JOURNAL OF COMPUTER AND SYSTEM SCIENCES **50**, 132–150 (1995)

On the Computational Power of Neural Nets*

HAVA T. SIEGELMANN[†]

Department of Information Systems Engineering, Technion, Haifa 32000, Israel

AND

EDUARDO D. SONTAG[‡]

Department of Mathematics, Rutgers University, New Brunswick, New Jersey 08903

Received February 4, 1992; revised May 24, 1993

This paper deals with finite size networks which consist of interconnections of synchronously evolving processors. Each processor updates its state by applying a "sigmoidal" function to a linear combination of the previous states of all units. We prove that one may simulate all Turing machines by such nets. In particular, one can simulate any multi-stack Turing machine in real time, and there is a net made up of 886 processors which computes a universal partial-recursive function. Products (high order nets) are not required, contrary to what had been stated in the literature. Non-deterministic Turing machines can be simulated by non-deterministic rational nets, also in real time. The simulation result has many consequences regarding the decidability, or more generally the complexity, of questions about recursive nets. © 1995 Academic Press, Inc.

1. INTRODUCTION

We study the computational capabilities of recurrent first-order neural networks, or as we shall also say from now on, *processor nets*. Our model 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 nodes in the graph are called "neurons." Each neuron updates its activation value by applying a composition of a certain one-variable function with an affine combination (i.e., linear combination and a bias) of the activations of all neurons x_j, $j = 1, ..., N$, and the external inputs u_k, $k = 1, ..., M$, with rational valued coefficients—also called *weights*—(a_{ij}, b_{ij}, c_i). That is, each processor's state is updated by an equation of the type

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

In our results, the function σ is the simplest possible "sigmoid," namely the saturated-linear function

$$\sigma(x) := \begin{cases} 0 & \text{if } x < 0 \\ x & \text{if } 0 \leqslant x \leqslant 1 \\ 1 & \text{if } x > 1. \end{cases} \tag{2}$$

This function has appeared in many applications of neural nets (e.g., [Bat91, BV88, Lip87, ZZZ92]). We focus on this specific activation function because it makes theorems easier to prove, and also because, as it was proved in [SS94], a large family of sigmoidal type activation functions result in models which are not stronger computationally than this one.

The use of simoidal 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 McCulloh and Pitts [WM43] and Kleene [Kle56]), how to simulate finite automata by such nets.

As part of the description, we assume that there is singled out a subset of the N processors, say $x_{i_1}, ..., x_{i_p}$; 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 arbitrary reals in the range [0, 1], but later we constrain them to binary values only.)

* This research was supported in part by U.S. Air force Grant AFOSR-91-0343.

[†] E-mail: iehava@ie.technion.ac.il. Work done while at Dept. of Computer Science, Rutgers University.

[‡] E-mail: sontag@control.rutgers.edu

We state that one can simulate all (multi-stack) Turing machines by nets having rational weights. The rational numbers we consider (as weights, not those numbers arising as intermediate values of computation) are simple, small, and do not require much precision. Furthermore, this simulation can be done with no slow down in the computation. In particular, it is possible to give a net made up of 886 processors which computes a universal partial-recursive function. Non-deterministic Turing machines can be simulated by non-deterministic nets, also in real time.

We restrict to rational (rather than real) states and weights in order to preserve computability. It turns our that using *real valued* weights results in processor nets that can "calculate" super-Turing functions; see [SS94].

1.1. *Related Work*

Most of the previous work on recurrent neural networks has focused on networks of infinite size. As each neuron is itself a processor, an infinite net is effectively equivalent to an infinite automaton, which by itself has unbounded processing power. Therefore, infinite models are less challenging for the investigation of computational power, compared to our model which has only a finite number of neurons and is bounded computationally.

There has been previous work concerned with computability by finite networks, however, starting with the classical work of McCulloch and Pitts, cited above, on finite automata. 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 using multiplications, as opposed to just linear combinations (as in Eq. (1)). That is, the value of x_i is updated by means of a formula of the type

$$x_i(t+1) = \sigma\left(\sum_{(|\omega| + |\beta|) \leq k} a_{i,\omega,\beta} x^{\omega}(t)\, u^{\beta}(t) \right), \quad i = 1 \cdots N, \tag{3}$$

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

$$x^{\omega} = x_1^{\omega_1} \cdots x_N^{\omega_N}, \quad u^{\beta} = u_1^{\beta_1} \cdots u_M^{\beta_M}.$$

Pollack left as an open question whether high-order connections are really necessary in order to achieve universality, although he conjectured that they are. High order networks (cf. [CSSM89, Elm90, GMC+92, Pol87, SCLG91, WZ89]) have been used in applications. One motivation often cited for the use of high-order nets was Pollack's conjecture regarding their superior computational power. 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, 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*. 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 stack configurations in multi-stack Turing machines.

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, [Son92] for questions of approximation and function interpolation and [MSS91] for questions of Boolean circuit complexity.

The computability of an optical beam-tracing system consisting of a finite number of elements was discussed in [RTY90]. One of the models described in that paper is somewhat similar to ours, since it involves operations which are linear combinations of the parameters, with rational coefficients only, passing through the optical elements and having recursive computational power. In that model, however, three types of basic elements are involved, and the simulated Turing machine has a unary tape. Furthermore, the authors of that paper assume that the system can instantaneously differentiate between two numbers, no matter how close, which is not a logical assumption for our model.

1.2. *Consequences and Future Work*

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 [BR88], p. 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 threshold and linear functions, this gap in decidability is perhaps remarkable. Given the linear-time simulation bound, it is of course also possible to transfer

NP-completeness results into the same questions for nets (with rational coefficients). Another consequence of our results (when using the halting problem) is that the problem of determining whether a dynamical system of the particular form

$$x(t+1) = \sigma(Ax(t) + c)$$

ever reaches an equilibrium point, from a given initial state, is effectively undecidable. Such models have been proposed in the neural and analog computation literature for dealing with content-addressable retrieval and optimization, notably in the work of Hopfield (see, e.g., [HT85]. In associative memories, for instance, the initial state is taken as the "input pattern" and the final state, as a class representative. All convergence results currently known assume special structure of the A matrix—for instance, that it is symmetric—but our undecidability conclusion shows that such results will not be possible for general systems of the above type.

Another corollary is that higher order networks are computationally equivalent, up to a polynomial time, to first-order networks. (A finite network with rational weights is finitely describable and can be efficiently simulated by a Turing machine, which in turn can be simulated by first-order networks.)

Many other types of "machines" may be used for universality (see [Son 90, especially Chap. 2] for general definitions of continuous machines). For instance, we can show that systems evolving according to equations $x(t+1) = x(t) + \tau(Ax(t) + bu(t) + c)$, where τ takes the sign in each coordinate, again are universal in a precise sense. It is interesting to note that this equation 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 [PE74] and others. A different approach to continuous-valued models of computation is given in [BSS89] 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.

The rest of this paper is organized as follows. Section 2 provides the exact definitions of the model and states the results. It is followed by Section 3, which highlights the main proof. Details of the proof are provided in Sections 4 to 7. In section 8, we describe the universal network, and we end in Section 9, where we briefly describe the non-deterministic version of networks.

2. THE MODEL AND MAIN RESULTS

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. Given a system of equations such as (1), 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 $t \geqslant 1$, 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.

We want to define what we mean by a net computing a function

$$\phi: \{0, 1\}^+ \mapsto \{0, 1\}^+,$$

where $\{0, 1\}^+$ denotes the free semigroup of binary strings (excluding the empty string). To do so, we must first define what we mean by 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 that

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

for each t. We always take the initial state $x(1)$ to be zero and to be an equilibrium state. We assume that there are two output processors, which also take the role of data and validation lines and are denoted by "G" and "H," respectively. The output of the network is then given by

$$y(t) = (H(t), G(t)) \in \{0, 1\}^2.$$

(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.)

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

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

One defines the *state at time t*, for each integer $t \geqslant 1$, as the value obtained by recursively solving the equations:

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

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

$$(q_1, ..., q_N) \mapsto (\sigma(q_1), ..., \sigma(q_N))$$

by

$$\boldsymbol{\sigma}_N : \mathbb{Q}^N \to \mathbb{Q}^N$$

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 somtimes 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

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

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, [MW89]).

It is obvious that—with zero, or in general a 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 are recursively computed. In other words, maps for which there is a multi-stack Turing machine $\mathcal{M}$ so that, when a word $\omega \in \{0, 1\}^+$ is written initially on the input/output stack, $\mathcal{M}$ halts on ω if and only if $\phi(\omega)$ is defined, and $\phi(\omega)$, in that case, is the content of that stack when the machine halts.

For each word

$$\omega = \omega_1 \cdots \omega_k \in \{0, 1\}^+$$

with

$$\phi(\omega) = \beta_1 \cdots \beta_l \in \{0, 1\}^+ \text{ or undefined}$$

and each $r \in \mathbb{N}$ (l is called the *length of the response/output* and r is called the *response time*, the time it takes to provide the first output bit), we encode the input and output as follows. Input is encoded as

$$u_\omega(t) = (V_\omega(t), D_\omega(t)), \quad t = 1, ...,$$

where

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

and

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

Output is encoded as

$$y_{\omega, r}(t) = (G_{\omega, r}(t), H_{\omega, r}(t)), \quad t = 1, ...,$$

where

$$G_{\omega, r}(t) = \begin{cases} 1, & \text{if } t = r, ..., (r + l - 1), \\ 0, & \text{otherwise,} \end{cases}$$

if the output $\phi(\omega)$ is defined, and is 0 when $\phi(\omega)$ is not defined, and finally

$$H_{\omega, r}(t) = \begin{cases} \beta_{t-r+1}, & \text{if } t = r, ..., (r + l - 1), \\ 0, & \text{otherwise,} \end{cases}$$

if the output $\phi(w)$ is defined, and is 0 when $\phi(\omega)$ is not defined.

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 p-stack Turing machine $\mathcal{M}$ (of one input stack and several working stacks) computes $\phi(\omega)$ in time $T(\omega)$, then one may take $r(\omega) = T(\omega) + O(|\omega|)$.

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 $N = 886$ processors are sufficient for computing such a universal partial recursive function.

2.1. *Restatement: No I/O*

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. (This would be analogous, for

Turing machines, to an encoding of the input into an input stack rather than having it arrive as an external input stream.)

For a 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 kth 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 $\phi\colon \{0,1\}^+ \to \{0,1\}^+$ 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

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

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

Theorem 2. *Let M be a p-stack Turing machine computing $\phi\colon \{0,1\}^+ \to \{0,1\}^+$ in time T. Then there exists a processor net $\mathcal{N}$ without inputs so that properties* (a) *and* (b) *below hold. In both cases, for each $\omega \in \{0,1\}^+$, we consider the corresponding initial state*

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

(a) *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 and computed in time T, than there exists $\mathcal{T} = O(T)$ so that*

$$\xi(\omega)_2^j = 0, \qquad j = 0, ..., \mathcal{T} - 1,$$
$$\xi(\omega)_2^{\mathcal{T}} = 1,$$

and $\xi(\omega)_1^{\mathcal{T}} = \delta[\phi(\omega)]$. (This is a linear time simulation.)

(b) *Furthermore, if one substitutes δ in Eq.* (4) *by a new map δ_p as*

$$\delta[a_1 \cdots a_k]_p := \sum_{i=1}^{k} \frac{10p^2 - 1 + 4p(a_i - 1)}{(10p^2)^i}, \tag{5}$$

then the construction can be done in such a way that $\mathcal{T} = T$. (This is a real time simulation.)

The next few sections include the proofs of both theorems. We start by proving Theorem 2, and then obtain Theorem 1 as a corollary. As the details of the proof of Theorem 2 are very technical, we start by sketching the main steps of the proof in Section 3. The proof itself is organized into several steps. We first show how to construct a network $\mathcal{N}$ that simulates a given multi-stack Turing machine M in time $\mathcal{T} = 4T$ (T is the computation time of the Turing machine). This is done in Sections 4 and 5. In Section 6, we modify the construction into a network that simulates a given multi-stack Turing machine with no slow down in the computation. This is done in three steps and is based on allowing for large negative numbers to act as inhibitors.

After Theorem 2 is proved, we show in Section 7 how to add inputs and outputs to a processor net without input/output, thus obtaining Theorem 1 as its corollary. This ends the proofs.

3. HIGHLIGHTS OF THE PROOF

This section is aimed at highlighting the main part of the proof of Theorem 2. We start with a 2-stack machine; this model is obviously equivalent to a 1-tape (standard) Turing machine [HU79]. Formally, a 2-stack machine consists of a finite control and two binary stacks, unbounded in length. Each stack is accessed by a read–write head, which is located at the top element of the stack. At the beginning of the computation, the binary input sequence is written on stack$_1$. The machine at every step reads the top element of each stack as a symbol $\alpha \in \{0, 1, \#\}$ (where # means that the stack is empty), checks the state of the control ($s \in \{1, 2, ..., |S|\}$), and executes the following operations:

1. For each stack, one of the following manipulations is made:

   (a) Popping the top element.

   (b) Pushing an extra element on the top of the stack (either 0 or 1).

   (c) No change in stack.

2. Changing the state of the control.

When the control reaches a special state, called the "halting state," the machine stops. Its output is defined as the binary sequence on stack$_1$. Thus, the I/O map or function, computed by a 2-stack machine, is defined by the binary sequences on stack 1 before and after the computation.

3.1. *Encoding the Stacks*

Assume that we were to encode a binary stream $\omega = \omega_1 \omega_2 \cdots \omega_n$ into the number

$$\sum_{i=1}^{n} \frac{\omega_i}{2^i}.$$

Such a value could be held in a neuron, since it ranges in [0, 1]. However, there are a few problems with this simple encoding. First, one cannot differentiate between the strings "β" and "$\beta \cdot 0$," where "$\cdot$" denotes the concatenation operator. Second even when assuming that each binary string ends with 1, the continuity of the activation function σ makes it impossible to retrieve the most significant bit (in radix 2) of a string in a constant amount of time. (For example, the values 0.100000000000 and 0.011111111111111 are almost indistinguishable by a net.) In summary, given streams of binary input signals, one does not want them to be encoded on a continuous range, but rather to have gaps between valid encodings of the strings. Such gaps would enable a quick decoding of the number by means of an operation requiring only finite precision or, equivalently, reading the most significant bit in some representation in a constant amount of time.

On the other hand, if we choose some set of "numbers with gaps" to encode the different binary strings, we have to assure that various manipulations on these numbers during the computation leave the stack encoding in the same set of "numbers with gaps."

As a solution, the encoding of stacks is chosen to be as follows. Read the binary elements in each stack from top to bottom as a binary string $\omega = \omega_1 \omega_2 \cdots \omega_n$. We encode this string into the number

$$\sum_{i=1}^{n} \frac{2\omega_i + 1}{4^i}.$$

This number ranges in $[0, 1)$, but not every value in $[0, 1)$ appears. If the string starts with the value 1, then the associated number has a value of at least $\frac{3}{4}$, and if it starts with 0, the value is in the range $[\frac{1}{4}, \frac{1}{2})$. The empty string is encoded as the value 0. The next element in the stack restricts the possible value further.

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 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 most significant digit in the base-4 representation.

3.2. *Stack Operations*

We next demonstrate the usefulness of our encoding of the stacks:

1. *Reading the top.* Assume that a stack has the value $\omega = 1011$; that is, it is encoded by the number $q = 0.3133_4$. As discussed above, the value of q is at least $\frac{3}{4}$ when the top of the stack is 1, and at most $\frac{1}{2}$ otherwise. The linear operation

$$4q - 2$$

transfers this range to at least 1 when the top element is 1, and at most 0 otherwise. Thus, the function $\text{top}(q)$,

$$\text{top}(q) = \sigma(4q - 2),$$

provides the value of the top element.

2. *Push.* Pushing 0 onto the stack $\omega = 1011$ changes the value to $\omega = 01011$. In terms of the encoding, $q = 0.3133_4$ is transferred to $q = 0.13133_4$. That is, the suffix remains the same and the new element is entered in the most significant location. This is easily done by the operation

$$\frac{q}{4} + \frac{1}{4}$$

(which is equivalent to $\sigma(q/4 + \frac{1}{4})$, given that $q \in [0, 1)$.) Pushing the value 1 onto the stack can be implemented by $q/4 + \frac{3}{4}$.

3. *Pop stack.* Popping a stack transfers $\omega = 1011$ to 011, or the encoding from $q = 0.3133_4$ to 0.133_4. When the top element is known, the operation

$$4q - (2\,\text{top}(q) + 1)$$

(or equivalently $\sigma(4q - (2\,\text{top}(q) + 1))$) has the effect of popping the stack.

4. *Non-empty stack.* The predicate *non-empty* indicates whether the stack ω is empty or not, which means in terms of the encoding, whether $q = 0$ or $q \geq 0.1_4$. This can be decided by the operation

$$\sigma(4q).$$

3.3. *General Construction of the Network*

From the above discussion, we construct a network architecture which has three neurons per stack. One neuron holds the stack encoding (q), one holds $\text{top}(q)$, and one indicates whether the stack is empty. In addition, the architecture has a few neurons which represent the finite control (this is the old McCulloh and Pitts result [WM43]) and a set of neurons which take their input both from the stack-reading neurons and from the finite control neurons and "compute" the next step.

3.4. *P Stack Machines*

It is known that p-stack machines ($p \geq 2$) are polynomially equivalent to 2-stack machines [HU79]. However, simulating a p-stack machine by a 2-stack machine involves a super-linear slowdown in the computation. It is interesting to note that in our neural network, we are able to simulate any p-stack machine (for any p) in real time. That is, for us, there is no slowdown in passing from one model to the other, and they are linear-time equivalent rather than polynomially equivalent. To emphasize this attractive property, and as it does not reduce the simplicity of the proof, we provide our simulation in terms of p-stack, rather than 2-stack, machines.

3.5. *Real Time Simulation*

The above-suggested encoding and the use of the lemma result in a Turing machine simulation that requires four steps of the network for each step of the Turing machine. To achieve a "step per step" simulation, a more sophisticated encoding is required, which relies upon a Cantor set with a specific size of gaps. Then, we use large negative numbers as inhibitors, and attain in this manner the desired simulation in real time. We turn now to the formal proof.

4. 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.

4.1. *p-Stack Machines*

Denote by $\mathscr{C}$ 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 $0 \leqslant k < \infty$ and each $a_i = 1$ or 3. (When $k = 0$, we interpret this sum as $q = 0$.) Elements of $\mathscr{C}$ are precisely those of the form $\delta[\omega]$, where δ is as in Eq. (4).

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

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

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

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

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

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

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. 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 $\{0, 1\}^2$.

Defintion 4.1. A *p-stack machine* $\mathscr{M}$ is specified by a $(p+4)$-tuple

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

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:

$$\theta_0 \colon S \times \{0, 1\}^{2p} \to S$$

$$\theta_i \colon S \times \{0, 1\}^{2p} \to \{(1, 0, 0), (\tfrac{1}{4}, 0, \tfrac{1}{4}), (\tfrac{1}{4}, 0, \tfrac{3}{4}), (4, -2, -1)\} \quad \text{for } i = 1, ..., p.$$

(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

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

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

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

$$\mathscr{P} \colon \mathscr{X} \to \mathscr{X}$$

defined by

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

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

Let $\omega \in \{0, 1\}^+$ be arbitrary. If there exists 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,

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

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

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

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

$$\phi_{\mathscr{M}}: \{0, 1\}^+ \to \{0, 1\}^+,$$

the *i/o map* of $\mathscr{M}$.

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

5. NETWORK WITH FOUR LEVELS: CONSTRUCTION

Assume that a p-stack machine $\mathscr{M}$ 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 $S := \{0, ..., s\}$, 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 $\mathscr{M}$ with a certain dynamical system over $\mathbb{Q}^{s+p}$. Writing a vector in $\mathbb{Q}^{s+p}$ as

$$(x_1, ..., x_s, q_1, ..., q_p),$$

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

$$e_i = (0, ..., 0, 1, 0, ..., 0)$$

(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

$$\beta_{ij}: \{0, 1\}^{2p} \to \{0, 1\} \tag{6}$$

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

$$\gamma_{ij}^k: \{0, 1\}^{2p} \to \{0, 1\} \tag{7}$$

for $i \in \{1, ..., p\}, j \in \{0, ..., s\}$, $k = 1, 2, 3, 4$ as

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

(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). y_{ij}^a is zero except in the following cases:

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

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

$$\gamma_{ij}^2(a_1, a_2, ..., a_p, b_1, b_2, ..., b_p) = 1$$
$$\Leftrightarrow \theta_i(j, a_1, a_2, ..., a_p, b_1, b_2, ..., b_p) = (\tfrac{1}{4}, 0, \tfrac{1}{4})$$

(if the control is in state j and the stack readings are a_1, a_2, ..., a_p, b_1, b_2, ..., b_p, then operation *Push*0 will occur on stack i);

$$\gamma_{ij}^3(a_1, a_2, ..., a_p, b_1, b_2, ..., b_p) = 1$$
$$\Leftrightarrow \theta_i(j, a_1, a_2, ..., a_p, b_1, b_2, ..., b_p) = (\tfrac{1}{4}, 0, \tfrac{3}{4})$$

(if the control is in state j and the stack readings are $a_1, a_2, ..., a_p, b_1, b_2, ..., b_p$, then operation *Push*1 will occur on stack i);

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

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

Let $\mathscr{P}$ be the map $\mathbb{Q}^{s+p} \to \mathbb{Q}^{s+p}$,

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

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

$$x_i^+ := \sum_{j=0}^{s} \beta_{ij}(a_1, ..., a_p, b_1, ..., b_p)\, x_j \tag{8}$$

for $i = 1, ..., s$ and

$$q_i^+ := \left(\sum_{j=0}^{s} \gamma_{ij}^1(a_1, ..., a_p, b_1, ..., b_p)\, x_j\right) q_i \qquad (9.1)$$

$$+ \left(\sum_{j=0}^{s} \gamma_{ij}^2(a_1, ..., a_p, b_1, ..., b_p)\, x_j\right) \times \left(\frac{1}{4} q_i + \frac{1}{4}\right) \qquad (9.2)$$

$$+ \left(\sum_{j=0}^{s} \gamma_{ij}^3(a_1, ..., a_p, b_1, ..., b_p)\, x_j\right) \times \left(\frac{1}{4} q_i + \frac{3}{4}\right) \qquad (9.3)$$

$$+ \left(\sum_{j=0}^{s} \gamma_{ij}^4(a_1, ..., a_p, b_1, ..., b_p)\, x_j\right) \times (4q_i - 2\zeta[q_i] - 1) \qquad (9.4)$$

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

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

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

It follows immediately from the construction that

$$\tilde{\mathscr{P}}(\pi(i, q_1, ..., q_p)) = \pi(\mathscr{P}(i, q_1, ..., q_p))$$

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

Applied inductively, the above implies that

$$\tilde{\mathscr{P}}^k(e_0, \delta[\omega], 0, ..., 0) = \pi(\mathscr{P}^k(0, \delta[\omega], 0, ..., 0))$$

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

$$(e_1, q_1, ..., q_p)$$

(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 $\mathscr{C}$ and its value is $\delta[\phi(\omega)]$.

• *Stage* 2. The second stage of the construction simulation the dynamics $\tilde{\mathscr{P}}$ by a net. We first need an easy technical fact.

Lemma 5.1. *Let $t \in \mathbb{N}$. For each function β: $\{0, 1\}^t \to \{0, 1\}$ there exist vectors*

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

and scalars

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

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

$$\beta(d_1, d_2, ..., d_t)x = \sum_{r=1}^{2^t} c_r \sigma(v_r \cdot \mu) \qquad (10)$$

and

$$\beta(d_1, d_2, ..., d_t)\, xq = \sigma\left(q + \sum_{r=1}^{2^t} c_r \sigma(v_r \cdot \mu) - 1\right), \qquad (11)$$

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

Proof. Write β as a polynomial

$$\beta(d_1, d_2, ..., d_t) = c_1 + c_2 d_1 + \cdots + c_{t+1} d_t + c_{t+2} d_1 d_2 + \cdots + c_{2^t} d_1 d_2 \cdots d_t, \qquad (12)$$

expand the product $\beta(d_1, d_2, ..., d_t)x$, and use that for any sequence $l_1, ..., l_k$ 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

$$\beta(d_1, d_2, ..., d_t)x = c_1\sigma(x) + c_2\sigma(d_1 + x - 1) + \cdots + c_{2^t}\sigma \times (d_1 + d_2 + \cdots + d_t + x - t) = \sum_{r=1}^{2^t} c_r \sigma(v_r \cdot \mu)$$

for suitable c_r's and v_r'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 $\tau = \beta(d_1, d_2, ..., d_t)x$ gives the desired result. ∎

Remark 5.2. The above construction overestimates the amount of neurons necessary. 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 for just 3^p terms in the summation, rather than 2^{2p}. We illustrate this phenomenon with the simple case of $p = 2$, that is, the case of two-stack machines. Here, when

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

one can easily verify that from the following nine elements

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

one can obtain—using a affine combinations only— the value of the remaining seven elements

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

Hence, a sum of the type in Eq. (10) requires in this case only nine (3^p) elements rather than sixteen (2^{2p}).

We now return to the proof of the theorem. Apply Lemma 5.1 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 $\zeta[q] = \sigma(4q - 2)$ for each $q \in \mathscr{C}$), and $\sigma(4q)$ whenever $\tau[q]$ appears. The result is that $\mathscr{P}$ can be written as a composition

$$\mathscr{P} = F_1 \circ F_2 \circ F_3 \circ F_4$$

of four "saturated-affine" maps, i.e., maps of the form $\boldsymbol{\sigma}(Ax + c)$: $F_4: \mathbb{Q}^{s+p} \to \mathbb{Q}^\mu$, $F_3: \mathbb{Q}^\mu \to \mathbb{Q}^\nu$, $F_2: \mathbb{Q}^\nu \to \mathbb{Q}^\eta$, $F_1: \mathbb{Q}^\eta \to \mathbb{Q}^{s+p}$, for some positive integers μ, ν, η. (The argument to the function F_4, called below z_1, of dimension $(s+p)$, represents the s x_i's of Eq. (8) and the two qi's of Eqs. (9). The functions F_1, F_2, F_3 compute the transition function of the x_i's and q_i's in three stages.)

Consider the following set of equations, where from now on we omit time arguments and use the superscript $^+$ to indicate a one-step time increment:

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

where the z_i's are vectors of sizes $s+p$, η, ν, and μ, respectively. This set of equations models the dynamics of a σ-processor net, with

$$N = s + p + \mu + \nu + \eta$$

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 $\mathscr{P}^k(e_0, \delta[\omega], 0)$.

All that is left is to add a mode-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 that $y_1(t) = y_2(t-1) = 1$. To force $z_{11}(t)$ not to output arbitrary values at times that are not divisible by 4, we modify it to

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

where "$\cdots$" 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 ($(s+1)$th coordinate of z_1) becomes the first coordinate, and the previous z_{11} (that represented the halting state s_H of the machine $\mathscr{M}$) becomes the second coordinate, Theorem 2(a) is proved. ∎

5.1. *A Layout of the Construction*

The above construction can be represented pictorially as in Fig. 1.

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 ouput of the ith level feeds into the $(i-1)$th 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 $3 + p$ 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 p processors in the last group stores an encoding of one of the p stacks.

Layer F_3 computes the 2^{2p} terms $\sigma(v_r \cdot \mu)$ that appear in Eq. (10) for each of the possible $s+1$ values of the vector x.

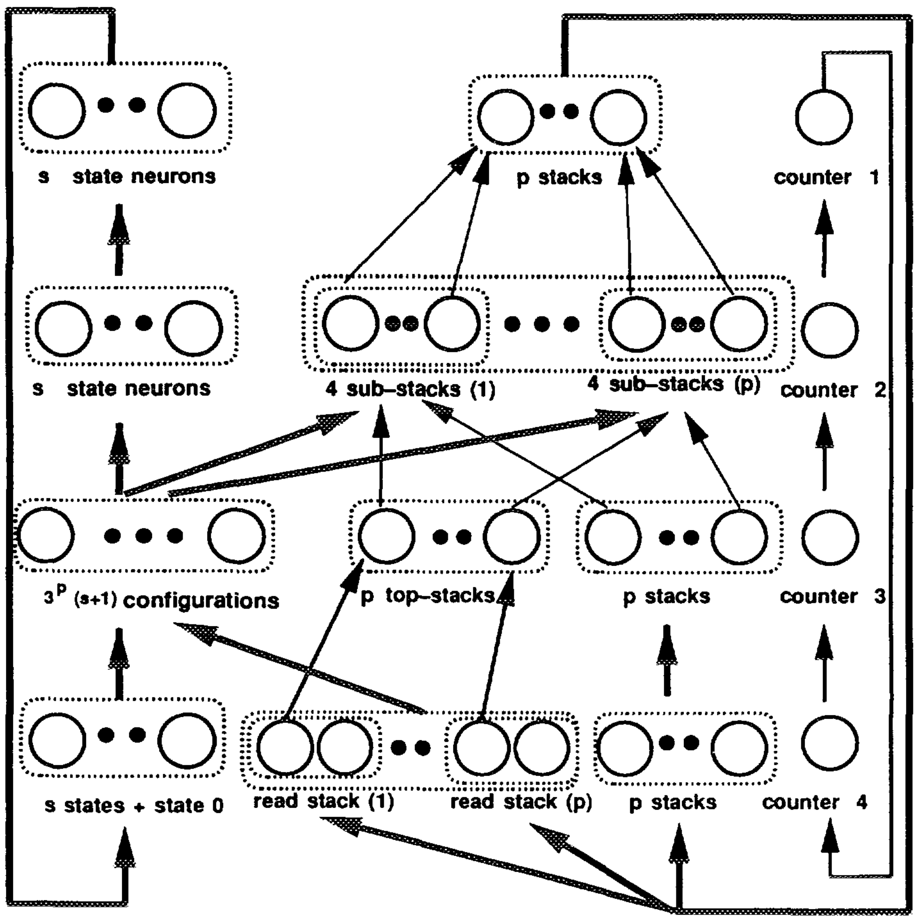

FIG. 1. The universal network.

Only $3^p(s+1)$ processors are needed, although, since there are only three possibilities for the ordered pair $(\zeta[q_i], \tau[q_i])$ for each of the two stacks; this was explained in Remark 5.2. (Note that each such μ contains $\zeta[q_1], ..., \zeta[q_p], \tau[q_1], ..., \tau[q_p]$, as well as $x_0, ..., x_s$, 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 p stacks, from level F_4. As a matter of fact, $3^p s+1$ neurons may substitute the $s^p(s+1)$, as the 0 state requires one neuron only.

At level F_2 we compute all the new states as described in Eq. (8) (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 Eqs. (9) for one stack. For instance, for the fourth main term we compute an expression of the form:

$$\sigma\left(4q_i - 2\zeta[q_i] - 1 + \sum_{j=0}^{s} \sum_{r=1}^{2^{2p}} c_{rj}\sigma(v_r \cdot \mu) - 1\right)$$

(obtained by applying Eq. (11)). Note that each of the terms q_i, $\zeta[q_i]$, $\sigma(v_r \cdot \mu)$ 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

$$t_1, x_0, ..., x_s, t_2, ..., t_p,$$

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 $\mu = s + 3p + 1$, $\nu = 3^p(s+1) + 2p$, and $\eta = s + 4p$.

If desired, one could modify this construction so that all activation values are reset to zero when the halting processor takes the value "1." This is discussed more thoroughly in Remark 7.1.

6. REAL TIME SIMULATION

Here, we refine the simulation of Section 5 to obtain the claimed real-time simulation. Thus, this section provides a proof of Theorem 2(b). This part is both less intuitive and less crucial for the main result.

We start in Subsection 6.1 by modifying the construction given in Section 5 in order to obtain a "two-level" neural network. At this point, we obtained a slow-down of a factor of two in the simulation. In Subsection 6.2, 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 Subsection 6.3, we show how to modify the last network into a standard one with one level only, thus achieving the desired real-time simulation of Turing machines.

In Subsection 6.2, we substitute the 4-Cantor set representation with a somewhat more complex Cantor set representation (to be explained there), which has larger gaps between consecutive admissible ranges of empty stacks, stacks with 0 top, and stacks with 1 top. These gaps, along with the use of large negative numbers as inhibitors, will enable us to speed up the simulation.

6.1. *Computing in Two Layers*

We can rewrite the dynamics of the stack q_i from Eqs. (9) as the sum of four components,

$$q_i = \sum_{j=1}^{4} q_{ij}, \tag{13}$$

where q_{ij} represents the row $(9 \cdot j)$ in Eqs. (9). We name these q_{ij} as the *sub-stack elements* of stack i. That is, q_{i1} may differ from 0 only if the last update of stack i was "no-operation." Similarly, the components q_{i2}, q_{i3}, q_{i4} may differ from 0 only if the last update of the ith stack were "push0," "push1," or "pop," respectively. We also define t_{ij} to be the *sub-top* of the sub-stack element q_{ij}, and e_{ij} to be the *sub-nonempty* test of the same sub-stack element. The top of stack i can be computed by

$$t_i = \sum_{j=1}^{4} t_{ij} \tag{14}$$

and

$$e_i = \sum_{j=1}^{4} e_{ij}. \tag{15}$$

As three out of the four sub-stack elements $\{q_{i1}, q_{i2}, q_{i3}, q_{i4}\}$ of each stack $i = 1, ..., p$ are 0, and the fourth has the value of the stack q_i, it is also the case that three out of four sub-top elements of t_i (and sub-nonempty elements of e_i) are 0, and the fourth one stores the value of the top (nonempty predicate) of the relevant stack.

Using Eq. (8), we can express the dynamics of the state constrol as

$$x_i^+ = \sum_{j=0}^{s} \beta_{ij}(t_{11}, ..., t_{p4}, e_{11}, ..., e_{p4}) x_j \tag{16}$$

for $i = 1, ..., s$ and the dynamics of the sub-stack elements as

$$q_{i1}^+ = \sigma\left(q_i + \sum_{k=0}^{s} \gamma_{ik}^1(t_{11}, ..., e_{p4})\, x_k - 1\right)$$
$$q_{i2}^+ = \sigma\left(\tfrac{1}{4}q_i + \tfrac{1}{4} + \sum_{k=0}^{s} \gamma_{ik}^2(t_{11}, ..., e_{p4})\, x_k - 1\right)$$
$$q_{ik}^+ = \sigma\left(\tfrac{1}{4}q_i + \tfrac{3}{4} + \sum_{k=0}^{s} \gamma_{ik}^3(t_{11}, ..., e_{p4})\, x_k - 1\right)$$
$$q_{i2}^+ = \sigma\left(4q_i - 2\zeta[q_i] - 1 + \sum_{k=0}^{s} \gamma_{ik}^4(t_{11}, ..., e_{p4})\, x_k - 1\right)$$

for all stacks q_i, $i = 1, ..., p$.

We introduce the notation

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

and summarize the above sub-stack dynamic equations by

$$q_{ij}^+ = \sigma\left(\text{next-}q_{ij} + \sum_{k=0}^{s} \gamma_{ik}^j(\cdot)\, x_k - 1\right). \tag{17}$$

Similarly, the sub-top and sub-nonempty are updated by

$$\begin{aligned} t_{ij}^+ &= \sigma\left(4\left[\text{next-}q_{ij} + \sum_{k=0}^{s} \gamma_{ik}^j(\cdot)\, x_k - 1\right] - 2\right), \\ e_{ij}^+ &= \sigma\left(4\left[\text{next-}q_{ij} + \sum_{k=0}^{s} \gamma_{ik}^j(\cdot)\, x_k - 1\right]\right) \end{aligned} \tag{18}$$

for all $i = 1, ..., p, j = 1, ..., 4$.

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 the sub-elements q_{ij}, t_{ij}, e_{ij}, $j = 1, ..., 4$, $i = 1, ..., p$. This will enable us a simulation of one step of a Turing machine by a "two level" rather than a "four level" network, as was suggested in the previous section.

By Lemma 5.1 and Eqs. (14), (15), the functions β_{ij} and γ_{ik}^j of Eqs. (16)–(18) can be written as the combination

$$\sum_{k=0}^{s} \sum_{r=1}^{3^p} c_r^a \sigma(v_r^a \cdot \tilde{\mu}), \tag{19}$$

where

$$\tilde{\mu} = \left(1, \sum_{j=1}^{4} t_{1j}, ..., \sum_{j=1}^{4} t_{pj}, \sum_{j=1}^{4} e_{1j}, ..., \sum_{j=1}^{4} e_{pj}, x\right)$$

c_r^a are scalar constants, v_r^a are vector constants, and a represents the multi-indices ij for β and ijk for γ. Thus, all the updated equations of

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

can be written as

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

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

The main layer consists of the sub-elements q_{ij}, t_{ij}, e_{ij} and the states x_r. In the hidden layer, we compute all elements $\sigma(\cdots)$ required by Lemma 5.1 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 and t_i to compute next-q_{ij}.

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

$$\mathscr{P} = F_1 \circ F_2$$

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

In summary:

- The main layer consists of:

1. s neurons x_k, $k = 1, ..., s$, that represent the state of the system unarily.

2. For each stack i, $i = 1, ..., p$, we have
   (a) four neurons $q_{ij}^1 \equiv q_{ij}$, $j = 1, 2, 3, 4$,
   (b) four neurons $t_{ij}^1 \equiv t_{ij}$, $j = 1, 2, 3, 4$
   (c) four neurons $e_{ij}^1 \equiv e_{ij}$, $j = 1, 2, 3, 4$.

- 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, $i = 1, ..., p$ we have
   (a) a neuron $q_i^2 \equiv q_i$,
   (b) a neuron $t_i^2 \equiv t_i$.

6.2. *Removing the Sigmoid from the Main Level*

Here, we proceed by shrinking the network and showing how to construct a network equivalent to the one above, in which neurons in the main level compute linear combinations only (and apply no σ function to it). In the following construction, we introduce a set of "noisy sub-stack" elements $\{\tilde{q}_{i1}, \tilde{q}_{i2}, \tilde{q}_{i3}, \tilde{q}_{i4}\}$ for each stack $i = 1, ..., p$. 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 sub-top" and "noisy sub-nonempty" functions applied to the noisy sub-stack elements may also produce values outside 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 now abandon the 4-Cantor set representation, using instead a slightly different Cantor set representation: If the network is to simulate a p-stack machine, then our encoding is the $10p^2$-Cantor set.

To motivate our new Cantor representation, we illustrate it first with the special case $p = 2$. Here, the encoding is a 40-Cantor set representation. We encode the bit 0 by $\varepsilon_0 = 31$ and 1 by $e_1 = 39$. We interpret the resulting sequence as a number in base 40. For example, the stack $w = 1101$ is encoded by

$$q = .(39)(31)(39)(39)_{40}.$$

In addition, we allow the empty stack to be represented by any non-positive value of q, rather than 0 only. This is where the sigmoids of the main level are being saved. The possible ranges of a value q representing the stacks are thus:

$[-\infty, 0]$	empty stack
$[\frac{31}{40}, \frac{32}{40}]$	top of stack is 0
$[\frac{39}{40}, 1]$	top of stack is 1.

Stack operations include push0, which is implemented by the operation $q/40 + \frac{31}{40}$; push1, which is implemented as $q/40 + \frac{39}{40}$; and pop which is implemented by $40q - t$, where t is the top element of the stack. We can compute the top and non-empty predicates by

$$\text{top}(q) := 5(40q - 38),$$
$$\text{nonempty}(q) := (40q - 30).$$

The ranges of the top values are $[5, 10]$, $[-35, -30]$, and $[-\infty, -190]$ for top $=1$, top $=0$, and empty stack, respectively; and the ranges of the non-empty predicate are $[9, 10]$, $[1, 2]$, and $[\infty, -30]$, stated with the same order. In particular, top is interpreted as being 1 when the function top(q) is in the range $[5, 10]$ and 0 for $[\infty, -30]$, while the non-empty predicate is taken as 1 in the range $[1, 10]$ and 0 in the range $[-\infty, -30]$. The gaps between the positive and negative ranges are large; in particular, the domain values which represent the value 0 in both predicates are at least 3 times larger than the domain values which represent the value 1. The special encoding was planned to have this property. In the case of p stack machines, we need an encoding for which the values of the negative domain are at least $(2p-1)$ times larger than the values of the positive domain.

For the general p-stack machine, we choose the base $b=10p^2$. Denote $c=2p+1$, $b=10p^2$, $\varepsilon_1=(10p^2-1)$, $\varepsilon_0=(10p^2-4p-1)$. That is, 0 is encoded by ε_0, 1 is encoded by ε_1, and the resulting sequence is interpreted in base b. The role of c will be explained below. The reading functions "noisy sub-top" and "noisy sub-nonempty" corresponding to the noisy sub-stack element $v \in \{\tilde{q}_{ij} \mid i=1, ..., p,\ j=1, ..., 4\}$ are defined as

$$\text{N-top}(v) := c(bv-(\varepsilon_1-1)) \tag{20}$$

$$\text{N-nonempty}(v) := bv-(\varepsilon_0-1). \tag{21}$$

We denote $\tilde{t}_{ij}=\text{N-top}(\tilde{q}_{ij})$ and $\tilde{e}_{ij}=\text{N-nonempty}(\tilde{q}_{ij})$ for $i=1, ..., p$, $j=1, ..., 4$.

We summarize dynamic equations of the noisy elements by

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

$$\tilde{t}_{ij}^{+}=\sigma\left(c\left[b\left(\text{next-}\tilde{q}_{ij}+\sum_{k=0}^{s}\gamma_{ik}^{j}(\cdot)\,x_k-1\right)-(\varepsilon_1-1)\right]\right),$$

$$e_{ij}^{+}=\sigma\left(b\left(\text{next-}\tilde{q}_{ij}+\sum_{k=0}^{s}\gamma_{ik}^{j}(\cdot)\,x_k-1\right)-(\varepsilon_0-1)\right), \tag{22}$$

where

$$\text{next-}q_{ij}=\begin{cases} q_i, & \text{if } j=1\\ \dfrac{1}{b}q_i+\dfrac{\varepsilon_0}{b}, & \text{if } j=2\\ \dfrac{1}{b}q_i+\dfrac{\varepsilon_1}{b}, & \text{if } j=3\\ bq_i-(\varepsilon_1-\varepsilon_0)\,\zeta[q_i]-\varepsilon_0, & \text{if } j=4,\end{cases}$$

and $\zeta[q_i]$ is the true binary value of the top of stack i.

The ranges of values of the noisy sub-top and sub-non-empty functions are

N-top(v)

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

N-nonempty(v)

$$\in\begin{cases}[4p+1, 4p+2] & \text{when the top is "1"}\\ [1, 2] & \text{when the top is "0"}\\ [-\infty,\ -10p^2+4p+2] & \text{for an empty stack;}\end{cases}$$

(Note that the values of $\sigma(\text{N-nonempty}(v))$ and $\sigma(\text{N-top}(v))$ provide the exact binary top and nonempty predicates.) The union of these ranges is

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

The parameter c is chosen so that to assure large gaps in the range U.

Property. For all $p \geqslant 2$, 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, Eq. (13) is not valid any more. That is, the elements $\tilde{q}_{ij}$, $j=1, ..., 4$, cannot be combined linearly to provide the real value of the stack q_i. This is also the case with the top and nonempty predicates (see Eq. (14), (15)).

In Lemma 5.1, we proved that for any Boolean function of the type β: $\{0,1\}^t \mapsto \{0,1\}$ and $x \in \{0,1\}$, one may express

$$\beta(d_1, ..., d_t)x=\sum_{r=1}^{2^t} c_r\sigma(v_r\cdot\mu)$$

for $\mu=(1, d_1, ..., d_t, x)$, constants c_r and constant vectors v_r. This was applicable for the functions β_{rj} and γ_{ik}^{j} in Eqs. (16)–(18). Next, we prove that using the noisy sub-top $\tilde{t}_{ij}$ and noisy sub-nonempty $\tilde{e}_{ij}$ elements—rather than the binary sub-top $(\sigma(\tilde{t}_{ij}))$ and sub-nonempty $(\sigma(\tilde{e}_{ij}))$ ones—one may still compute the functions β_{rj} and γ_{ik}^{j} using one hidden layer only.

It is not true for any function β: $U^t \mapsto \{0,1\}$ and $x \in \{0,1\}$ that $\beta(\cdot)x$ is computable in one hidden layer network; this is true in particular cases only, including ours.

DEFINITION 6.1. A function $\beta(v_1, ..., v_t)$ is said to be *sign-invariant* if

$$v_i' = \text{sign}(v_i)\ \forall i = 1, ..., t \Rightarrow \beta(v_1 \cdots v_t) = \beta(v_1', ..., v_t').$$

LEMMA 6.2. *For each $t, r \in \mathbb{N}$, let U_t be the range*

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

and let

$$S_{r,t} = \{d \mid d = (d_1^{(1)}, ..., d_1^{(r)}, d_2^{(1)}, ..., d_2^{(r)}, ..., d_t^{(1)}, ..., d_t^{(r)})$$
$$\in R_t^{rt}, \text{ and } \forall i = 1, ..., t \text{ at most one of } d_i^1, ..., d_i^r \text{ is positive}\}.$$

We denote by I the set of multi-indices $(i_1, ..., i_t)$, with each $i_j \in \{0, 1, ..., r\}$. For each function β: $S_{r,t} \to \{0, 1\}$ 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 $(d_1^{(1)}, ..., d_t^{(r)}) \in S_{r,t}$) and any $x \in \{0, 1\}$, we can write

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

where

$$\mu_i = \mu_{(i_1, ..., i_t)} = (1, d_1^{(i_1)}, d_2^{(i_2)}, ..., d_t^{(i_t)}, x)$$
$$\text{and we are defining } d_i^0 = 0. \qquad (23)$$

Here the size of I is $|I| = (r+1)^t$, and "$\cdot$" 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

$$\beta(d_1^{(1)}, d_1^{(2)}, ..., d_t^{(r)}) = \beta(\sigma(d_1^{(1)}), \sigma(d_1^{(2)}), ..., \sigma(d_t^{(r)})).$$

Thus, β can be viewed as a Boolean function on $\{0, 1\}^{rt} \mapsto \{0, 1\}$, and we can express it as a polynomial (see Eq. (12)):

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

(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 $l_1, ..., l_k$ of ($k \leq t$) elements in U_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 U_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 $\beta(d_1^1, ..., d_t^r)x$, using the above observation and the fact $x = \sigma(x)$. This gives that

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

for suitable c_i's and v_i's, where μ_i is defined as in (23). ∎

Remark 6.3. 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.

We conclude that the noisy sub-stack elements $\tilde{q}_{ij}$ as well as the next state control, are computable in the one hidden layer network from $\tilde{t}_{ij}$ and $\tilde{e}_{ij}$. We next provide the exact network architecture.

Network Description. The network consists of two levels. The main level consists of both s state neurons and noisy sub-stack neurons accompanied by sub-stack noisy-readings neurons: $\tilde{q}_{ij}^1, \tilde{t}_{ij}^1, \tilde{e}_{ij}^1$, $i = 1, ..., p$, $j = 1, ..., 4$— representing respectively noisy sub-stack-elements, noisy sub-top elements, and noisy sub-nonempty elements.

For simplicity, we first provide the update equation of the network which simulates a 2-stack machine, and later we generalize it to p-stack machines. Recall that for a 2-stack machine the encoding was a 40-Cantor set representation with $\varepsilon_0 = 31$ and $\varepsilon_1 = 39$. We use the same notations for the functions γ_{ij} as in Eq. (9); let q_i represent the ith stack, and t_i represents the true (i.e., clean) value of the top of the stack. The noisy sub-stack $\tilde{q}_{i1}$ updates for no-operation as in Eq. (9.1),

$$\tilde{q}_{i1}^{1+} = q_i + \sum_{k=0}^{s} \gamma_{ik}^j x_k - 1;$$

the noisy sub-stack $\tilde{q}_{i2}$ updates for push0 as in Eq. (9.2),

$$\tilde{q}^1_{i2} = \frac{q_i}{40} + \frac{31}{40} + \sum_{k=0}^{s} \gamma^j_{ik} x_k - 1;$$

the noisy sub-stack $\tilde{q}_{i3}$ updates for push1 as in Eq. (9.3),

$$\tilde{q}^{1+}_{i3} = \frac{q_i}{40} + \frac{39}{40} + \sum_{k=0}^{s} \gamma^j_{ik} x_k - 1;$$

and the noisy sub-stack $\tilde{q}_{i4}$ updates for pop as in Eq. (9.4),

$$\tilde{q}^{1+}_{i4} = 40q_i - 8t_i - 31 + \sum_{k=0}^{s} \gamma^j_{ik} x_k - 1.$$

For general p-stack machines, the bases are $10p^2$ and the update equations are given by

$$\tilde{q}^{1+}_{ij} = \text{next-}\tilde{q}_{ij} + \sum_{k=0}^{s} \gamma^j_{ik} x_k - 1, \tag{24}$$

$$\tilde{t}^{1+}_{ij} = (2p+1)\left[10p^2\left(\text{next-}\tilde{q}_{ij} + \sum_{k=0}^{s} \gamma^j_{ik} x_k - 1\right) - (10p^2 - 2)\right], \tag{25}$$

$$\tilde{e}^{1+}_{ij} = 10p^2\left(\text{next-}\tilde{q}_{ij} + \sum_{k=0}^{s} \gamma^j_{ik} x_k - 1\right) - (10p^2 - 4p - 2), \tag{26}$$

where

$$\text{next-}\tilde{q}_{ij} = \begin{cases} q_i, & \text{if } j=1 \text{ (not updated)} \\ \dfrac{1}{10p^2} q_i + \dfrac{10p^2 - 4p - 1}{10p^2}, & \text{if } j=2 \text{ (push0)} \\ \dfrac{1}{10p^2} q_i + \dfrac{10p^2 - 1}{10p^2}, & \text{if } j=3 \text{ (push1)} \\ 10p^2 q_i - 4pt_i - (10p^2 - 4p - 1), & \text{if } j=4 \text{ (pop)}, \end{cases}$$

and q_i and t_i are the exact values of the stacks and top elements. Using Lemma 6.2, all the expressions of the type $\beta(\cdot)x$ and $\gamma(\cdot)x$ can be written as linear combinations of terms like σ (linear combinations of $\tilde{t}^1_{ij}$, $\tilde{e}^1_{ij}$). These $\tilde{q}_{ij}$, $\tilde{t}_{ij}$, and $\tilde{e}_{ij}$ constitute the main level.

The hidden layer consists of both up to $(5^{2p}s + 2)$ configuration detectors neurons (as proved in Lemma 6.2) and the stack and top neurons,

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

which are updated by the equations

$$q^{2+}_{ij} := \sigma(\tilde{q}^1_{ij}), \quad \left[q_i = \sum_{j=1}^{4} q^2_{ij}\right],$$

$$t^{2+}_{ij} := \sigma(\tilde{t}^1_{ij}), i = 1, ..., p, j = 1, ..., 4, \quad \left[t_i = \sum_{j=1}^{4} t^2_{ij}\right].$$

6.3. *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. Note that as the net consists of one level only, no "counters" are required. This ends the proof of Theorem 2. ▌

7. INPUTS AND OUTPUTS

We now explain how to deduce Theorem 1 from Theorem 2. (We present details in the linear-time simulation case only. The real-time case is entirely analogous, but the notations become more complicated.) 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. In this section we concentrate on the function δ, thus proving that Theorem 2(a) implies a linear time simulation of a Turing machine via a network with input and output. A similar proof, when applied to Theorem 2(b) (while substituting δ by δ_p) implies Theorem 1.

So assume that we are given a net with no inputs,

$$x^+ = \sigma(Ax + c), \tag{27}$$

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

$$y^+ = \sigma(Fy + gu_1 + hu_2) \tag{28}$$

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

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

that is,

$$y_4(t) = \begin{cases} \delta[\omega], & \text{if } t = |\omega| + 2, \\ 0, & \text{otherwise;} \end{cases}$$

$$y_5(t) = \begin{cases} 0, & \text{if } t \leqslant |\omega| + 2, \\ 1, & \text{otherwise.} \end{cases}$$

Once this is done, modify the original net (27) as follows. The new state consists of the pair (x, y), with y evolving according to (28) 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):

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

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

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 $t \geqslant |\omega| + 3$. The system (28) can be constructed as

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

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 $x_1(k) = \delta[\omega]$ 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

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

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

One can verify that this can be done by

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

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

Remark 7.1. If one would also like to achieve a resetting of the whole network after completing the operation, it is possible to add the processor

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

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

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

where "$\cdots$" is the formerly defined operation of the processor.

8. UNIVERSAL NETWORK

The number of neurons required to simulate a Turing machine consisting of s states and p stacks, with a slowdown of a factor of two in the computation, is

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

To estimate the number of processors required for 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 four letters and seven control states [Min 67]. 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. 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.)

9. NON-DETERMINISTIC COMPUTATION

A *non-deterministic processor net* is a modification of a determinstic one, obtained by incorporating a *guess* input line ($\mathcal{G}$) in addition to the validation and data lines. Hence, the dynamics map of the network is now

$$\mathcal{F}\colon \mathbb{Q}^N \times \{0, 1\}^3 \rightarrow \mathbb{Q}^N.$$

Similarly to Definition 2.1, we define a non-deterministic processor network as follows.

DEFINITION 9.1. A *non-deterministic σ-processor net $\mathcal{N}$* with two binary inputs and a guess line is a dynamical system having a dynamics map of the form

$$\mathcal{F}(x, u, g) = \sigma(Ax + b_1 u_1 + b_2 u_2 + b_3 g + c),$$

for some matrix $A \in \mathbb{Q}^{N \times N}$ and four vectors $b_1, b_2, b_3, c \in \mathbb{Q}^N$.

A formal non-deterministic network is one that adheres to an input–output encoding convention similar to the one for deterministic systems, as in Section 2. For each pair of words $\omega, \gamma \in \{0, 1\}^+$, the input to the network is encoded as

$$v_{\omega, \gamma}(t) = (V_\omega(t), D_\omega(t), \mathcal{G}_\gamma(t)), \quad t = 1, ...,$$

where the input lines V_ω and D_ω are the same as described in Section 2 and $\mathcal{G}\gamma$ is defined as

$$\mathcal{G}_\gamma(t) = \begin{cases} \gamma_t, & \text{if } t = 1, ..., |\gamma|, \\ 0, & \text{otherwise.} \end{cases}$$

The output ecoding is the same as the one used for the deterministic networks.

We restrict our attention to formal non-deterministic networks that compute binary output values only, that is, $\phi_{\mathcal{N}}(\omega, \gamma) \in \{0, 1\}$, where $\phi_{\mathcal{N}}(\omega, \gamma)$ is the function computed by a formal non-deterministic network when the input is $\omega \in \{0, 1\}^+$ and the guess is $\gamma \in \{0, 1\}^+$. The language L computed by a non-deterministic formal network in time B is

$$L = \{\omega \in \{0, 1\}^+ \mid \exists \text{ a } guess\ \gamma_\omega, \phi_{\mathcal{N}}(\omega, \gamma_\omega) = 1, |\gamma_\omega| \leqslant T_{\mathcal{N}}(\omega) \leqslant B(|\omega|)\}.$$

Note that the input ω and the guess γ_ω do not have to have the same length; the only requirement regarding their synchronization is that $\omega(1)$ and $\gamma(1)$ appear as an input to the network simultaneously. The function $T_{\mathcal{N}}$ is the amount of time required to compute the response to a given input ω; and its bound, the function B, is called the computation time. The length of the guess, $|\gamma_\omega|$, is bounded by the function $T_{\mathcal{N}}$.

When restricted to the case of language recognition rather than general function computing, Theorems 1 and 2 can be restated for the non-deterministic model in which $\mathcal{N}$ is a non-deterministic processor net and $\mathcal{M}$ is a non-deterministic Turing machine. The proofs are similar and are omitted.

ACKNOWLEDGMENT

We thank Robert Solovay for his useful comments during the early stages of this research.

REFERENCES

[ADO91] N. Alon, A. K. Dewdney, and T. J. Ott, Efficient simulation of finite automata by neural nets, *J. Assoc. Comput. Mach.* **38**, No. 2 (1991), 495–514.

[Bat91] R. Batruni, A multilayer neural network with piecewise-linear structure and back-propagation learning, *IEEE Trans. Neural Networks* **2** (1991), 395–403.

[BR88] J. Berstel and C. Reutenauer, "Rational Series and their Languages," Springer-Verlag, Berlin, 1988.

[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. Amer. Math. Soc.* **21** (1989), 1–46.

[BV88] J. R. Brown, M. M. Garber, and S. F. Vanable, Artificial neural network on a SIMD architecture, *in* "Proceedings, 2nd Symposium on the Frontier of Massively Parallel Computation, Fairfax, 1988," pp. 43–47.

[CSSM89] A. Cleeremans, D. Servan-Schreiber, and J. McClelland, Finite state automata and simple recurrent networks, *Neural Comput.* **1**, No. 3 (1989), 372.

[Elm90] J. L. Elman, Finding structure in time, *Cog. Sci.* **14** (1990), 179–211.

[FG90] S. Franklin and M. Garzon, Neural computability, *in* "Progress in Neural Networks" (O. M. Omidvar, Ed.), pp. 128–144, Ablex, Norwood, NJ, 1990.

[GF89] M. Garzon and S. Franklin, Neural computability, *in* "Proceedings, 3rd Int. Joint Conf. Neural Networks, 1989," Vol. 2, pp. 631–637.

[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 Comput.* **4**, No. 3 (1992), 393–405.

[HS87] R. Hartley and H. Szu, A comparison of the computational power of neural network models, *in* "Proceedings, IEEE Conf. Neural Networks, 1987," pp. 17–22.

[HT85] J. J. Hopfield and D. W. Tank, Neural computation of decisions in optimization problems, *Biol. Cybern.* **52** (1985), 141–152.

[HU79] J. E. Hopcroft and J. D. Ullman, "Introduction to Automata Theory, Languages, and Computation," Addison–Wesley, Reading, MA, 1979.

[Kle56] S. C. Kleene, Representation of events in nerve nets and finite automata, *in* "Automata Studies" (C. E. Shannon and

J. McCarthy, Ed.), pp. 3–41, Princeton Univ. Press, Princeton, NJ, 1956.

[Lip87] R. P. Lippmann, An introduction to computing with neural nets, *IEEE Acoust. Speech Signal Process. Mag.* April (1987), 4–22.

[Min67] M. L. Minsky, "Computation: Finite and Infinite Machines," Prentice–Hall, Engelwood Cliffs, NJ 1967.

[MSS91] W. Maass, G. Schnitger, and E. D. Sontag, On the computational power of sigmoid versus boolean threshold circuits, *in* "Proceedings, 32nd Annu. Sympos. on Foundations of Computer Science, 1991," pp. 767–776.

[MW89] C. M. Marcus and R. M. Westervelt, Dynamics of iterated-map neural networks, *Phys. Rev. Ser. A* **40** (1989), 3355–3364.

[PE74] M. Pour-El, Abstract computability and its relation to the general purpose analog computer, *Trans. Amer. Math Soc.* **290** (1974), 1–29.

[Pol87] J. B. Pollack, "On Connectionist Models of Natural Language Processing," Ph.D. thesis, Computer Science Dept, Univ. of Illinois, Urbana, 1987.

[RTY90] J. H. Reif, J. D. Tygar, and A. Yoshid, The computability and complexity of optical beam tracing, *in* "Proceedings, 31st Annu. Sympos. on Foundations of Computer Science, 1990," pp. 106–144.

[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 International Joint Conference on Neural Networks, IEEE, 1991."

[Sha56] C. E. Shannon, A universal turing machine with two internal states, *in* "Automata Studies" (C. E. Shannon and J. McCarthy, Eds.), pp. 156–165, Princeton Univ. Press, Princeton, NJ, 1956.

[Son90] E. D. Sontag, "Mathematical Control Theory: Deterministic Finite Dimensional Systems," Springer-Verlag, New York, 1990.

[Son92] E. D. Sontag, Feedforward nets for interpolation and classification, *J. Comput. System. Sci.* **45** (1992), 20–48.

[SS94] H. T. Siegelmann and E. D. Sontag, Analog computation via neural networks, *Theoret. Comput. Sci.* **131** (1994), 331–360.

[WM43] W. Pitts and W. S. McCulloch, A logical calculus of ideas immanent in nervous activity, *Bull. Math. Biophys.* **5** (1943), 115–133.

[Wol91] D. Wolpert, "A Computationally Universal Field Computer Which Is Purely Linear," Technical Report LA-UR-91-2937, Los Alamos National Laboratory, 1991.

[WZ89] R. J. Williams and D. Zipser, A learning algorithm for continually running fully recurrent neural networks, *Neural Comput.* **1**, No. 2 (1989).

[ZZZ92] B. Zhang, L. Zhang, and H. Zhang, A quantitative analysis of the behavior of the pln network, *Neural Networks* **5** (1992), 639–661.