On The Computational Power Of Neural Nets *

Hava T. Siegelmann
Department of Computer Science
Rutgers University, New Brunswick, NJ 08903
E-mail: siegelma@yoko.rutgers.edu

Eduardo D. Sontag
Department of Mathematics
Rutgers University, New Brunswick, NJ 08903
E-mail: sontag@control.rutgers.edu

Abstract

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.

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, $j = 1, \ldots, N$ and the external inputs u_k, $k = 1, \ldots M$, 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

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

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

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

This function has appeared in many applications of neural nets (e.g. [2, 4, 15, 30]). We focus on this specific activation function because it makes theorems easier to prove, and also because, as it

*This research was supported in part by US Air Force Grant AFOSR-91-0343.

was proved in [24], a large family of sigmoidal type activation functions result in models which are not stronger computationally than this one.

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 ([18], [14]), 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 $x_{i_1}, \ldots, x_{i_p}$; these are the p *output processors*, and they are used to communicate the outputs of the network to the environment. (Generally, the output values are arbitrary reals in the range $[0,1]$, but later we constrain them to binary values only.)

We state that one can simulate all (multi-stack) Turing Machines by nets having rational weights. The rational numbers we consider 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 out that using *real valued* weights results in processor nets that can "calculate" super-Turing functions, see [24].

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, such models of infinite power are less interesting for the investigation of computational power, compared to our model which has only a finite number of neurons.

There has been previous work concerned with computability by finite networks, however, starting with the classical work of McCulloch and Pitts cited above on finite automata. Another related result was due to Pollack[20]. Pollack argued that a certain recurrent net model, which he called a "neuring machine," is universal. The model in [20] 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 Equation 1). That is, the value of x_i is updated by means of a formula of the following type:

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

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

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

Pollack left as an open question whether high-order connections are really necessary in order to achieve universality, though he conjectured that they are. High order networks (i.e. [6, 7, 11, 20, 28]) 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 ([12]) and by Franklin and Garzon ([8] and [9]), 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 [29], 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 [29] 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 [26] for questions of approximation and function interpolation, and [16] for questions of Boolean circuit complexity.

The computability of an optical beam tracing system consisting of a finite number of elements was discussed in [21]. 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 [3], 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. 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 if a dynamical system

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

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 (work of Hopfield and others: here the initial state is taken as the "input pattern" and the final state as a class representative).

Another corollary is that higher order networks are computationally equivalent, up to a polynomial time, to first order networks.

Many other types of "machines" may be used for universality (see [25], especially Chapter 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 and others. A different approach to continuous-valued models of computation is given in [5] 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

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

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

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

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

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*

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

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

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

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

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

by

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

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 [17]).

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 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 $r \in \mathbb{N}$, we encode the input and output as follows. Input is encoded as

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

where

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

and

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

Output is encoded as

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

where

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

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

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

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

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

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

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

Theorem 2 *Let M be a p-stack Turing Machine computing $\phi : \{0,1\}^+ \to \{0,1\}^+$ 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*

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

(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

$$\begin{aligned} \xi(\omega)_2^j &= 0\,, \quad j = 0, \ldots, \mathcal{T}-1, \\ \xi(\omega)_2^{\mathcal{T}} &= 1\,, \end{aligned}$$

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

(b)) Furthermore, if one substitutes δ in equation (4) by a new map δ_p as follows:

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

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 in 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 known to be equivalent to a 1-tape (standard) Turing Machine. 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 ($s \in \{1, 2, \ldots |S|\}$), 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 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 can not differentiate between the strings "β" and "$\beta \cdot 0$", where '$\cdot$' denotes the concatenation operator. Secondly, 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 .100000000000 and .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 encoding 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 started with the value 1, then the associated number has a value of at least $\frac{3}{4}$, and if it started with 0, the value is in the range $[\frac{1}{4}, \frac{1}{2})$. The empty string is encoded into 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 a stack has the value $\omega = 1011$, that is, it is encoded by the number $q = .3133_4$. 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 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 into $\omega = 01011$. In terms of the encoding, $q = .3133_4$ is transferred into $q = .13133_4$. That is, the suffix remains the same and the new element is entered into the most significant location. This is easily done by the operation

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

(which is equivalent to $\sigma(\frac{q}{4} + \frac{1}{4})$ given that $q \in [0, 1)$.)

3. **Pop Stack:** Popping a stack, transfers $\omega = 1011$ into 011, or the encoding from $q = .3133_4$ to $.133_4$. 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.

One can verify that $\sigma(4q)$ provides the "non-empty" predicate, and that pushing the value 1 can be implemented by $\frac{q}{4} + \frac{3}{4}$.

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 top(q), and one measures whether the stack is empty. In addition, the architecture has a few neurons which represent the finite control (this is the old McCulloch and Pitts result [18]) and a set of neurons which take their input both from the stack reading neurons and from the finite control neurons and "computes' the next step.

3.4 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 complicated encoding is required, which relies upon a Cantor set with a specific size of gaps. Then, we use large negative numbers as inhibitors in a more sophisticated manner and attain in this manner the desired simulation in real time.

We turn now to the formal proof. The proof is given for the general case of p-stack machines, not just the case $p = 2$ which would be sufficient for establishing Turing universality. This, more general case, is more interesting when establishing time bounds on computation.

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

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

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

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

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

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

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

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

and:

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

We think of $\zeta[\cdot]$ as the "top of stack," as in terms of the base-4 expansion, $\zeta[q] = 0$ when $a_1 = 1$ (or $q = 0$), and $\zeta[q] = 1$ when $a_1 = 3$. We interpret $\tau[\cdot]$ as the "nonempty stack" operator. It can never happen that $\zeta[q] = 1$ while $\tau[q] = 0$; hence the pair $(\zeta[q], \tau[q])$ can have only three possible values in $\{0,1\}^2$.

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

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

where S is a finite set, s_I and s_H are elements of S called the *initial* and *halting states*, respectively, and the θ_i's are maps as follows:

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

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

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

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

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

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

defined by

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

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

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

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

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

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

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

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

the *i/o map* of $\mathcal{M}$. □

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

5 Network with Four Levels: Construction

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

We build the net in two stages.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

$$\begin{aligned} q_i^+ \quad := \sigma \quad & \left(\sum_{j=0}^{s} \gamma_{ij}^1(a_1, \ldots, a_p, b_1, \ldots b_p) x_j \right) q_i + \\ & \left(\sum_{j=0}^{s} \gamma_{ij}^2(a_1, \ldots, a_p, b_1, \ldots b_p) x_j \right) (\frac{1}{4} q_i + \frac{1}{4}) + \\ & \left(\sum_{j=0}^{s} \gamma_{ij}^3(a_1, \ldots, a_p, b_1, \ldots b_p) x_j \right) (\frac{1}{4} q_i + \frac{3}{4}) + \\ & \left(\sum_{j=0}^{s} \gamma_{ij}^4(a_1, \ldots, a_p, b_1, \ldots b_p) x_j \right) (4q_i - 2\zeta[q_i] - 1) \end{aligned} \tag{9}$$

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

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

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

It follows immediately from the construction that

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

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

Applied inductively, the above implies that

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

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

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

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

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

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

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

and scalars

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

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

$$\beta(d_1, d_2, \ldots, d_t)x = \sum_{i=1}^{2^t} c_i \sigma(v_i \cdot \mu) \tag{10}$$

and

$$\beta(d_1, d_2, \ldots, d_t)xq = \sigma\left(q + \sum_{i=1}^{2^t} c_i \sigma(v_i \cdot \mu) - 1 \right), \tag{11}$$

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

Proof. Write β as a polynomial

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

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

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

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

$$\begin{aligned} &\beta(d_1, d_2, \ldots, d_t)x = \\ &\quad c_1\sigma(x) + c_2\sigma(d_1 + x - 1) + \cdots + c_{2^t}\sigma(d_1 + d_2 + \cdots + d_t + x - t) = \\ &\quad \sum_{i=1}^{2^t} c_i \sigma(v_i \cdot \mu) \end{aligned}$$

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

Remark 5.2 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 Equation (10) requires in this case only nine elements rather than sixteen.

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 $q \in \mathcal{C}$), 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 $\vec{\sigma}(Ax + c)$: $F_4 : \mathbb{Q}^{s+p} \to \mathbb{Q}^{\mu}, F_3 : \mathbb{Q}^{\mu} \to \mathbb{Q}^{\nu}, F_2 : \mathbb{Q}^{\nu} \to \mathbb{Q}^{\eta}, F_1 : \mathbb{Q}^{\eta} \to \mathbb{Q}^{s+p}$, 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 Equation (8) and the two q_i's of equation (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 $\widetilde{\mathcal{P}}^k(e_0, \delta[\omega], 0)$.

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

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

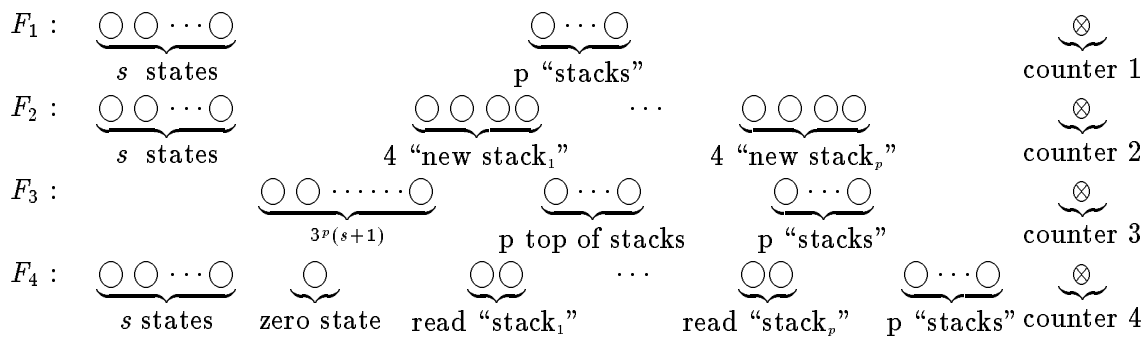
the first coordinate $z_{11}(t)$ of $z_1(t)$ equals 1 and also $y_1(t) = y_2(t-1) = 1$. To force $z_{11}(t)$ not to output arbitrary values at times that are not divisible by 4, we modify it to

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

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(a) is proved. ∎

5.1 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 $3+p$ 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 $\sigma(v_i \cdot \mu)$ that appear in Equation (10) for each of the possible $s+1$ values of the vector x. Only $3^p(s+1)$ 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 $\zeta[q_1], \ldots, \zeta[q_p], \tau[q_1], \ldots, \tau[q_p]$, 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 p stacks, from level F_4.

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

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

(obtained by applying equation (11)). 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

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

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 $\eta = s + 4p$.

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).

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.

6.1 Computing in Two Layers

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

$$q_i = \sum_{j=1}^{4} q_{ij} \quad , \tag{12}$$

where each q_{ij} is the jth term (row) in Equation (9). That is, q_{i1} may differ from 0 only if the last operation on stack i was "no-operation." Similarly, the components q_{i2}, q_{i3}, q_{i4} may differ from 0 only if the last operations of the ith stack were push0, push1, or pop, respectively.

Using Lemma 5.1, we can write

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

where

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

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

$$\begin{aligned} t_i &= \sum_{j=1}^{4} t_{ij} \quad , \\ t_{ij} &= \sigma\left(4\left[\text{next-}q_{ij} + \sum_{k=0}^{s} \gamma_{ik}^{j} x_k - 1\right] - 2\right) \quad , \end{aligned} \tag{13}$$

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

$$\begin{aligned} e_i &= \sum_{j=1}^{4} e_{ij} \quad , \\ e_{ij} &= \sigma\left(4\left[\text{next-}q_{ij} + \sum_{k=0}^{s} \gamma_{ik}^{j} x_k - 1\right]\right) \quad . \end{aligned} \tag{14}$$

Here, t_{ij} is the "top" of the element q_{ij}, and e_{ij} is the nonempty test of the same element. As three out of the four stack elements $\{q_{i1}, q_{i2}, q_{i3}, q_{i4}\}$ of each stack $i = 1, \ldots, p$ are 0, and the fourth has

the value of the stack i, it is also the case that three out of four elements of t_i (and e_i) are 0, and the fourth one holds the correct value of the top (nonempty predicate) of the relevant stack.

We construct a network in which the stacks and their readings are not kept explicitly in values q_i, t_i, e_i, but implicitly only via q_{ij}, t_{ij}, e_{ij} , $j = 1, \ldots, 4$, $i = 1, \ldots, p$. By Lemma 5.1, all update equations of

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

can be written as

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

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

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

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

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

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

In summary:

- The main layer consists of:
  1. s neurons x_k, $k = 1, \ldots, s$ that represent the state of the system unarily.
  2. For each stack i, $i = 1, \ldots, p$ we have
     (a) four neurons $q_{ij}^1 \equiv q_{ij}, j = 1, 2, 3, 4$,
     (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, $i = 1, \ldots p$ 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 show how to construct an equivalent network to the above, in which neurons in the main level compute linear combinations only. In the following construction, we introduce a set of stack "noisy-elements" $\{\tilde{q}_{i1}, \tilde{q}_{i2}, \tilde{q}_{i3}, \tilde{q}_{i4}\}$ for each stack $i = 1, \ldots p$. These may assume not only values in $[0, 1]$, but also negative values. Negative values of the stacks are interpreted as the value 0, while positive values are the true values of the stack. As in last section, only one of these four elements may assume a non-negative value at each time. The Noisy-top and Noisy-nonempty functions applied to the noisy-elements of the stacks may also produce values outside of the range $[0, 1]$.

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

$$\text{N-top } (\tilde{q}_{ij}) := c(b\tilde{q}_{ij} - (\epsilon_1 - 1)) \quad (15)$$

$$\text{N-nonempty } (\tilde{q}_{ij}) := b\tilde{q}_{ij} - (\epsilon_0 - 1)\ . \quad (16)$$

The ranges of values of these functions are

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

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

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

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

Note that:

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

The large negative numbers operate as inhibitors. We will see later how this property assists in constructing the network. As for the possibility of maintaining negative values in stack elements rather than 0, Equation (12) is not valid any more. That is, the elements $\tilde{q}_{ij}$, $j = 1, \ldots, 4$ can not be combined linearly to provide the real value of the stack q_i. This is also the case with the expression in Equations (15), and (16), and thus Equations (13) and (14) are no longer valid.

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

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

then

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

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

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

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

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

Lemma 6.1 For each $t, r \in \mathbb{N}$, let R_t be the range

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

and let

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

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

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

and scalars

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

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

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

where

$$\mu_i = \mu_{(i_1,\ldots,i_t)} = (1, d_1^{(i_1)}, d_2^{(i_2)}, \ldots d_t^{(i_t)}, x), \quad \text{and where we are defining} \quad d_i^0 = 0 \ . \qquad (17)$$

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

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

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

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

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

This is due to two facts:

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

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

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

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

for suitable c_i's and v_i's, where μ_i is defined as in (17). ∎

Remark 6.2 Note that in the case where the arguments are the functions N-top and N-nonempty, the arguments are dependent and not all $(r+1)^t$ terms are needed.

Network Description

The network consists of two levels. The main level consists of both s state neurons that are updated as beforehand, and stack noisy-neurons accompanied by stack noisy-readings neurons: $\tilde{q}_{ij}^1,\tilde{t}_{ij}^1,\tilde{e}_{ij}^1,\quad i=1,\ldots,p\quad j=1,\ldots,4$ — representing respectively stack noisy-elements, noisy-top elements, and noisy-nonempty elements. Using the same notations for the functions γ_{ij} as in Equation (9), the update equations are

$$\tilde{q}_{ij}^{1+} = \text{next-}\tilde{q}_{ij}+\sum_{k=0}^{s}\gamma_{ik}^j x_k-1\ , \tag{18}$$

$$\tilde{t}_{ij}^{1+} = (2p+1)\left[10p^2(\text{next-}\tilde{q}_{ij}+\sum_{k=0}^{s}\gamma_{ik}^j x_k-1)-(10p^2-2)\right]\ , \tag{19}$$

$$\tilde{e}_{ij}^{1+} = 10p^2(\text{next-}\tilde{q}_{ij}+\sum_{k=0}^{s}\gamma_{ik}^j x_k-1)-(10p^2-4p-2)\ , \tag{20}$$

where

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

and q_i and t_i are the exact values of the stacks and top elements. Using Lemma 6.1, all the expressions of the type γx can be written as linear combinations of terms like $\sigma($ linear combinations of $\tilde{t}_{ij}^1,\tilde{e}_{ij}^1)$.

The hidden layer consists of both up to $(5^{2p}s+2)$ configuration detectors neurons (as proved in Lemma 6.1) and the stack and top neurons:

$$q_{ij}^2,t_{ij}^2,\quad i=1,\ldots,p\quad j=1,\ldots,4\ ,$$

which are updated by the equations

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

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. This ends the proof of Theorem 2. ∎

7 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. 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 we are given a net with no inputs

$$x^+ = \sigma(Ax + c) \tag{21}$$

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

$$y^+ = \sigma(Fy + gu_1 + hu_2) \tag{22}$$

(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 (21) as follows. The new state consists of the pair (x, y), with y evolving according to (22) and the equations for x modified in this manner (using A_i to denote the ith row of A and c_i for the ith entry of c):

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

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

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

The system (22) 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 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 of time,

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

where "..." 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 slow-down of a factor of two in the computation, is:

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

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 4 letters and 7 control states, [19]. Shannon showed in [23] 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 [1] 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 deterministic 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

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

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

$$\mathcal{F}(x,u,g) \;=\; \vec{\sigma}(Ax + b_1u_1 + b_2u_2 + b_3g + c),$$

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-determinitic 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

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

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

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

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

We restrict our attention to formal non-determinictic networks that compute binary output values only, that is, $\phi_{\mathcal{L}}(\omega,\gamma) \in \{0,1\}$, where $\phi_{\mathcal{L}}(\omega,\gamma)$ 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

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

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 simultanously. 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.

10 Acknowledgment

We wish to thank Robert Solovay for his useful comments during the early steps of this research.

References

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

[2] R. Batruni, "A multilayer neural network with piecewise-linear structure and back-propagation learning," *IEEE Transactions on Neural Networks* II, May 1991: 395-403.

[3] J. Berstel, C. Reutenauer, *Rational Series and Their Languages*, Springer-Verlag, Berlin, 1988.

[4] J.R. Brown, M.M. Garber and S.F. Vanable, "Articifial neural network on a SIMD architecture," in *Proc. 2nd Symposium on the Frontier of Massively Parallel Computation,* Fairfax, VA, 1988: 43-47.

[5] L. Blum, M. Shub, and S. Smale, "On a theory of computation and complexity over the real numbers: NP completeness, recursive functions, and universal machines," *Bull. A.M.S.* **21**(1989): 1-46.

[6] A. Cleeremans, D. Servan-Schreiber, J. McClelland, "Finite State Automata and Simple Recurrent Networks," *Neural Computation*, vol. 1 (1989), p. 372.

[7] J.L. Elman, "Finding Structure in Time", *Cognitive Science*, vol. 14 (1990): 179-211.

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

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

[10] Sun, G.Z., H.H. Chen, Y.C. Lee and C.L. Giles, "Turing Equivalence of Neural Networks with Second Order Connection Weights," *Proceedings of the International Joint Conference on Neural Networks, IEEE* II, 1991: 357-362.

[11] C.L. Giles, D. Chen, C.B. Miller, H.H. Chen, G.Z. Sun, Y.C. Lee, "Second-order recurrent neural networks for grammatical inference," *Proceedings of the International Joint Conference on Neural Networks*, Seattle, Washington, IEEE Publication, vol. 2 (1991): 273-278.

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

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

[14] S.C. Kleene, "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.

[15] Lippmann, R.P., "An introduction to computing with neural nets," *IEEE Acoustics Speech and Signal Processing Magazine,* April 1987: 4-22.

[16] W. Maass, G. Schnitger, 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.

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

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

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

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

[21] J.H. Reif, J.D. Tygar, A. Yoshida "The computability and complexity of optical beam tracing," *Proc. of the 31st Annual Symp. on Foundations of Computer Science* (1990): 106-114.

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

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

[24] H.T. Siegelmann, E.D. Sontag, "Analog computation, neural networks, and circuits," *The Second Israel Symposium on Theory of Computing and Systems*, Natanya, Israel, June, 1993.

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

[26] E.D. Sontag, "Feedforward nets for interpolation and classification," *J. Comp. Syst. Sci.* **45**(1992): 20-48.

[27] G.Z. Sun, 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-.

[28] R.J. Williams, D. Zipser, A Learning Algorithm for Continually Running Fully Recurrent Neural Networks, *Neural Computation*, Vol. 1 (1989): 270-280.

[29] Wolpert, D., "A computationally universal field computer which is purely linear," techreport LA-UR-91-2937, Los Alamos National Laboratory, 1991.

[30] B. Zhang, L. Zhang and H. Zhang, "A quantitative analysis of the behavior of the PLN network," *Neural Networks* 5, 1992: 639-661.