Report *SYCON-91-11*

ON THE COMPUTATIONAL POWER OF NEURAL NETS

Hava T. Siegelmann, Department of Computer Science
Eduardo D. Sontag, Department of Mathematics

Rutgers University, New Brunswick, NJ 08903
E-mail: siegelma@paul.rutgers.edu, sontag@hilbert.rutgers.edu

ABSTRACT

This paper deals with the simulation of Turing machines by neural networks. Such networks are made up of interconnections of synchronously evolving processors, each of which updates its state according to a "sigmoidal" linear combination of the previous states of all units.

The main result states that one may simulate all Turing machines by nets, 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.

(This is an update of Report SYCON-91-08; new results include the simulation in linear time of binary-tape machines, as opposed to the unary alphabets used in the previous version.)

Key words: neural networks, Turing machines

Research supported in part by US Air Force Grant AFOSR-91-0343.

Rutgers Center for Systems and Control
November 1991

ON THE COMPUTATIONAL POWER OF NEURAL NETS*

Hava T. Siegelmann, Department of Computer Science
Eduardo D. Sontag, Department of Mathematics

Rutgers University, New Brunswick, NJ 08903
E-mail: siegelma@paul.rutgers.edu, sontag@hilbert.rutgers.edu

ABSTRACT

This paper deals with the simulation of Turing machines by finite neural networks. Such networks are made up of interconnections of synchronously evolving processors, each of which updates its state according to a "sigmoidal" linear combination of the previous states of all units.

The main result states that one may simulate all Turing machines by nets, 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.

1 Introduction

We study the computational capabilities of recurrent first-order neural networks, or as we shall say from now on, *processor nets*. Such nets consist of interconnections (with possible feedback) of a finite number N of synchronously evolving processors. The state $x_i(t)$ of the ith processor at each time $t = 1, 2, \ldots$ is described by a scalar quantity; $x_i(t)$ is updated at each t according to $\sigma(\ldots)$, where the expression inside the parenthesis is an affine (i.e., linear + bias) combination, with rational coefficients, of the previous states $x_i(t-1)$ of all processors and an external input signal $u(t)$. The function σ is the simplest possible "sigmoid," namely the saturated-linear function

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

One of the processors is singled out as the "output node" of the net. Another processor, called the "validation node," signals when the output occurs. We shall prove that processor nets are universal computing devices.

The use of sigmoidal functions —as opposed to hard thresholds— is what distinguishes this area from older work that dealt only with finite automata. Indeed, it has long been known, at least since the classical papers by McCulloch and Pitts ([12], [9]), how to implement logic gates by threshold networks, and therefore how to simulate finite automata by such nets. For us, however, nets are essentially *analog* computational devices, in accordance with models currently used in neural net practice.

In [14], Pollack argued that a certain recurrent net model, which he called a "neuring machine," is universal. The model in [14] 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 using multiplications as opposed to just linear combinations. Pollack left as an open question that of establishing if high-order connections

*This research was supported in part by US Air Force Grant AFOSR-91-0343.
Keywords: neural networks, Turing machines.

are really necessary in order to achieve universality, though he conjectured that they are. High-order models have also been used in [6] and [20], as well as by many other authors, often with an added infinite external memory device. Underlying the use of high-order nets is the conjecture that their computational power is superior to that of linearly interconnected nets.

Also related is the work reported in [7], [4], and [5], some of which deals with cellular automata. There one assumes an unbounded number of neurons, as opposed to a finite number fixed in advance. This potential infinity is analogous to the potentially infinite tape in a Turing machine; in our work, the activation values themselves are used instead to encode unbounded information, much as is done with the standard computational model of register machines.

Our main result states that one can simulate all Turing machines by nets, using only first-order (i.e., linear) connections. 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.

Note that we restrict to rational states and weights in order to preserve computability. It is possible to prove that using *real valued* weights results in processor nets that can "compute" *arbitrary* partial functions (not necessarily recursive); this can be achieved (see [17]) by encoding all information in the network weights. In fact, with zero initial states, and restricting for simplicity to language recognition, the situation is as follows:

Weights	Recog. Power
integer	regular
rational	recursive
real	arbitrary

The idea of using continuous-valued neurons in order to attain gains in computational capabilities as compared with threshold gates had been investigated before. However, prior work considered only the special case of feedforward nets –see for instance [19] for questions of approximation and function interpolation, and [10] for questions of Boolean circuit complexity.

The paper is organized as follows. First we define precisely nets and we state the main result. Then we prove the result through a few intermediate steps, and we provide an estimate of the number of processors used. Finally, we make some remarks on further directions of research.

2 Statement of Result

We model a net, as described in the Introduction, as a dynamical system. At each instant, the state of this system is a vector $x(t) \in \mathbb{Q}^N$ of rational numbers, where the ith coordinate keeps track of the activation value of the ith processor. There are two input lines. The first one has the role of a *data line*: it carries a binary input signal. When no signal is being transferred, the data line assumes the default value "0". The other line is used to indicate when the input is active. It is set to "1" as long as the input is present, and is "0" when the information line is not active. This line, the *validation line*, is reset to "0" only once, thus guaranteeing that the input data will appear consecutively. We refer to these lines as "D" and "V". A similar convention applies to the output processors.

In general, a discrete-time dynamical system (with two binary inputs) is specified by a *dynamics map*

$$\mathcal{F} \ : \ \mathbb{Q}^N \times \{0,1\}^2 \to \mathbb{Q}^N$$

where N is a positive integer, the *dimension* of the system. Given any initial state $x^0 \in \mathbb{Q}^N$, and any infinite sequence

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

with each $u_i(t) = (D_i(t), V_i(t)) \in \{0,1\}^2$ (thought of as external inputs), one defines the *state at time t*, for each integer $t \geq 1$, as the value obtained by recursively solving the equations:

$$x(1) := x^{init}, \quad x(t+1) \ := \ \mathcal{F}(x(t), u(t)) \, , \quad t = 1, 2 \ldots$$

From now on we write just $x^+ = \mathcal{F}(x, u)$ to display such a difference equation. We also assume that two coordinates of x, x_o and x_v (the "output node" and "validation node", respectively) have been selected. The sequence

$$y(t) = (x_o(t), x_v(t)), \ \ 1 = 1, 2, \ldots$$

is called the "output produced by the input u" (for a given initial state).

For each $N \in \mathbb{N}$, we denote the mapping

$$(q_1, \ldots, q_N) \mapsto (\sigma(q_1), \ldots, \sigma(q_N)) \quad \text{by} \quad \vec{\sigma}_N \ : \ \mathbb{Q}^N \to \mathbb{Q}^N \, ,$$

and we drop the subscript N when clear from the context. (We think of elements of $\mathbb{Q}^N$ as column vectors, but for display purposes we sometimes show them as rows, as above. As usual, $\mathbb{Q}$ denotes the rationals, and $\mathbb{N}$ denotes the natural numbers, not including zero.)

Here is a formal definition.

Definition 2.1 A *σ-processor net* $\mathcal{N}$ with two binary inputs is a dynamical system having a dynamics map of the form

$$\mathcal{F}(x, u) \ = \ \vec{\sigma}(Ax + b_1 u_1 + b_2 u_2 + c) \, ,$$

for some matrix $A \in \mathbb{Q}^{N \times N}$ and three vectors $b_1, b_2, c \in \mathbb{Q}^N$. □

The "bias" vector c is not needed, as one may always add a processor with constant value 1, but using c simplifies the notation and allows the use of zero initial states, which seems more natural. When $b_1 = b_2 = 0$, one has a net *without inputs*. Processor nets appear frequently in neural network studies, and their dynamic properties are of interest (see for instance [11]).

It is obvious that —with zero, or more general rational, initial state— one can simulate a processor net with a Turing machine. We wish to prove, conversely, that any function computable by a Turing machine can be computed by such a processor net. We look at partially defined maps $\phi : \{0,1\}^* \to \{0,1\}^*$ that can be recursively computed. In other words, maps for which there is a Turing machine M so that, when a word $\omega \in \{0,1\}^*$ is written initially on the tape, M halts on ω if and only if $\phi(\omega)$ is defined, and $\phi(\omega)$, in that case, is the content of the tape when the machine halts.

In order to state precisely the simulation result, we need to encode the information about ϕ into suitable pairs of signals (data, validation) for inputs and outputs of a dynamical system. This is done as follows: For each $\omega = a_1 \cdots a_k \in \{0,1\}^*$, with $\phi(\omega) = b_1 \cdots b_l \in \{0,1\}^*$ or undefined, and each $r \in \mathbb{N}$, define the six functions

$$V_\omega, D_\omega, G_{\omega,r}, H_{\omega,r} : \mathbb{N} \to \{0,1\} ,$$

and

$$u_\omega, y_\omega : \mathbb{N} \to \{0,1\}^* ,$$

as follows:

$$V_\omega(t) = 1 \text{ for } t = 1, \ldots, k, \qquad V_\omega(t) = 0 \text{ for } t > k ,$$
$$D_\omega(t) = a_{k-t+1} \text{ for } t = 1, \ldots, k, \qquad D_\omega(t) = 0 \text{ for } t > k ,$$

$G_{\omega,r}(t) = 1$ for $t = r, \ldots, (r + l - 1)$ if the output $\phi(\omega)$ is defined, and is 0 when $\phi(\omega)$ is not defined, or if t is outside of the given range; and finally $H_{\omega,r}(t) = b_{t-r+1}$ for $t = r, \ldots, (r+l-1)$ if the output $\phi(\omega)$ is defined, and is 0 when $\phi(\omega)$ is not defined, or if t is outside of the given range. The functions

$$u_\omega(t) = (D_\omega(t), V_\omega(t)) , \qquad y_{\omega,r}(t) = (H_{\omega,r}(t), G_{\omega,r}(t))$$

are defined for all t.

Theorem 1 *Let $\phi : \{0,1\}^* \to \{0,1\}^*$ be any recursively computable partial function. Then, there exists a processor net $\mathcal{N}$ with the following property: If $\mathcal{N}$ is started from the zero (inactive) initial state, and the input sequence u_ω is applied, $\mathcal{N}$ produces the output $y_{\omega,r}$ (where $H_{\omega,r}$ appears in the "output node," and $G_{\omega,r}$ in the "validation node") for some r. Furthermore, if a Turing machine M computes $\phi(\omega)$ in time $T(\omega)$, then one may take $r(\omega) = 4T(\omega) + O(|\omega|)$.*

Note that in particular it follows that one can obtain the behavior of a universal Turing machine via some net. An upper bound from the construction shows that $N = 1058$ processors are sufficient for computing such a universal partial recursive function.

We now describe the main ideas of the proof; details are provided below. First of all, one starts with a simulation of $\mathcal{M}$ through a push-down automaton with two binary stacks ([8]). The control unit can be easily simulated by a net; this is basically the old automata result ([12]), but care must be taken in seeing that it is possible to let inputs (from stacks) enter additively rather than multiplicatively. More than that, the time per simulated operation must be kept to a constant.

The contents of each stack can be viewed as a rational number between 0 and 1 of the form $\frac{p}{4^q}, 0 < p < 4^q$. We think of the ith element from the top as corresponding to the ith element to the right of the decimal point in a finite expansion in base 4. A "0" stored in the stack is

associated with a "1" in the expansion, while a "1" in the stack is associated with the value "3". An empty stack is represented by zero. Thus, only numbers of the special form

$$\sum_{i=1}^{n} \frac{a_i}{4^i} \quad \text{with } a_i \in \{1,3\}$$

(or zero) will ever appear as activation values in the corresponding processor. These numbers form a "Cantor-like set". For such numbers, affine operations are sufficient: the stack operation "push(I)" where $I \in \{0,1\}$, corresponds to $q_s \mapsto \frac{1}{4}q_s + \frac{I}{2} + \frac{1}{4}$, while "pop(I)" corresponds to $q_s \mapsto 4q_s - 2I - 1$. Reading a nonempty stack is done in constant time: if q_s encodes the stack value, then $\sigma(4q_s - 2) = 1$ if and only if the top symbol is "1", and $\sigma(4q_s - 2) = 0$ if the top element is "0". This encoding also makes the emptiness test easy: an empty stack is one in which $\sigma(4q_s) = 0$, and it is not empty precisely when this value is 1. *Note that a binary representation, rather than a base-4 representation, would not allow such simple stack operations.*

A critical aspect of the construction is to show that the whole design can be integrated without introducing high-order connections, that is, products. This is achieved basically by using negative values that act as "inhibitors" when fed into the activation function σ. As an illustration, consider just the "no-op" and "pop" actions, and assume a binary control signal c (which is computed from the current states and stacks) is given, so that the required effect, on a stack having value $q_s(t)$, is:

$$q_s(t+1) \;=\; \begin{cases} q_s(t) & \text{if } c = 0 \\ 4q_s(t) - 2I - 1 & \text{if } c = 1 \end{cases}$$

where I is the top element. The net guarantees that $q_s(t) \neq 0$ in the second case, that is, one does not attempt to pop an empty stack. Then one may use the update:

$$q_s(t+1) \;=\; \sigma[\sigma\,(4q_s(t) - 2I + 3c - 4) + \sigma(q_s(t) - c)]$$

(the outside σ is redundant, but is needed in order to obtain the desired form).

The rest of the paper provides the details of the construction. In the last section, we make some remarks about issues for further research, and we discuss relations to other work.

3 Preliminaries

As a departure point, we pick 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. We choose to represent the values in the stacks as fractions with denominators which are powers of four. An algebraic formalization is as follows.

3.1 Two-Stack Machines

Denote by $\mathcal{C}_4$ the "Cantor 4-set" consisting of all those rational numbers q which can be written in the form

$$q = \sum_{i=1}^{k} \frac{a_i}{4^i} \tag{2}$$

with $0 \leq k < \infty$ and each $a_i = 1$ or 3. (When $k = 0$, we interpret this sum as $q = 0$.)

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

$$\delta[a_1 \cdots a_k] := \sum_{i=1}^{k} \frac{2a_i + 1}{4^i} \in \mathcal{C}_4 \ . \tag{3}$$

Elements of $\mathcal{C}_4$ are precisely those of the form $\delta[\omega]$. (Note that the empty sequence gets mapped into 0.)

The instantaneous description of a two-stack machine, with a control unit of n states, can be represented by a 3-tuple

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

where s is the state of the control unit, and the stacks store the words ω_1 and ω_2, 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 $q \in \mathcal{C}_4$, 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" and $\tau[\cdot]$ as the "empty stack" operators, respectively.) 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 $\{0,1\}^2$. Note that in terms of the base-4 expansion (2), $\zeta[q] = 0$ when $a_1 = 1$ (or $q = 0$), and $\zeta[q] = 1$ when $a_1 = 3$.

Definition 3.1 A *two-stack machine* $\mathcal{M}$ is specified by a 6-tuple

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

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:

$$\begin{aligned} \theta_0 \ : \ S \times \{0,1\}^4 \ &\rightarrow \ S \\ \theta_i \ : \ S \times \{0,1\}^4 \ &\rightarrow \ \{(1,0,0), (\frac{1}{4}, 0, \frac{1}{4}), (\frac{1}{4}, 0, \frac{3}{4}), (4, -2, -1)\} \end{aligned}$$

for $i = 1, 2$.

(The function θ_0 computes the next state, while the functions θ_1 and θ_2 compute the next stack operation for stack1 and stack2, respectively. The actions depend only on the state of the control unit and the symbol being read from each stack. The values $(1,0,0), (\frac{1}{4},0,\frac{1}{4}), (\frac{1}{4},0,\frac{3}{4}), (4,-2,-1)$ of the θ_i should be interpreted as "no operation", "push0", "push1", and "pop", respectively.)

The set $\mathcal{X} := S \times \mathcal{C}_4 \times \mathcal{C}_4$ is called the *instantaneous description set* of $\mathcal{M}$, and the map

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

defined by

$$\begin{aligned} \mathcal{P}(s, q_1, q_2) \;\; := \;\; & [\theta_0(s, \zeta[q_1], \zeta[q_2], \tau[q_1], \tau[q_2]) \,, \\ & \theta_1^T(s, \zeta[q_1], \zeta[q_2], \tau[q_1], \tau[q_2]) \;\cdot\; (q_1, \zeta[q_1], 1) \,, \\ & \theta_2^T(s, \zeta[q_1], \zeta[q_2], \tau[q_1], \tau[q_2]) \;\cdot\; (q_2, \zeta[q_2], 1)] \,, \end{aligned}$$

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_1, θ_2 are such that $\theta_1(s, \zeta[q_1], \zeta[q_2], 0, \tau[q_2])$, $\theta_2(s, \zeta[q_1], \zeta[q_2], \tau[q_1], 0) \neq (4, -2, -1)$ for all s, q_1, q_2 (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 second stack, the machine reaches after k steps the halting state s_H, that is,

$$\mathcal{P}^k(s_I, \delta[\omega], 0) \;=\; (s_H, \delta[\omega_1], \delta[\omega_2]) \tag{4}$$

for some k, then the machine $\mathcal{M}$ is said to *halt on the input* ω. If ω is like this, let k be smallest possible so that $\mathcal{P}^k(s_I, \delta[\omega], 0)$ has the form (4). 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 $\phi_{\mathcal{M}} : \mathcal{C}_4 \to \mathcal{C}_4$ of two-stack machines as defined here; it is only necessary to identify words in $\{0,1\}^*$ and elements of $\mathcal{C}_4$ via the above encoding map δ. Our proof will then be based on simulating two-stack machines by processor nets.

3.2 A Restatement

It will be convenient to have a version of Theorem 1 that does not involve inputs but rather an encoding of the initial data into the initial state.

For a processor net without inputs (that is, with $b_1 = b_2 = 0$), 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.

Theorem 2 *Let $\phi : \{0,1\}^* \to \{0,1\}^*$ be any recursively computable partial function. 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 there is some k so that

$$\xi(\omega)_2^j \ = \ 0 \ , \ j = 0, \ldots, k-1 \ , \quad \xi(\omega)_2^k \ = \ 1 \ , \tag{5}$$

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

In section 4.3, we show how Theorem 1 can be derived from Theorem 2.

4 Proof of the Result

The proof of Theorem 1 is organized as follows. In section 4.1, we build a processor net without inputs as needed for Theorem 2. In section 4.2, we describe in graphical terms the architecture being used, accompanied by a detailed explanation. In section 4.3, we prove that a processor net without inputs is equivalent to a net with inputs.

Assume that a two-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 $s_H = 1$.

4.1 Mathematical Construction

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 $\mathbb{Q}^{s+2}$. Writing a vector in $\mathbb{Q}^{s+2}$ as $(x_1, \ldots, x_s, q_1, q_2)$, 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. Formally, define

$$\beta_{ij} \ : \ \{0,1\}^4 \to \{0,1\}, \ \text{ for } \ i \in \{1, \ldots, s\} \ , \ j \in \{0, \ldots, s\}$$

and

$$\gamma_{ij}^k \ : \ \{0,1\}^4 \to \{0,1\}, \ \text{ for } \ i = 1,2 \ , \ j \in \{0, \ldots, s\} \ , \ k = 1,2,3,4$$

as follows:

$$\beta_{ij}(a,b,d,e) = 1 \quad \Longleftrightarrow \quad \theta_0(j,a,b,d,e) = i$$

(intuitively: there is a transition from state j of the control part to state i iff the readings from the stacks are: top of stack1 is a, top of stack2 is b, the emptyness test on stack1 gives c, and the emptyness test on stack2 gives d),

$$\gamma_{ij}^1(a,b,d,e) = 1 \quad \Longleftrightarrow \quad \theta_i(j,a,b,d,e) = (1,0,0)$$

(if the control is in state j and the stack readings are a, b, c, d, then the stack i will not be changed),

$$\gamma_{ij}^2(a,b,d,e) = 1 \quad \Longleftrightarrow \quad \theta_i(j,a,b,d,e) = (\frac{1}{4}, 0, \frac{1}{4})$$

(if the control is in state j and the stack readings are a, b, c, d, then the operation *Push 0* will occur on stack i),

$$\gamma_{ij}^3(a,b,d,e) = 1 \quad \Longleftrightarrow \quad \theta_i(j,a,b,d,e) = (\frac{1}{4}, 0, \frac{3}{4})$$

(if the control is in state j and the stack readings are a, b, c, d, then the operation *Push 1* will occur on stack i),

$$\gamma_{ij}^4(a,b,d,e) = 1 \quad \Longleftrightarrow \quad \theta_i(j,a,b,d,e) = (4, -2, -1)$$

(if the control is in state j and the stack readings are a, b, c, d, then the operation *Pop* will occur on stack i).

Let $\widetilde{\mathcal{P}}$ be the map

$$\mathbb{Q}^{s+2} \to \mathbb{Q}^{s+2} \; : \; (x_1, \ldots, x_s, q_1, q_2) \mapsto (x_1^+, \ldots, x_s^+, q_1^+, q_2^+)$$

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

$$x_i^+ \;\; := \;\; \sigma \left[\sum_{j=0}^{s} \beta_{ij}(\zeta[q_1], \zeta[q_2], \tau[q_1], \tau[q_2])\, x_j \right] \tag{6}$$

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

$$\begin{aligned} q_i^+ := \quad \sigma \quad & \left[\left(\sum_{j=0}^{s} \gamma_{ij}^1(\zeta[q_1], \zeta[q_2], \tau[q_1], \tau[q_2])\, x_j \right) q_i \right. \\ + \quad & \left(\sum_{j=0}^{s} \gamma_{ij}^2(\zeta[q_1], \zeta[q_2], \tau[q_1], \tau[q_2])\, x_j \right) (\frac{1}{4} q_i + \frac{1}{4}) \\ + \quad & \left(\sum_{j=0}^{s} \gamma_{ij}^3(\zeta[q_1], \zeta[q_2], \tau[q_1], \tau[q_2])\, x_j \right) (\frac{1}{4} q_i + \frac{3}{4}) \\ + \quad & \left. \left(\sum_{j=0}^{s} \gamma_{ij}^4(\zeta[q_1], \zeta[q_2], \tau[q_1], \tau[q_2])\, x_j \right) (4q_i - 2\zeta[q_i] - 1) \right] \end{aligned} \tag{7}$$

for $i = 1, 2$. Note that the "σ" does not need to appear on the right hand side of both equations. It is used only for consistency in obtaining the desired result. Recall that $\zeta[q_i]$ is the map that provides the symbol at the top of stack i, and $\tau[q_i]$ performs the emptiness test on this stack.

Let $\pi : \mathcal{X} = S \times \mathcal{C}_4 \times \mathcal{C}_4 \to \mathbb{Q}^{s+2}$ be defined by

$$\pi(i, q_1, q_2) \; := \; (e_i, q_1, q_2) \; .$$

It follows immediately from the construction that

$$\widetilde{\mathcal{P}}(\pi(i, q_1, q_2)) \;\; = \;\; \pi(\mathcal{P}(i, q_1, q_2)) \tag{8}$$

for all $(i, q_1, q_2) \in \mathcal{X}$.

Applied inductively, Equation (8) implies that

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

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

$$(e_1, q_1, q_2)$$

(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 $\mathbb{Q}^{s+2}$). If such a state is reached, then q_1 is in $\mathcal{C}_4$ and its value is $\delta[\phi(\omega)]$.

• **Stage 2:** The second stage of the construction simulates the dynamics $\widetilde{\mathcal{P}}$ by a net. Subsection 4.2 provides a pictorial description of the following mathematical construction. We first need an easy technical fact.

Lemma 4.1 For each function $\beta : \{0,1\}^4 \to \{0,1\}$ there exist vectors

$$v_1, v_2, \ldots, v_{16} \in \mathbb{Q}^6$$

and scalars

$$c_1, c_2, \ldots, c_{16} \in \mathbb{Q}$$

such that, for each $a, b, d, e, x \in \{0,1\}$ and each $q \in [0,1]$,

$$\beta(a,b,d,e)x = \sum_{i=1}^{16} c_i \sigma(v_i \cdot \mu) \tag{9}$$

and

$$\beta(a,b,d,e)xq = \sigma\left(q + \sum_{i=1}^{16} c_i \sigma(v_i \cdot \mu) - 1\right) \;, \tag{10}$$

where we denote $\mu = (1, a, b, d, e, x)$ and "$\cdot$" = dot product in $\mathbb{Q}^6$.

Proof. Write β as a polynomial

$$\begin{aligned}\beta(a,b,d,e) = \quad & c_1 + c_2 a + c_3 b + c_4 d + c_5 e + c_6 ab + c_7 ad + c_8 ae + c_9 bd + \\ & c_{10} be + c_{11} de + c_{12} abd + c_{13} abe + c_{14} ade + c_{15} bde + c_{16} abde \;,\end{aligned}$$

expand the product $\beta(a,b,d,e)x$, 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

$$\begin{aligned}\beta(a,b,d,e)x \;&=\; c_1\sigma(x) + c_2\sigma(a + x - 1) + \cdots + c_{16}\sigma(a + b + d + e + x - 4) \\ &=\; \sum_{i=1}^{16} c_i \sigma(v_i \cdot \mu)\end{aligned} \tag{11}$$

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

$$\tau q \;=\; \sigma(q + \tau - 1) \tag{12}$$

(just check separately for $\tau = 0, 1$), so substituting formula (11) with $\tau = \beta(a, b, d, e)x$ into (12) gives the desired result. ∎

Apply Lemma 4.1 repeatedly, with the "β" of the Lemma corresponding to each of the β_{ij}'s and γ_{ij}^{l}'s, and using variously $q = q_i$, $q = (\frac{1}{4}q_i + \frac{1}{4})$, $q = (\frac{1}{4}q_i + \frac{3}{4})$, or $q = (4q_i - 2\zeta[q_i] - 1)$. Write also $\sigma(4q_i - 2)$ whenever $\zeta[q_i]$ appears, using that

$$\zeta[q] \;=\; \sigma(4q - 2) \;\text{ for each } q \in \mathcal{C}_4 \;,$$

and $\sigma(4q)$ whenever $\tau[q]$ appears. The result is that $\widetilde{\mathcal{P}}$ can be written as a composition

$$\widetilde{\mathcal{P}} \;=\; F_1 \circ F_2 \circ F_3 \circ F_4$$

of four "saturated-affine" maps, i.e. maps of the form $\vec{\sigma}(Ax + c)$,

$$F_4 : \mathbb{Q}^{s+2} \to \mathbb{Q}^{\mu}, \;\; F_3 : \mathbb{Q}^{\mu} \to \mathbb{Q}^{\nu}, \;\; F_2 : \mathbb{Q}^{\nu} \to \mathbb{Q}^{\eta}, \;\; F_1 : \mathbb{Q}^{\eta} \to \mathbb{Q}^{s+2}$$

for some positive integers μ, ν, η. (The argument to the function F_4, called below z_1, of dimension $(s + 2)$, represents the s x_i's of Equation (6) and the two t_i's of equation (7). Functions F_1, F_2, F_3 compute the transition function of the x_i's and t_i's in three stages.)

Consider the set of equations

$$\begin{aligned} z_1^+ &= F_1(z_2) \\ z_2^+ &= F_2(z_3) \\ z_3^+ &= F_3(z_4) \\ z_4^+ &= F_4(z_1) \end{aligned}$$

where the z_i's are vectors of sizes $s + 2$, η, ν, and μ respectively. This set of equations models the dynamics of a σ-processor net, with $N = s + 2 + \mu + \nu + \eta$ processors. For an initial state of type $z_1 = (e_0, \delta[\omega], 0)$ and $z_i = 0$, $i = 2, 3, 4$, it follows that at each time of the form $t = 4k$ the first block of coordinates, z_1, equals $\widetilde{\mathcal{P}}^k(e_0, \delta[\omega], 0)$.

All that is left is to add a mod-4 counter to impose the constraint that state values at times that are not divisible by 4 should be disregarded. The counter is implemented by adding a set of equations

$$\begin{aligned} y_1^+ &= \sigma(y_2) \\ y_2^+ &= \sigma(y_3) \\ y_3^+ &= \sigma(y_4) \\ y_4^+ &= \sigma(1 - y_2 - y_3 - y_4) \;. \end{aligned}$$

When starting with all $y_i(0) = 0$, it holds that $y_1(t) = 1$ if $t = 4k$ for some positive integer k, and is zero otherwise.

In terms of the complete system, $\phi(\omega)$ is defined if and only if there exists a time t such that, starting at the state

$$z_1 = (e_0, \delta[\omega], 0), \; z_i = 0, i = 2, 3, 4, \;\; y_i = 0, i = 1, 2, 3, 4 \;,$$

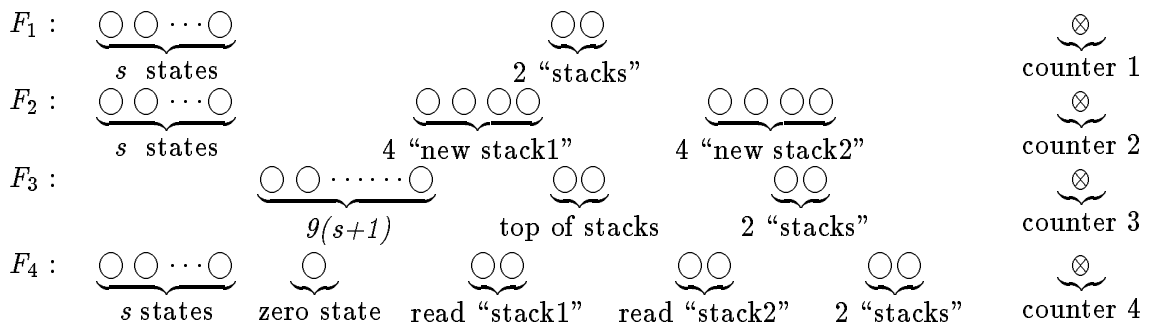
the first coordinate $z_{11}(t)$ of $z_1(t)$ equals 1 and also $y_1(t) = y_2(t - 1) = 1$. To force $z_{11}(t)$ not to output arbitrary values at times that are not divisible by 4, we modify it to

$$z_{11}^+ \;=\; \sigma(\ldots + l(y_2 - 1)) \;,$$

where "..." is as in the original update equation for z_{11}, and l is a positive constant bigger than the largest possible value of z_{11}. The net effect of this modification is that now $z_{11}(t) = 0$ for all t that are not multiples of 4, and for $t = 4k$ it equals 1 if the machine should halt and 0 otherwise. Reordering the coordinates so that the first stack ($s+1$'st coordinate of z_1) becomes the first coordinate, and the previous z_{11} (that represented the halting state s_H of the machine $\mathcal{M}$) becomes the second coordinate, Theorem 2 is proved. ∎

4.2 A Layout Of The Construction

The above construction can be represented pictorially as follows:



For now, ignore the rightmost element at each level, which is the counter. The remainder corresponds to the functions F_4, F_3, F_2, F_1, ordered from bottom to top. The processors are divided in levels, where the output of the ith level feeds into the $(i-1)$st level (and the output of the top level feeds back into the bottom). The processors are grouped in the picture according to their function.

The bottom layer, F_4, contains five groups of processors. The leftmost group of processors stores the values of the s states to pass to level F_3. The "zero state" processor outputs 1 or 0, outputting 1 if and only if all of the s processors in the first group are outputting 0. The "read stack i" group computes the top element $\zeta[q_i]$ of stack i, and $\tau[q_i] \in \{0,1\}$, which equals 0 if and only if stack i is empty. Each of the two processors in the last group stores an encoding of one of the two stacks.

Layer F_3 computes the 16 terms $\sigma(v_i \cdot \mu)$ that appear in equation (9) for each of the possible $s+1$ values of the vector x. Only $9(s+1)$ processors are needed, though, since there are only three possibilities for the ordered pair $(\zeta[q_i], \tau[q_i])$ for each of the two stacks. (Note that each such μ contains $\zeta[q_1]$, $\zeta[q_2]$, $\tau[q_1]$, $\tau[q_2]$, as well as $x_0, \ldots, x_s$, that were computed at the level F_4.) In this level we also pass to level F_2 the values $\zeta[q_1]$, $\zeta[q_2]$ and the encoding of the two stacks, from level F_4.

At level F_2 we compute all the new states as described in equation (6) (to be passed along without change to the top layer). Each of the four processors in the "new stack i" group computes one of the four main terms (rows) of equation (7) for one stack. For instance, for the fourth main term we compute an expression of the form:

$$\sigma(4q_i - 2\zeta[q_i] - 1 + \sum_{j=0}^{s} \sum_{i=1}^{16} c_{ij}\sigma(v_i \cdot \mu) - 1)$$

(obtained by applying equation (10)). Note that each of the terms q_i, $\zeta[q_i]$, $\sigma(v_i \cdot \mu)$ has been evaluated in the previous level.

Finally, at the top level, we copy the states from level F_3, except that the halting state x_1 is modified by counter 2. We also add the four main terms in each stack and apply σ to the result.

After reordering coordinates at the top level to be

$$t_1, x_0, \ldots, x_s, t_2 \ ,$$

the data processor and the halting processor are first and second, respectively, -as required to prove Theorem 2. Note that this construction results in values $\mu = s + 7$, $\nu = 9s + 13$, and $\eta = s + 8$.

4.3 Inputs and Ouputs

We now explain how to deduce Theorem 1 from Theorem 2. 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. As explained above, there are two input lines: $D = u_1$ carries the data, and $V = u_2$ validates it.

So assume we are given a net with no inputs

$$x^+ \ = \ \sigma(Ax + c) \tag{13}$$

as in the conclusion of Theorem 2. Suppose that we have already found a net

$$y^+ \ = \ \sigma(Fy + g u_1 + h u_2) \tag{14}$$

(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

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

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 (13) as follows. The new state consists of the pair (x, y), with y evolving according to (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):

$$\begin{aligned} x_1^+ &= \sigma(A_1 x + c_1 y_5 + y_4) \\ x_i^+ &= \sigma(A_i x + c_i y_5) \ , \quad i = 2, \ldots, n \ . \end{aligned}$$

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 (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 y_1, 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 x_1 and x_2, where $x_1(k) = \delta[w]$ for some k and $x_2(t) = 0$ for $t < k$, $x_2(k) = 1$ ($x_1(t) \in [0,1]$ for $t \neq k$, $x_2(t) \in [0,1]$ 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 $y = (z_{10}, z_9)$.

Remark 4.2 If one would also like to achieve a resetting of the whole network after completing the operation, it is possible to add the processor

$$z_{11}^+ = \sigma(z_{10}) ,$$

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

$$v_i^+ = \sigma(\ldots + z_{11} - z_{10}) , \; v \in \{x, y, z\} ,$$

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

5 Size Of The Net

Following the construction above, we can estimate the size of a network needed to compute a recursively computable partial function. Let ϕ be a recursively computable partial function, and let s be the number of states in the control unit of some two-stack machine computing ϕ. Then there exists a processor net $\mathcal{N}$ that computes ϕ, and consists of

$$\underbrace{[\mu+\nu+\eta+s+2+4]}_{\text{system without inputs}}+\underbrace{[5]}_{\text{input}}+\underbrace{[10+1]}_{\text{output}}=12s+50$$

processors.

• The Universal Net

A consequence of Theorm 1 is the existence of a universal processor net, which upon receiving an encoded description of a recursively computable partial function (in terms of a Turing machine) and an input string, would do what the encoded Turing machine would have done on the input string.

To approximate the number of processors in such a 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, [13]. Shannon showed in [16] how to change the number of letters in a Turing machine. Following his construction, we obtain a 2-letter 63-state 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 $12s + 50$, we conclude that there is a universal net with 1058 processors.

6 Remarks

Our result opens up a number of interesting problems. One obvious question deals with the use of other activation functions. It is not hard to see that various other choices are suitable. Using the "standard sigmoid" $1/(1+e^{-x})$ presents some technical difficulties, because rational numbers are harder to deal with. For instance, requiring an output sequence of exact "1's" is too stringent, but there are obvious modifications that can be done; we are currently studying this extension. On the other hand, an equation of the type $x^+ = \tau(Ax + bu + c)$, where τ is a hard threshold (Heaviside) function, can only simulate a finite automaton, as all states are essentially binary. These models are all closely related to the classical linear systems from control theory ([18]); see for instance [15] for some basic control-theoretic facts about systems that use identity, σ, and τ activations.

Note that 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 appears to become 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 [2], 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.

Our estimate of 1058 processors for a universal machine is conservative. It would certainly be interesting to have a better bound. 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 [1] applied to the control unit (here we used a very inefficient simulation).

Many other types of "machines" may be used for universality (see [18], especially Chapter 2, for general definitions of continuous machines). For instance, we can show that systems evolving according to equations

$$x^+ = x + \tau(Ax + bu + c)\ , \tag{15}$$

where τ takes the sign in each coordinate, again are universal in a precise sense. It is interesting to note also that equation (15) represents an Euler approximation of a differential equation; this suggests the existence of continuous-time simulations of Turing machines, quite different technically from the work on analog computation by Pour-El and others.

A different approach to continuous-valued models of computation is given in [3] and other papers by the same authors; in that context, our processor nets can be viewed as programs with loops in which linear operations and linear comparisons are allowed, but with an added restriction on branching that reflects the nature of the saturated response we use.

There are close relationships between our work and notions of circuit complexity. Certainly an "unfolding" of the dynamics up to time T provides a circuit (with σ-gates) that has size NT, where N in the number of processors. Lower bound results based on this are being currently investigated.

References

[1] Alon, N., A.K. Dewdney, and T.J. Ott, "Efficient simulation of finite automata by neural nets," *J. A.C.M.* 38 (1991): 495-514.

[2] Berstel, J. and C. Reutenauer, *Rational series and their Languages*, Springer-Verlag, Berlin, 1988.

[3] Blum, L., 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**(1989): 1-46.

[4] Franklin, S., and M. Garzon, "Neural computability," in *Progress In Neural Networks, Vol 1*)(O. M. Omidvar, ed.), Ablex, Norwood, NJ, 1990, pp. 128-144.

[5] Garzon, M., and S. Franklin, "Neural computability II," in *Proc. 3rd Int. Joint Conf. Neural Networks* (1989): I, 631-637.

[6] Giles, C.E., G.Z. Sun, H.H. Chen, Y.C. Lee, and D. Chen, "Higher order networks recurrent and grammatical inference," in *Advances in Neural Information Processing Systems 2* (D.S. Touretzky, ed.), Morgan Kaufmann, San Mateo, CA, 1990: 380-.

[7] Hartley, R., and H. Szu, "A comparison of the computational power of neural network models," in *Proc. IEEE Conf. Neural Networks* (1987): III, 17-22.

[8] John E. Hopcroft, and Jeffrey D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, 1979.

[9] Kleene, S.C., "Representation of events in nerve nets and finite automata," in Shannon, C.E., and J. McCarthy, eds., *Automata Studies*, Princeton Univ. Press 1956: 3-41.

[10] Maass, W., G. Schnitger, and E.D. Sontag, "On the computational power of sigmoid versus boolean threshold circuits," *Proc. of the 32nd Annual Symp. on Foundations of Computer Science*, 1991: 767-776.

[11] Marcus, C.M., and R.M. Westervelt, "Dynamics of iterated-map neural networks," *Phys. Rev. Ser. A* **40**(1989): 3355-3364.

[12] McCulloch, W.S., and W. Pitts, "A logical calculus of the ideas immanent in nervous activity," *Bull. Math. Biophys.* **5**(1943): 115-133.

[13] Minsky, M.L., *Computation: Finite and Infinite Machines*, Prentice Hall, Engelwood Cliffs, 1967.

[14] Pollack, J.B., *On Connectionist Models of Natural Language Processing*, Ph.D. Dissertation, Computer Science Dept, Univ. of Illinois, Urbana, 1987.

[15] Schwarzschild, R., and E.D. Sontag, "Algebraic theory of sign-linear systems," in *Proceedings of the Automatic Control Conference,* Boston, MA, June 1991: 799-804.

[16] Shannon, Claude E., "A Universal Turing machine with two internal states," in Shannon, C.E., and J. McCarthy, eds., *Automata Studies*, Princeton Univ. Press 1956: 157-165.

[17] Siegelmann, H.T., and E.D. Sontag, "Recognizing arbitrary languages with finite, first-order, neural networks with real-valued weights," in preparation.

[18] Sontag, E.D., *Mathematical Control Theory: Deterministic Finite Dimensional Systems*, Springer, New York, 1990.

[19] Sontag, E.D., "Remarks on interpolation and recognition using neural nets," in *Advances in Neural Information Processing Systems 3* (R.P. Lippmann, J. Moody, and D.S. Touretzky, eds), Morgan Kaufmann, San Mateo, CA, 1991, pp. 939-945.

[20] Sun, G.Z., H.-H. Chen, Y.-C. Lee, and C.L. Giles, "Turing equivalence of neural networks with second order connection weights," in *Int.Jt.Conf.Neural Nets*, Seattle, 1991:II,357-.