons are nonnegative integers, and the transition matrices consist of binary entries only. This gives a rise to a linear-activation network accepting L with $\gamma = \frac{1}{2}$. This network is not necessarily the smallest in size.

We can then use the algorithm described in [Son79] to get a minimal network. (The minimal network is not unique, but any two such networks can be shown to coincide up to a change of basis in the space of neuron states.) The algorithm takes time polynomial in the size of the regular expression. (See the related discussion in [Son88].)

From the above construction, we conclude that

Observation 12.2.1 For every language L, $\mathcal{H}$-complexity$(L) \leq |\ \mathrm{NFA}(L)|$, where $|\mathrm{NFA}(L)|$ is the number of states of a minimal size non-deterministic finite automaton accepting L. □

In Figure 12.2, the first to third columns (ignore the column labeled "experiment" for now), we use nine languages to compare the size of the associated minimal DFA with the complexity we found. The first seven are known by the name *Tomita languages*, see [Tom82], and the last two are parity and dual parity. The comparison with DFA sizes is given here because much of the previous literature had used this quantity as an estimate of "neural" complexity. (Clearly, the size of a minimal NFA is not larger than that of the minimal DFA.) The particular languages used for comparisons are those that were used in past grammatical inference studies based on neural networks, e.g. [GMC+92]. (In the case of Dualparity, the l-complexity can be estimated as 2, since

The Language		DFA's Size	$\mathcal{H}$-Complexity	Experiment
Tomita1:	1*	2	1	1
Tomita2:	(10*)	3	2	2
Tomita3:	no (1^{odd}) followed by (0^{odd})	5	3	3
Tomita4:	does not contain 000 substring	4	3	2
Tomita5:	([01+10] [01+10])*	7	4	4
Tomita6:	#1 - #0 is a multiple of 3	3	3	3
Tomita7:	0*1*0*1*	5	2	2
Parity:	even number of 1's	2	2	2
Dualparity:	even number of 0's, 1's	4	4	2

Figure 12.2: Comparison of DFA, $\mathcal{H}$-Complexity, and Experimental results

$r = C_\omega \omega$ with $C_\omega = (-1)^k + (-1)^l$ (k = number of zeroes, and l = numbers of ones) has rank 2, and $L(r,2)$ = Dualparity. This illustrates the fact that $\mathcal{H}$-complexity is not equal to the desired l-complexity, but it merely provides a bound.)

Remark 12.2.2 In the more realistic case in which the language L is not available, and the only data are a set of "training" strings together with information regarding their membership in the language, then one can show that even the question of finiteness of rank for the Hankel matrix is undecidable [Son75]. However, different heuristics might be useful is this case, such as repetitively enlarging the matrix until no change in its rank results after a few iterations.

Remark 12.2.3 The gap between l-complexity and $\mathcal{H}$-complexity can be arbitrarily large. This is easy to see from counting arguments. Indeed, there are an uncountable number of languages with l-complexity equal to 2—see for instance [Rab66], page 311—but it is easy to see that there are only a countable number of languages with finite $\mathcal{H}$-complexity.

12.3 Different Activation Functions: Using The H-Complexity As a Bound

The space complexity using networks employing *sigmoidal* activation functions:

$$\sigma(x) = \frac{1}{1 + e^{-x}} \quad ,$$

is in a sense bounded by the H-complexity: For any finite number of strings, we may use a low-gain approximation so as to cause the sigmoid to be close enough to the identity function in any given bounded domain. Thus, for any finite sample of a strings, the number of neurons required to accept it in the sigmoidal model is bounded by the number in the linear-activation one.

To estimate better the space required in the sigmoidal model, we used the second order neural network recognizer developed in [GMC+92], and trained the weights to generate acceptors for the languages shown in Figure 12.2. The number of neurons found experimentally as required to accept a language in the sigmoidal model is described in the fourth column. This number of neurons is clearly only an upper bound on the true minimum number of neurons required. The correlation

between the values predicted by the $\mathcal{H}$-complexity and the experimental values for sigmoids is striking, especially in comparison with the values that would be predicted from minimal automata size. Using high gain approximations, one can show that the space complexity using $\sigma_s(x)$ is also bounded by the complexity using $\mathcal{H}(x)$, on any finite sample.

Generally, the picture looks like:

Figure 12.3: The space complexity relations in a second order network

Chapter 13

Conclusions and Final Remarks

In this dissertation, we addressed several theoretical issues concerning recurrent neural networks. In our model, there are a finite number of simple processors. Each processor applies a piecewise linear activation function to an affine combination of the other activations and the external inputs. Such a model is analog by nature, given the continuity of the activation functions.

We formalized a digital input-output convention, in Chapter 3, which allows measuring the computational power of our networks. We showed that our homogeneous, simple model is not weaker (up to at most a polynomial slow down in the computation) than any other "reasonable" recurrent dynamical system consisting of a finite number of processors. This result motivates the following time bounded analog analogy of Church's thesis:

Any reasonable analog computer will have no more power (up to polynomial time) than first-order recurrent networks.

Focusing on the specific piecewise-linear recurrent network model, we related its computational power to the types of real constants allowed as weights. We established the existence of an infinite hierarchy of complexity classes of languages recognized by networks. When no constraints are placed on the weights, that is, they can be arbitrary real numbers, our networks compute in polynomial time exactly the nonuniform class P/poly (Chapter 6). If the weights can only be rational numbers (Chapter 5), networks are polynomial time equivalent to Turing machines. Taking it to an extreme, networks with integer weights are equivalent to finite automata (Chapter 4).

The proper hierarchy of classes was revealed in the range of numbers between the rationals and the reals. There, we characterized numbers in an information theoretic manner, using a time bounded Kolmogorov type characterization (Chapter 7). We related the associated networks to

different non-uniform complexity classes.

Another way to define a proper hierarchy of networks was obtained by setting precision constraints on the individual neurons, in Chapter 9. These classes were compared against space classes in digital computation.

A lower bound was obtained on the complexity of networks utilizing a very general class of activation functions. We showed that any network using any activation function which has different limits at $\pm\infty$ simulates at least finite automata (Chapter 10).

An interesting structural complexity question arises if one allows neurons to compute more complex functions. We showed that second class power (i.e. unbounded parallelism) is attained by networks with certain specific (and very rich) type of neurons (Chapter 11).

Finally, we complemented our theoretical work with heuristic estimates on the size of networks required to recognize specific regular languages (Chapter 12). This bound was also compared with experimental results.

The work described in this thesis opens up many further questions regarding the foundations of recurrent networks and analog computation. Let me summarize a few of them:

- The exact characterization of networks whose activation is the classical sigmoid is still open. They were proved in [KS93] to compute any recursive function, and in Chapter 8 we proved that they compute not more than the class P/poly (i.e., according to our "Analog Church's Thesis"). This range is, however, very large.

- Our networks are deterministic (non-random) and may compute exact real values. An interesting question is to characterize networks for which these two principles do not hold.

- Many dynamical systems that perform computations are closely related to our model. In particular, there are models in which processors compute discontinuous functions, as in the model of complexity over the reals ([BSS89]). The precise relationships among such models and our networks is still unclear.

- The question of finding bounds on the minimum sizes of different types of networks needed to compute specified functions in "real-time" is of great implementational importance. Here, we have provided very preliminary work.

- We developed a real-time language (Chapter 2) that compiles into a recurrent network. Our development was for theoretical purposes. However, after dealing with a few time concepts and safety properties, one might obtain a powerful language useful for analog implementations.
- Most of our work set foundations for understanding the computational power of networks. It is of a great importance to develop learning algorithms and design procedures that will allow this power to be usefully exploited.
- Our models operate in discrete time. Other approaches to analog computing (see e.g. [Won92] and the references there) are based on differential equations. We strongly conjecture that the "Analog Church's Thesis," when properly formulated, will still be valid for continuous time systems.

References

[AA87] J. Alspector and R.B. Allen. A neuromorphic vlsi learning system. In P. Loseleben, editor, *Advanced Research in VLSI: Proceedings of the 1987 Stanford Conference*, pages 313–349, Cambridge, MA, 1987. MIT Press.

[ADO91] N. Alon, A.K. Dewdney, and T.J. Ott. Efficient simulation of finite automata by neural nets. *J. A.C.M.*, 38:495–514, 1991.

[ASM93] F. Albertini, E.D. Sontag, and V. Maillot. Uniqueness of weights for neural networks. In R. Mammone, editor, *Artificial Neural Networks with Applications in Speech and Vision*. Chapman and Hall, London, 1993.

[Atk89] K.E. Atkinson. *An Introduction to Numerical Analysis*. Wiley, New York, 1989.

[Bar92] A.R. Barron. Neural net approximation. In *Proc. Seventh Yale Workshop on Adaptive and Learning Systems*, pages 69–72, Yale University, 1992.

[Bat91] R. Batruni. A multilayer neural network with piecewise-linear structure and back-propagation learning. *IEEE Transactions on Neural Networks*, 2:395–403, May 1991.

[BDG90] J.L. Balcázar, J. Díaz, and J. Gabarró. *Structural Complexity*, volume I and II. Springer-Verlag EATCS Monographs, Berlin, 1988-1990.

[BGSS93] J. L. Balcázar, R. Gavaldà, H.T. Siegelmann, and E. D. Sontag. Some structural complexity aspects of neural computation. In *IEEE Structure in Complexity Theory Conference*, pages 253–265, San Diego, CA, May 1993.

[BGV88] J.R. Brown, M.M. Garber, and S.F. Vanable. Articifial neural network on a simd architecture. In *Proc. 2nd Symposium on the Frontier of Massively Parallel Computation*, pages 43–47, Fairfax, VA, 1988.

[BH89] E.B. Baum and D. Haussler. What size net gives valid generalization? *Neural Computation*, 1:151–160, 1989.

[BHM92] J. L. Balcázar, M. Hermo, and E. Mayordomo. Characterizations of logarithmic advice complexity classes. *Information Processing 92, IFIP Transactions A-12*, 1:315–321, 1992.

[BR88] J. Berstel and C. Reutenauer. *Rational Series and their Languages*. Springer-Verlag, Berlin, 1988.

[BR90] A. Blum and R.L. Rivest. Training a 3-node neural network is np-complete. In D.S. Touretzky, editor, *Advances in Neural Information Processing Systems 2*, pages 9–18. Morgan Kaufmann, San Mateo, CA, 1990.

[BSS89] L. Blum, M. Shub, and S. Smale. On a theory of computation and complexity over the real numbers: Np completeness, recursive functions, and universal machines. *Bull. A.M.S.*, 21:1–46, 1989.

[CD89] S.M. Carroll and B.W. Dickinson. Construction of neural nets using the radon transform. In *Proc. Int. Joint Conf. Neural Networks*, volume I, pages 607–611, 1989.

[Che80] G.W. Cherry. *Pascal Programming Structure: An Introduction To Systematic Programming*. Reston Publishing, Reston Virginia, 1980.

[CSSM89] A. Cleeremans, D. Servan-Schreiber, and J. McClelland. Finite state automata and simple recurrent recurrent networks. *Neural Computation*, 1, No. 3:372, 1989.

[Cyb89] G. Cybenko. Approximation by superpositions of a sigmoidal function. *Math. Control, Signals, and Systems*, 2:303–314, 1989.

[DDGS93] C. Darken, M. Donahue, L. Gurvits, and E. Sontag. Rate of approximation results motivated by robust neural network learning. In *Proc. Sixth ACM Workshop on Computational Learning Theory*, Santa Cruz, July 1993.

[DS92] D. DasGupta and G. Schnitger. The power of approximating: a comparison of activation functions. In *Conf. on Neural Information Processing Systems*, Denver, 1992.

[EDK+89] S. P. Eberhardt, T. Daud, D. A. Kerns, T. X. Brown, and A. P. Thakoor. Competitive neural architecture for hardware solution to the assignment problem. *Neural Networks*, 4, 1989.

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

[FG90] S. Franklin and M. Garzon. Neural computability. In O. M. Omidvar, editor, *Progress In Neural Networks*, pages 128–144. Ablex, Norwood, NJ, 1990.

[Fra89] J.A. Franklin. On the approximate realization of continuous mappings by neural networks. *Neural Networks*, 2:183–192, 1989.

[GF89] M. Garzon and S. Franklin. Neural computability. In *Proc. 3rd Int. Joint Conf. Neural Networks*, volume II, pages 631–637, 1989.

[GJ79] M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman, New York, 1979.

[GMC+92] C. L. Giles, C.B. Miller, D. Chen, H.H. Chen, G.Z. Sun, and Y.C. Lee. Learning and extracting finite state automata with second-order recurrent neural networks. *Neural Computation*, 4(3), 1992.

[Has87] J. H. Hastard. *Computational limitations for small depth circuits*. PhD thesis, Massachusetts Institute of Technology, 1987.

[HMP92] T.A. Henziger, Z. Manna, and A. Pnueli. Temporal proof methodologies for real-time systems. Technical report, School of Formal techniques in real-time and fault-tolerant systems, Univ. of Nijmegen, The Netherlands, 1992.

[Hon88] J.W. Hong. On connectionist models. *On Pure and Applied Mathematics*, 41, 1988.

[Hop84] J.J. Hopfield. Neurons with graded responses have collective computational properties like those of two-state neurons. In *Proc. of the Natl. Acad. of Sciences*, volume 81, pages 3088–3092, USA, 1984.

[Hor91] K. Hornik. Approximation capabilities of multilayer feedforward networks. *Neural Networks*, 4:251–257, 1991.

[HS87] R. Hartley and H. Szu. A comparison of the computational power of neural network models. In *Proc. IEEE Conf. Neural Networks*, pages 17–22, 1987.

[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.

[HU79] J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.

[Jud90] J.S. Judd. *Neural Network Design and the Complexity of Learning*. MIT Press, Cambridge, MA, 1990.

[KL82] R.M. Karp and R. Lipton. Turing machines that take advice. *Enseign. Math.*, 28, 1982.

[Kle56] S. C. Kleene. Representation of events in nerve nets and finite automata. In C.E. Shannon and J. McCarthy, editors, *Automata Studies*, pages 3–41. Princeton Univ. Press, 1956.

[Kob81] K. Kobayashi. On compressibility of infinite sequences. Technical Report Research Report C-34, Department of Information Sciences, Tokyo Institute of Technology, 1981.

[KR90] R.M. Karp and V. Ramachandran. Parallel algorithms for shared-memory machines. In *Handbook of Theoretical Computer Science*, volume A, pages 869–941. MIT/Elsevier, 1990.

[KS93] J. Kilian and H.T. Siegelmann. On the power of sigmoid neural networks. In *Proc. Sixth ACM Workshop on Computational Learning Theory*, Santa Cruz, July 1993.

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

[LLPS93] M. Leshno, V. Y. Lin, A. Pinkus, and S. Schocken. Multilayer feedforward networks with a non-polynomial activation function can approximate any function. *Neural Networks*, 6, 1993.

[Maa93] W.G. Maass. Bounds for the computational power and learning complexity of analog neural nets. In *Proceeding of the 25 Annual ACM Symposium on Theory of Computation*, San Diego, May 1993.

[Mac92] B. J. MacLennan. Continuous symbol systems: The logic of connectionism. In D.S. Levine and M.Ãparicio IV, editors, *Neural Networks for Knowledge Representation and Inference*. Lawrence Erlbaum, Hillsdale, NJ, 1992.

[Mat92] M. Matthews. On the uniform approximation of nonlinear discrete-time fading-memory systems using neural network models. Technical Report Ph.D. Thesis, ETH No. 9635, E.T.H. Zurich, 1992.

[Min67] M. L. Minsky. *Computation: Finite and Infinite Machines.* Prentice Hall, Engelwood Cliffs, 1967.

[Mot93] L. Motus. Time concepts in real-time software. *Control Enginnering Practice*, 1:21–33, Feb 1993.

[MP43] W. S. McCulloch and W. Pitts. A logical calculus of the ideas immanent in nervous activity. *Bull. Math. Biophys*, 5:115–133, 1943.

[MP88] M. Minsky and S. Papert. *Perceptrons: An Introduction to Computational Geometry.* MIT Press, Cambridge, 1988.

[MS93] M. Macintyre and E.D. Sontag. Finiteness results for sigmoidal 'neural' networks. In *Proceeding of the 25 Annual ACM Symposium on Theory of Computation*, San Diego, May 1993.

[MSS91] W. Maass, G. Schnitger, and E.D. Sontag. On the computational power of sigmoid versus boolean threshold circuits. In *Proc. 32nd IEEE Symp. Foundations of Comp. Sci*, pages 767–776, 1991.

[Mul56] D.E. Muller. Complexity in electronic switching circuits. *IRE Trans. Electronic Comp.*, 5:15–19, 1956.

[Mur71] S. Muroga. *Threshold Logic and its Applications.* Wiley, New York, 1971.

[Par92] I. Parberry. Knowledge, understanding, and computational complexity. Technical Report CRPDC-92-2, Department of Computer Sciences, University of North Texas, Feb 1992.

[Par93] I. Parberry. *The Computational and Learning Complexity of Neural Networks.* draft, 1993.

[Pen89] R. Penrose. *The Emperor's New Mind.* Oxford University Press, Oxford, 1989.

[PI91] M. M. Polycarpou and P.A. Ioannou. Identification and control of nonlinear systems using neural network models: Design and stability analysis. Technical Report Report 91-09-01, Department of EE/Systems, USC, Los Angeles, Sept 1991.

[Pol87] J. B. Pollack. *On Connectionist Models of Natural Language Processing.* PhD thesis, Computer Science Dept, Univ. of Illinois, Urbana, 1987.

[Pol90] J.B. Pollack. The induction of dynamical recognizers. Technical Report 90-JP-Automata, Dept of Computer and Information Science, Ohio State U., 1990.

[PS76] V.R. Pratt and L.J. Stockmeyer. A characterization of the power of vector machines. *Journal of Computer and System Sciences*, 12:198–221, 1976.

[Rab66] M. Rabin. Lectures on classical and probabilistic automata. In E.R. Caianiello, editor, *Automata Theory.* Academic Press, London, 1966.

[Ros62] F. Rosenblatt. *Principles of Neurodynamics.* Spartan Books, New York, 1962.

[Sav76] J.E. Savage. *The Complexity of Computing.* Wiley, New York, 1976.

[SCLG91] G.Z. Sun, H.H. Chen, Y.C. Lee, and C.L. Giles. Turing equivalence of neural networks with second order connection weights. In *Proceedings of the International Joint Conference on Neural Networks, IEEE*, 1991.

[Sha56] C. E. Shannon. A universal turing machine with two internal states. In C.E. Shannon and J. McCarthy, editors, *Automata Studies*, pages 156–165. Princeton Univ., 1956.

[Son75] E.D. Sontag. On certain questions of rationality and decidability. *J. Comp. Syst. Sci.*, 11:375–381, 1975.

[Son79] E. D. Sontag. Realization theory of discrete-time nonlinear systems: Part i- the bounded case. *IEEE Trans.Circuits and Syst.*, 26:342–356, 1979.

[Son88] E.D. Sontag. Controllability is harder to decide than accessibility. *SIAM J. Control and Optimization*, 26(6):1106–1118, 1988.

[Son90] E. D. Sontag. *Mathematical Control Theory: Deterministic Finite Dimensional Systems*. Springer, New York, 1990.

[Son92a] E. D. Sontag. Feedforward nets for interpolation and classification. *J. Comp. Syst. Sci*, 45:20–48, 1992.

[Son92b] E.D. Sontag. Feedback stabilization using two-hidden-layer nets. *IEEE Trans. Neural Networks*, 3:981–990, 1992.

[Son92c] E.D. Sontag. Neural nets as systems models and controllers. In *Proc. Seventh Yale Workshop on Adaptive and Learning Systems*, pages 73–79, Yale University, 1992.

[SS78] A. Salomaa and M. Soittola. *Automata-Theoretic Aspects of Formal Power Series*. Springer-Verlag, New-York, 1978.

[SS91a] H. T. Siegelmann and E. D. Sontag. Turing computability with neural nets. *Appl. Math. Lett.*, 4(6):77–80, 1991.

[SS91b] E.D. Sontag and H.J. Sussmann. Backpropagation separates where perceptrons do. *Neural Networks*, 4:243–249, 1991.

[SS92] H. T. Siegelmann and E. D. Sontag. On the computational power of neural nets. In *Proc. Fifth ACM Workshop on Computational Learning Theory*, pages 440–449, Pittsburgh, July 1992.

[SS93] H. T. Siegelmann and E. D. Sontag. Analog computation via neural networks. In *The second Israel Symposium on Theory of Computing and Systems*, Natanya, Israel, June 1993.

[SSG92] H. T. Siegelmann, E. D. Sontag, and C. L. Giles. The complexity of language recognition by neural networks. In J. van Leeuwen, editor, *Algorithms, Software, Architecture (Proceedings of IFIP 12th World Computer Congress)*, pages 329–335, Amsterdam, 1992. North Holland.

[SSar] H. T. Siegelmann and E. D. Sontag. Analog computation via neural networks. *Theoretical Computer Science*, to appear.

[Ste84] G.L. Steeler. *Common LISP: The language.* Digital Equipment Cooperation USA, 1984.

[Sus92] H.J. Sussmann. Uniqueness of the weights for minimal feedforward nets with a given input-output map. *Neural Networks*, 5:589–593, 1992.

[SW90] M. Stinchcombe and H. White. Approximating and learning unknown mappings using multilayer feedforward networks with bounded weights. In *Proceedings of the International Joint Conference on Neural Networks, IEEE*, 1990.

[Tom82] M. Tomita. Dynamic construction of finite-state automata from examples using hill-climbing. In *Proceedings of the Fourth Annual Cognitive Science Conference*, pages 105–108, Ann Arbor MI, 1982.

[VEB90] P. Van Emde Boas. Machine models and simulations. In *Handbook of Theoretical Computer Science*, volume A, pages 1–66. MIT/Elsevier, 1990.

[VSD86] A. Vergis, K. Steiglitz, and B. Dickinson. The complexity of analog computation. *Math. and Computers in Simulation*, 28:91–113, 1986.

[Wol91] D. Wolpert. A computationally universal field computer which is purely linear. Technical Report LA-UR-91-2937, Los Alamos National Laboratory, 1991.

[Won92] W.S. Wong. Solving combinatorial optimization problems by gradient flows. In *Proc. IEEE Conf. Decision and Control*, pages 1494–1496, Tucson, Dec 1992. IEEE.

[WZ89] R.J. Williams and D. Zipser. A learning algorithm for continually running fully recurrent neural networks. *Neural Computation*, 1, No. 2, 1989.

[Yao85] A. Yao. Separating the polynomial-time hierarchy by oracles. In *Proc. 22nd IEEE Symp. Foundations of Comp. Sci.*, pages 1–10, 1985.

[Yas71] A. Yasuhara. *Recursive Function Theory and Logic.* Academic Press, New York, 1971.

[ZZZ92] B. Zhang, L. Zhang, and H. Zhang. A quantitative analysis of the behavior of the pln network. *Neural Networks*, 5:639–661, 1992.

Vita

Hava (Eve) Tova Siegelmann

1980 Graduated from Hare'ali Ha-ivri High school, Haifa, Israel.

1982-84 Military Personnel, Israel Defense Forces, Israel.

1984-88 Undergraduate in Technion (IIT), Haifa, Israel. B. A. in Computer Science.

1987-88 Graduate Teaching Assistant, Departments of Computer Science and Mathematics, Israel Institute of Technology (Technion), Haifa, Israel.

1988 Summer Intern, Telecommunication Research Laboratories of Finland, Helsinki, Finland.

1988-92 Graduate work in Hebrew University, Jerusalem, Israel. M. Sc. in Computer Science.

1988 Graduate Teaching Assistant, Department of Computer Science, The Hebrew University, Jerusalem, Israel.

1989 Research Associate, School of Computer Information Science, Syracuse University, Syracuse, New York.

1990-93 Graduate work in Computer Science, Rutgers, The State University of New Jersey.

1990-91 Rutgers Doctoral Fellow, Department of Computer Science, Rutgers.

1990 Summer - Graduate Research Assistant, Department of Computer Science, Rutgers.

1991 Summer intern in NEC Research, Princeton, New Jersey.

1992-93 Graduate Research Assistant, Department of Computer Science, Rutgers.

1992 Summer intern in NEC Research, Princeton, New Jersey.

1993 Ph.D. in Computer Science.

Publications

1991 **1.** Siegelmann H. T. and B. R. Badrinath, "Integrating Implicit Answers with Object-Oriented Queries," *Proceedings of the Conference on Very Large Data Bases,* Barcelona, Spain, September 1991.

2. Siegelmann H. T. and O. Frieder, "The Allocation of Documents in Multiprocessor Information Retrieval Systems: An Application of Genetic Algorithms," *Proceedings of the IEEE Conference on Systems, Man, and Cybernetics,* Charlottesville, Virginia, October 1991.

3. Frieder O. and H. T. Siegelmann, "On the Allocation of Documents in Information Retrieval Systems," *Proceedings of the ACM Fourteenth Conference on Information Retrieval (SIGIR),* Chicago, Illinois, October 1991.

4. Siegelmann H. T. and E. D. Sontag, "Turing Computability with Neural Networks," *Applied Mathematics Letters, 4(6),* (1991): 77-80.

1992

5. Siegelmann H. T. and E. D. Sontag, "On the Computational Power of Neural Networks" *Proceedings of the Fifth ACM Workshop on Computational Learning*, Pittsburgh, July 1992, 440-449. *Theory,* Pittsburgh, Penn., July 1992.

6. Siegelmann H. T., E. D. Sontag and C. L. Giles, "The Complexity of Language Recognition by Neural Networks," *Algorithms, Software, Architecture, (J. van Leeuwen, ed), North Holland, Amsterdam,* 1992, pp. 329-335. (Proceedings of IFIP 12th World Computer Congress.)

7. Siegelmann H. T. and E. D. Sontag, "Some Recent Results on Computing With 'Neural Nets'," *IEEE Conference on Decision and Control,* Tuscon, Arizona, December 1992: 1476-1481.

1993

8. Balcázar J. L., R. Gavaldà, H. T. Siegelmann, and E. D. Sontag, "Some Structural Complexity Aspects of Neural Computation," *IEEE Structure in Complexity Theory Conference,* San Diego, California, May 1993.

9. Siegelmann H. T. and E. D. Sontag, "Analog Computation Via Neural Networks," *The second Israel Symposium on Theory of Computing and Systems,* Natanya, Israel, June 1993.

10. Kilian J. and H. T. Siegelmann, "Computability With The Classical Sigmoid," *Proceedings of the Fifth ACM Workshop on Computational Learning*, Santa Cruz, July 1993.

To appear **11.** Siegelmann H. T. and O. Frieder, "Document Allocation in Multiprocessor Information Retrieval Systems," chapter in *Lecture note series in Computer Science: Advanced Database Concepts and Research Issues*, editors Nabil R. Adam and Bharat Bhargava, Springer Verlag, November 1993.

12. Siegelmann H. T. and E. D. Sontag, "Analog Computation Via Neural Networks," in *Theoretical Computer Science.*