Report *SYCON-92-05*

NEURAL NETWORKS WITH REAL WEIGHTS: ANALOG COMPUTATIONAL COMPLEXITY

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

We pursue a particular approach to analog computation, based on dynamical systems of the type used in neural networks research.

Our systems have a fixed structure, invariant in time, corresponding to an unchanging number of "neurons". If allowed exponential time for computation, they turn out to have unbounded power. However, under polynomial-time constraints there are limits on their capabilities, though being more powerful than Turing Machines. (A similar but more restricted model was shown to be polynomial-time equivalent to classical digital computation in the previous work [17].) Moreover, there is a precise correspondence between nets and standard non-uniform circuits with equivalent resources, and as a consequence one has lower bound constraints on what they can compute. This relationship is perhaps surprising since our analog devices do not change in any manner with input size.

We note that these networks are not likely to solve polynomially NP-hard problems, as the equality "$\mathrm{P} = \mathrm{NP}$" in our model implies the almost complete collapse of the standard polynomial hierarchy.

In contrast to classical computational models, the models studied here exhibit at least some robustness with respect to noise and implementation errors.

Key words:
analog computation, neural networks, dynamical systems, circuit complexity.
Research supported in part by US Air Force Grant AFOSR-91-0343.

Rutgers Center for Systems and Control
September 1992

NEURAL NETWORKS WITH REAL WEIGHTS:
ANALOG COMPUTATIONAL COMPLEXITY *

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

We pursue a particular approach to analog computation, based on dynamical systems of the type used in neural networks research.

Our systems have a fixed structure, invariant in time, corresponding to an unchanging number of "neurons". If allowed exponential time for computation, they turn out to have unbounded power. However, under polynomial-time constraints there are limits on their capabilities, though being more powerful than Turing Machines. (A similar but more restricted model was shown to be polynomial-time equivalent to classical digital computation in the previous work [17].) Moreover, there is a precise correspondence between nets and standard non-uniform circuits with equivalent resources, and as a consequence one has lower bound constraints on what they can compute. This relationship is perhaps surprising since our analog devices do not change in any manner with input size.

We note that these networks are not likely to solve polynomially NP-hard problems, as the equality "P = NP" in our model implies the almost complete collapse of the standard polynomial hierarchy.

In contrast to classical computational models, the models studied here exhibit at least some robustness with respect to noise and implementation errors.

1 Introduction

"Neural networks" have attracted much attention lately as models of analog computation. Such nets consist of a *finite* number of simple processors, each of which computes a scalar —*real-valued, not binary*— function of an integrated input. This scalar function, or "activation," is meant to reflect the graded response of biological neurons to the net sum of excitatory and inhibitory inputs affecting them. The existence of feedback loops in the interconnection graph gives rise to a dynamical system. In this paper, we introduce a mathematical model for such recurrent neural networks, and we study their computational abilities.

*This research was supported in part by US Air Force Grant AFOSR-91-0343.
Keywords: analog computation, neural networks, dynamical systems, circuit complexity.

1.1 Main Results

We focus on recurrent neural networks. In these networks, the activation of each processor is updated according to a certain type of piecewise affine function of the activations (x_j) and inputs (u_j) at the previous instant, with real coefficients —also called *weights*— (a_{ij}, b_{ij}, c_i). Each processor's state is updated by an equation of the type

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

where N is the number of processors and M is the number of external input signals. The function σ is the simplest possible "sigmoid," namely the saturated-linear function:

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

We will give later a precise definition of language acceptance for these computational models.

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

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

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

and

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

This equivalence will allow us to make use of results from the theory of (nonuniform) circuit complexity.

We show that if one allows multiplications in addition to only linear operations in each neuron, that is, if one considers instead what are often called *high order* neural nets, the computational power does not increase. Even further, and perhaps more surprising, no increase in computational power (up to polynomial time) can be achieved by letting the activation function be not necessarily the simple saturated linear one in equation (2), but any function which satisfies certain reasonable assumptions. Also, no increase results even if the activation functions are not necessarily identical in the different processors.

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

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

A Previous Related Result

In our previous work [17], we showed that if one restricts to nets all whose interconnection weights are rational numbers, which we call *rational nets*, then one obtains a model of computation that is polynomially related to Turing Machines. More precisely, given any multi-tape Turing Machine, one can simulate it in real time by some network with rational weights, and of course the converse simulation in polynomial time is obvious. Here we are interested in the case when weights are arbitrary real numbers. (It turns out that, as far as the results given here, the existence of just one irrational weight is all that is needed.)

1.2 The Model

The model we work with is that described by an iteration equation such as (1). For notational simplicity, we often summarize this equation, writing "$x^+(t)$" instead of "$x(t+1)$" and then dropping arguments t; we also write this in vector form, as

$$x^+ = \sigma(Ax + Bu + c) \tag{3}$$

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

As part of the description, we assume that we have 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. Thus a net is specified by the data (A, B, c) together with a subset of its nodes.

In our further development, both input and output channels will be forced to carry only binary data. Input and output are streams, that is, one input letter is transferred at each time (via M binary lines) and one output letter is produced at a time (and appears in the output via p binary lines). As opposed to the I/O, the computations inside the network will in general involve continuous real values.

We call a system defined by equations such as (3) simply a *network* or *processor network*. In the neural network literature, these are called recurrent *first-order* neural nets. We show later that considering *higher-order nets*, those in which multiplications of activations and/or inputs are allowed, does not result in any gain in computational capabilities (up to polynomial time).

The Finite Structure

We should emphasize from the outset that our networks are built up of *finitely many* processors, whose number *does not increase* with the length of the input. There is a small number of input channels (just two in our main result), into which inputs get presented sequentially. We assume that the structure of the network, including the values of the interconnection weights, does not change in time but rather remains constant. What changes in time are the activation values, or outputs of each processor, which are used in the next iteration. (A synchronous update model is used.) In this sense our model is very "uniform" in contrast with certain models used in the past, including those used in [9] or in the cellular automata literature, which allow the number of units to increase over time and often even the structure to change depending on the length of inputs being presented.

The Meaning of (Non-Computable) Real Weights

One may ask about the meaning of real weights. In response, we recall that our intention is to model systems in which certain real numbers —corresponding to values of resistances, capacitances, physical constants, and so forth— may not be directly measurable, indeed may not even be computable real numbers, but they affect the "macroscopic" behavior of the system. For instance, imagine a spring/mass system. The dynamical behavior of this system is influenced by several real valued constants, such as stiffness and friction coefficients. On any finite time interval, one could replace these constants by rational numbers, and the same qualitative behavior is observed, but the long-term characteristics of the system depend on the true values. We take this use of real numbers as a basic feature of analog computation. (Another characteristic would be the use of differential as opposed to difference equations, but technical difficulties make that further study harder, and we will defer it to future work.)

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

1.3 Previous work

Many authors have reported successful applications when using neural networks for various computational tasks, including classification and optimization problems. Special purpose analog chips are being built to implement these solutions directly in hardware; see for instance [1], [6]. However, very little work has been done in the direction of exploring the ultimate capabilities of such devices from a theoretical standpoint. Part of the problem is that, much interesting work notwithstanding, analog computation is hard to model, as difficult questions about precision of data and readout of results are immediately encountered —see for instance [18], and the many references there.

With the constraint of an unchanging structure, it is easy to see that classical McCulloch-Pitts—that is, binary—neurons would have no more power than finite automata, which is not an interesting situation from a theoretical complexity point of view. Therefore, and also because this is what is done in practical applications of neural nets, and because it provides a closer analogy to biological systems, we take our neurons to have a graded, analog, response. For mathematical tractability, we pick this response function to be the saturation function σ defined in Equation (2). This is a "sigmoidal" nonlinearity; one could also develop a theory using instead of σ a differentiable function such as

$$1/(1+e^{-x}) \, ,$$

but this presents technical difficulties which we prefer to avoid in this presentation. In any case, we prove below that no gain in computational abilities, even in polynomial time computation, can be achieved by using more general sigmoids, as long as they satisfy certain reasonable assumptions.

It is important to note that graded responses, as opposed to a threshold-binary output, are more reasonable in models of computing devices, as it is not reasonable to assume that physical devices can discern clearly two values which are arbitrarily close. This continuity in behavior is a basic characteristic of our model.

In the paper [19], the author 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, however, that the number of "neurons" be allowed to be infinite —as a matter of fact, in [19] the number of such units is even uncountable— as the construction relies on using different neurons to encode different possible tape configurations in Turing Machines.

The work closest to ours seems to be that on real-number-based computation started by Blum, Shub and Smale (see e.g. [4]); we believe that our setup is far simpler, and is much more appropriate if one is interested in studying neural networks or distributed processor environments. In the related previous paper [11], there were different models for each input size, the model allowed for no loops, and the emphasis was on comparisons with similar models made up of binary processors.

The remainder of this paper is organized as follows: Section 2 includes the basic definitions of networks and circuits, and states the main theorem regarding the relationships between these two models. Sections 3 and 4 contain the proof of this theorem: Section 3 shows that CIRCUIT $(F(n)) \subseteq$NET $(\text{Poly}(F(n)))$, and section 4 proves that NET $(F(n)) \subseteq$CIRCUIT $(\text{Poly}(F(n)))$. Section 5 states some corollaries for neural networks which follow from the above relation with circuits. We also define there a notion of nondeterministic network. In section 6, we briefly compare our model to the Blum, Shub, and Smale model of computation over the reals. In section 7, we show that our model does not gain power if one lets each neuron compute a polynomial function —rather than just affine combinations— of the activations of all the neurons and the external inputs, or by allowing more general activation functions than the piecewise linear one. We conclude in section 8 with a discussion on analog and non-Turing computation.

We now turn to precise definitions.

2 Basic Definitions

As we discussed above, we consider synchronous networks which can be represented as dynamical systems whose state at each instant is a real vector $x(t) \in \mathbb{R}^N$. The ith coordinate of this vector represents the activation value of the ith processor at time t. In matrix form, the equations are as in (3), for suitable matrices A, B and vector c.

Given a system of equations such as (3), 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.

2.1 Recognizing Languages

To define what we mean by a net recognizing a language

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

we must first define a *formal network*, a network which adheres to a rigid encoding of its input and output. We proceed as in [17] and define formal nets with two binary input lines. The first of these is a *data line*, and it is used to carry a binary input signal; when no signal is present, it defaults to zero. The second is the *validation line*, and it indicates when the data line is active; it takes the value "1" while the input is present there and "0" thereafter. We use "D" and "V" to denote the contents of these two lines, respectively, so

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

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

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

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

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

We now encode each word

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

as follows. Let

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

where

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

and

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

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

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

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

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

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

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

2.2 Circuit Families

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

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

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

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

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

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

The qualifier "nonuniform" serves as a reminder that there is no requirement that circuit families be recursively described. It is this lack of classical computability that makes circuits a possible model of resource-bounded "computing," as emphasized in [14]. We will show that recurrent neural networks, although more "uniform" in the sense that they have an unchanging physical structure, share exactly the same power.

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

2.3 Statement Of Result

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

Theorem 1 *Let F be so that $F(n) \geq n$. Then,* NET $(F(n)) \subseteq$CIRCUIT $(\mathrm{Poly}(F(n)))$, *and* CIRCUIT $(F(n)) \subseteq$NET $(\mathrm{Poly}(F(n)))$. ∎

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

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

3 Circuit Families Are Simulated By Networks

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

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

The proof is provided in the remainder of this section.

3.1 The circuit Encoding

Given a circuit c—with size s, width w, and w_i gates in the ith level—we encode it as a finite sequence over the alphabet $\{0, 2, 4, 6\}$, as follows:

- The encoding of each level i starts with the letter 6. Levels are encoded successively, starting with the bottom level and ending with the top one.
- At each level, gates are encoded successively. The encoding of a gate g consists of three parts—a starting symbol, a 2-digit code for the gate type, and a code to indicate which gate feeds into it:
  - It starts with the letter 0.
  - A two digit sequence $\{42, 44, 22\}$ denotes the type of the gate, {AND, OR, NOT} respectively.
  - If gate g is in level i, then the input to g is represented as a sequence in $\{2, 4\}^{w_{i-1}}$, such that the jth position in the sequence is 4 if and only if the jth gate of the $(i-1)$th level feeds into gate g.

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

Example 3.1 The circuit c_1 in Figure 1 is encoded as

$$\mathrm{en}[c_1] = \mathbf{6}\,\underbrace{042444}_{g_1}\underbrace{044424}_{g_2}\underbrace{022242}_{g_3}\mathbf{6}\,\underbrace{044444}_{g_4}\ .$$

For instance, the NOT gate corresponds to the subsequence "022242": It starts with the letter 0, followed by the two digits "22," denoting that the gate is of type NOT, and ends with "242," which indicates that only the second input feeds into the gate. □

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

$$e(\mathbf{C}) = 8\ \overline{\mathrm{en}}[c_1]\ 8\ \overline{\mathrm{en}}[c_2]\ 8\ \overline{\mathrm{en}}[c_3]\ \cdots\ , \tag{4}$$

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

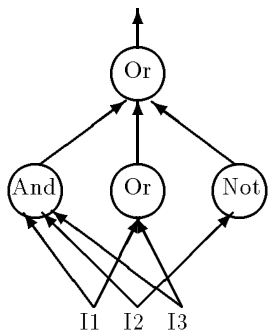

Figure 1: Circuit c_1

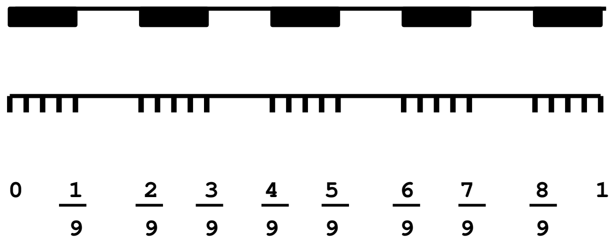

Figure 2: Values of the circuit encoding

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

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

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

We can interpret formula (4) in base 9. We denote this representation of the family of circuits $\mathbf{C}$ as $\hat{\mathbf{C}}$,

$$\hat{\mathbf{C}} = 8\, \overline{\text{en}}[c_1]\; 8\, \overline{\text{en}}[c_2]\; 8\, \overline{\text{en}}[c_3]\; \cdots|_9 \ . \tag{5}$$

Let c_i be the ith circuit in the family. We denote by $\widehat{\text{en}}[c_i]$, the encoding $\text{en}[c_i]$ interpreted in base 9.

3.2 Cantor Like Set Encoding

A number which encodes a family of circuits, or one which is a suffix of such an encoding, is a number between 0 and 1. However, not every number in the range $[0, 1]$ can appear in this manner. If the first digit to the right of the decimal point is 0, then the value of the encoding ranges in $[0, \frac{1}{9}]$; if it is 2, the value ranges in $[\frac{2}{9}, \frac{3}{9}]$, and so forth. The number cannot lie in any of the ranges $[\frac{2i-1}{9}, \frac{2i}{9}]$, for $i = 1, 2, 3, 4$. The second digit after the decimal point decides the possible range relative to the currently candidate range; see Figure 2.

In summary, not every value in $[0,1]$ appears. The set of possible values is not continuous and has "holes". Such a set of values "with holes" is a Cantor set. Its self-similar structure means that bit (base 9) 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 desired circuit out of the encoding; the circuit can be then retrieved with finite-precision operations employing a finite number of neurons.

3.3 A Circuit Retrieval

Lemma 3.2 For each (non-uniform) family of circuits $\mathbf{C}$ there exists a 16-processor network $\mathcal{N}_R(\mathbf{C})$ with one input line such that, starting from the zero initial state and given the input signal

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

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

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

Proof. Let $\Sigma = \{0,2,4,6,8\}$. Denote by $\mathcal{C}_9$ the "Cantor 9-set," which consists of all those real numbers q which admit an expansion of the form

$$q \;=\; \sum_{i=1}^{\infty}\frac{\alpha_i}{9^i} \tag{6}$$

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

$$\Lambda[x] \;:=\; \begin{cases} 0 & \text{if } x<0 \\ 9x-\lfloor 9x\rfloor & \text{if } 0\le x\le 1 \\ 1 & \text{if } x>1\;. \end{cases} \tag{7}$$

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

$$\Xi[x] \;:=\; \begin{cases} 0 & \text{if } x<0 \\ 2\lfloor \frac{9x}{2}\rfloor & \text{if } 0\le x\le 1 \\ 1 & \text{if } x>1\;. \end{cases} \tag{8}$$

Note that, for each

$$q=\sum_{i=1}^{\infty}\alpha_i/9^i \;\in\; \mathcal{C}_9\;,$$

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

$$\Xi[q]=\alpha_1\;,$$

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

$$\Lambda[q] \;=\; \sum_{i=1}^{\infty}\alpha_{i+1}/9^i \;\in\; \mathcal{C}_9\;.$$

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

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

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

Procedure Retrieval($\hat{C}, n$)
Variables counter, y, z
Begin
    counter $\leftarrow$ 0, y $\leftarrow$ 0, z $\leftarrow \hat{C}$,
    While counter $< n$
        Parbegin
            $z \leftarrow \Lambda[z]$
            if $\Xi[z] = 8$ then increment counter
        Parend,
    While $\Xi[z] < 8$
        Parbegin
            $z \leftarrow \Lambda[z]$
            $y \leftarrow \frac{1}{9}(y + \Xi[z])$
        Parend,
    Return(y)
End

The functions Λ and Ξ can not be programmed with the neural network model due to their discontinuity. However, we can program the functions $\tilde{\Lambda}, \tilde{\Xi}$, which coincide with Λ, Ξ respectively on $\mathcal{C}_9$:

$$\tilde{\Lambda}[q] = \sum_{j=0}^{8} (-1)^j \sigma(9q - j) , \tag{9}$$

and

$$\tilde{\Xi}[q] = 2 \sum_{j=0}^{3} \sigma(9q - (2j+1)) . \tag{10}$$

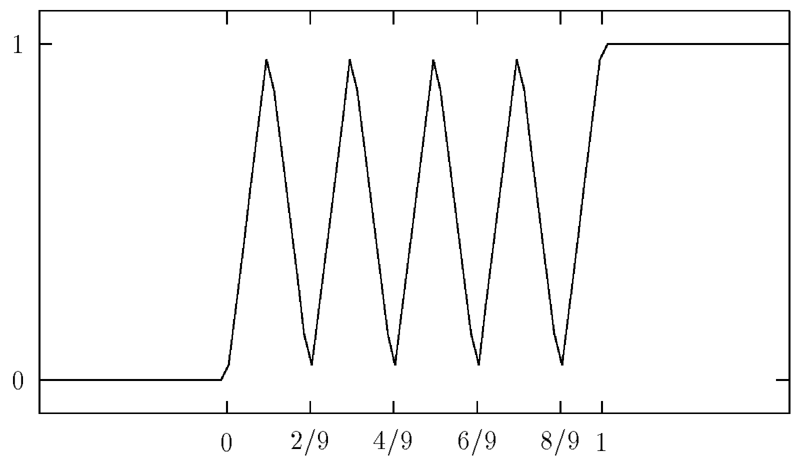

Figure 2: the function $\tilde{\Lambda}[x]$.

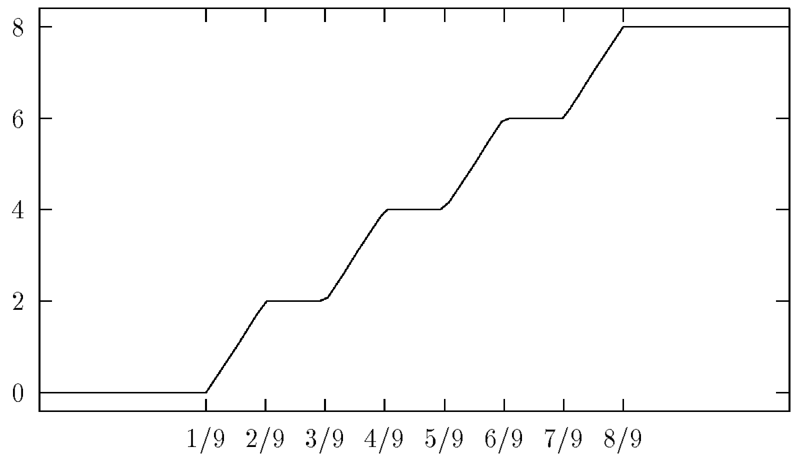

Figure 3: the function $\tilde{\Xi}[x]$.

The retrieval prodecure is, then, achieved by the following network:

$$
\begin{aligned}
x_i^+ &= \sigma(9x_{10} - i) \quad i = 0, \ldots, 8 \\
x_9^+ &= \sigma(2u) \\
x_{10}^+ &= \sigma(\hat{C}x_9 + x_0 - x_1 + x_2 - x_3 + x_4 - x_5 + x_6 - x_7 + x_8) \\
x_{11}^+ &= \sigma(\frac{1}{9}x_{12} + \frac{2}{9}(x_1 + x_3 + x_5 + x_7) - 2x_{13}) \\
x_{12}^+ &= \sigma(x_{11}) \\
x_{13}^+ &= \sigma(u + x_{14} + x_{15}) \\
x_{14}^+ &= \sigma(2x_{13} + x_7 - 2) \\
x_{15}^+ &= \sigma(x_{13} - x_7) \\
x_{16}^+ &= \sigma(x_{12} + x_7 - 1)\,.
\end{aligned}
$$

If the input u arrives at time 1, then $x_{10}(2k+3) = \tilde{\Lambda}^k[\hat{C}]$ (because of equation (9)). Processors x_{13}, x_{14}, x_{15} serve to implement the counter, and processor x_{16} is the output processor. This network satisfies the requirements of the lemma.

∎

3.4 Circuit Simulation By A Network

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

Lemma 3.3 There exists a network $\mathcal{N}_s$, such that for each circuit c and binary sequence α, starting from the zero initial state and applying the input signal

$$u_1 = \widehat{\mathrm{en}}[c]\,0\,0\cdots \quad u_2 = \widehat{\mathrm{en}}[\alpha]\,0\,0\cdots \;,$$

$\mathcal{N}_s$ outputs

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

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

Proof. It is easy to verify that, given any circuit, there is a three-tape Turing Machine which can simulate the given circuit in time $O(l(c) + |\alpha|)$. This Turing Machine would employ its tapes to store the circuit encoding, the input and output encoding, and the current level's calculation, respectively. Now we can simulate this machine by a net. Indeed, we proved in ([17]) that if M is a k-tape Turing Machine with s states which computes in time T a function f on binary input strings, then there exists a rational network $\mathcal{N}$, which consists of

$$9^k s + s + 28k + 2$$

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

Remark 3.4 If the lemma would only require an estimate of a polynomial number of processors, as opposed to the more precise estimate that we obtain, the proof would have been immediate from the consideration of the *circuit value problem* (CVP). This is the problem of recognizing the set of all pairs $<x, y>$, where $x \in \{0,1\}^+$, and y encodes a circuit with $|x|$ input lines which outputs 1 on input x. It is known that CVP $\in P$ ([3] volume I, pg 110). □

3.5 Proof: Circuit Families Are Simulated By Networks

Proof of Theorem 2.
Let **C** be a circuit family. We construct the required formal network as a composition of the following three networks:

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

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

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

- A retrieval network, $\mathcal{N}_R(c)$, as described in Lemma 3.2, which receives $u_2|_2$ from $\mathcal{N}_I$, and computes $\widehat{\text{en}}[c_{|\alpha|}]$. (Note that during the encoding operation, network $\mathcal{N}_I$ produces an output of zero, and $\mathcal{N}_R(c)$ remains in its initial state 0.)

- A simulation network, $\mathcal{N}_S$, as stated in Lemma 3.3, which receives $\widehat{\text{en}}[c_{|\alpha|}]$ and $\widehat{\text{en}}[\alpha|$, and computes

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

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

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

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

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

Remark 3.5 In case of bounded fan-in, the "standard encoding" of circuit c_n is of length $l(c_n) = O(S_{\mathbf{C}}(n)\log(S_{\mathbf{C}}(n)))$. The total running time of the algorithm is then $O(n\, S_{\mathbf{C}}(n)\log(S_{\mathbf{C}}(n)))$. □

4 Networks Are Simulated By Circuit Families

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

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

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

4.1 Linear Precision Suffices

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

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

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

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

Lemma 4.1 Let $\mathcal{N}$ be an output designated network. If $\mathcal{N}$ computes in time T, there exists a family of $T(n)$-Truncation output designated networks $\mathcal{N}_1(n)$ such that

- For each n, $\mathcal{N}_1(n)$ has the same number of processors and input and output channels as $\mathcal{N}$ does.
- The weights feeding into the internal processors of $\mathcal{N}_1(n)$ are like those of $\mathcal{N}$, but truncated after $O(T(n))$ bits.

- For each designated output processor in $\mathcal{N}$, if this processor computes $x_i^+ = \sigma(f)$, where f is a linear function of processors and inputs, then the respective processor in $\mathcal{N}_1(n)$ computes $\sigma(2\tilde{f} - .5)$, where $\tilde{f}$ is the same as the linear function f but applied instead to the processors of $\mathcal{N}_1(n)$ and with weights truncated at $O(T(n))$ bits.

- The respective output processors of $\mathcal{N}$ and $\mathcal{N}_1(n)$ have the same activation values at all times $t \leq T(n)$.

Proof. We first measure the difference (error) between the activations of the corresponding internal processors of $\mathcal{N}_1(n)$ and $\mathcal{N}$ at time $t \leq T(n)$. This calculation is analogous to that of the chop error in floating point computation, [2].

We use the following notations:

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

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

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

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

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

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

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

Therefore,

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

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

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

and

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

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

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

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

4.2 The Network Simulation by a Circuit

Lemma 4.2 Let $\mathcal{N}_1$ be a family of $T(n)$-Truncation output designated networks, where all networks $\mathcal{N}_1(n)$ consist of N processors and the weights are all rational numbers with $O(T)$ bits. Then, there exists a circuit family $\mathbf{C}$ of size $O(T^3)$, depth $O(T\log(T))$, and width $O(T^2)$, so that c_n accepts the same language as $\mathcal{N}_1(n)$ does on $\{0,1\}^n$.

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

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

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

Let

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

be a processor of $\mathcal{N}_1(n)$. We associate with each processor p a subcircuit $sc(p)$. Each processor $p \in \mathcal{N}_1(n)$ computes a truncated sum of up to $N+2$ numbers, N of which are multiplications of two T-bit numbers. Hardwiring the weights, we can say that each processor computes a sum of $(TN+2)$ $(2T)$-bit numbers. Using the carry-look-ahead method, [16], the summation can be computed via a subcircuit of depth $O(\log(TN))$, width $O(T^2N)$, and size $O(T^2N)$. (This depth is of the same order as the lower bound of similar tasks, see [5], [7].)

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

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

after the operation

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

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

We place the N subcircuits

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

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

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

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

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

The proof of Theorem 3 follows immediately from Lemma 4.1 and Lemma 4.2.

5 Corollaries

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

Corollary 5.1 NET-P = CIRCUIT-P and NET-EXP = CIRCUIT-EXP .

The class CIRCUIT-P is often called "P/poly" and coincides with the class of languages recognized by Turing Machine "with advice sequences" in polynomial time. The following corollary states that this class also coincides with the class of languages recognized in polynomial time by Turing Machines that consult oracles, where the oracles are sparse sets. A sparse set S is a set in which for each length n, the number of words in S of length at most n is bounded by some polynomial function. For instance, any tally set, that is, a subset of 1^*, is an example of a sparse set. The class $P(S)$, for a given sparse set S, is the class of all languages computed by Turing machines in polynomial time and using queries from the oracle S.

From [3], volume I, Theorem 5.5, pg 112, and Corollary 5.1, we conclude as follows:

Corollary 5.2

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

From [3], volume I, Theorem 5.11, pg 122 (originally, [13]), we conclude as follows:

Corollary 5.3 NET-EXP includes all possible binary languages. Furthermore, most Boolean functions require exponential time complexity.

Nondeterministic Neural Networks

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

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

It is easy to see that Corollary (5.1), stated for the deterministic case, holds for the nondeterministic case as well. That is, if we define NET-NP to be the class of languages accepted by nondeterministic formal networks in polynomial time, and CIRCUIT-NP to be the class of languages accepted by nondeterministic non-uniform families of circuits of polynomial size, then:

Corollary 5.4 NET-NP = CIRCUIT-NP . □

Since NP $\subseteq$ NET-NP (one may simulate a nondeterministic Turing Machine by a nondeterministic network with rational weights), the equality NET-NP = NET-P implies NP $\subseteq$ CIRCUIT-P = P/poly. Thus, from [10] we conclude:

Theorem 4 *If* NET-NP = NET-P *then the polynomial hierarchy collapses to* Σ_2. ∎

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

6 Complexity Over The Reals

Blum, Shub, and Smale introduced in [4] a powerful model of computation over the real numbers. This model allows one possible formalization of the notion of analog computing. Three main characteristics differentiate our neural network model from the BSS model, namely:

- The BSS model allows real-valued inputs, rather than only binary.
- In the BSS model, values can be compared for exact equality to any particular value, e.g., zero. That is, exact precision is available. This is not possible in our model, as discontinuities are not allowed. By appropriate choice of weights, we are able, however, to distinguish, for any fixed $\epsilon > 0$, between any two values x and y so that $|y - x| > \epsilon$.
- The BSS model allows for an infinite range of values in registers –which correspond to our "neurons"– whereas our model restricts the possible range of values to an adjustable, but finite, bound.

The BSS model is closely related to the model that is obtained if two types of neurons are available: "Heaviside" neurons that compute linear threshold functions and identity neurons. This model allows for discontinuous branching, as in the BSS model.

7 Equivalence of Different Dynamical Systems

We consider dynamical systems –which we will call *generalized processor networks*– with far less restrictive structure than the recurrent neural network model which was described in equation (1). We show that these networks are not more powerful, up to polynomial time slowdown, than the previously considered model.

Let N, M, p be natural numbers. A *generalized processor network* is a dynamical system that consists of N processors $x_1, x_2, \ldots x_N$, and receives its input $u_1(t), u_2(t), \ldots u_M(t)$ via M input lines. A subset of the N processors, say $x_{i1}, \ldots, x_{ip}$, is the set of output processors of the system, used to communicate the output of the system to the environment. In vector form, a generalized processor network D updates its processors via the dynamic equation

$$x^+ = f(x, u) ,$$

where x is the current state of the network (a vector), u is an external input (also possibly a vector), and f is a composition of functions:

$$f = \psi \circ \pi ,$$

where

$$\pi : \mathbb{R}^{N+M} \mapsto \mathbb{R}^N$$

is some vector polynomial in $N + M$ variables with real coefficients, and

$$\psi : \mathbb{R}^N \mapsto \mathbb{R}^N$$

is any vector function which has a bounded range and is locally Lipschitz. (Thus, the composite function $f = \psi \circ \pi$ again satisfies the same properties.)

We also assume, as part of the definition of generalized processor network, that, at least for binary inputs of the type considered in the definition of "formal networks," given in section 2.1, D outputs "soft" binary information. That is, there exist two constants α, β, satisfying $\alpha < \beta$ and called the *decision thresholds*, so that each output neuron of D outputs a stream of numbers each of which is either smaller than α or larger than β. We interpret the outputs of each output neuron y as a binary value:

$$\text{binary}(y) = \begin{cases} 0 & \text{if } y \leq \alpha \\ 1 & \text{if } y \geq \beta \, . \end{cases}$$

In the usual model we studied earlier, the values are always binary, but we allow more generality to show that even if one allows more general analog values, no increase in computational power is attained, at least up to polynomial time.

Remark 7.1 The above assumptions imply that, for each $\rho > 0$, there exists a constant C, such that, for all x and $\tilde{x}$ satisfying that

$$|x - \tilde{x}| < \rho \quad \text{and} \quad x \in \text{Range}\,(\psi)$$

(the absolute value sign indicates Euclidean norm), the following property holds:

$$|\psi(x, u) - \psi(\tilde{x}, u)| \leq C|x - \tilde{x}|$$

for any binary vector u. A similar property holds for f. □

A neural network is a special case of a generalized processor network, in which all coordinates of the function ψ compute the same sigmoidal function, and the polynomial π is a first order polynomial, that is, an affine function.

For the next result, we need to generalize the notion of a *truncation* network, introduced in section 4.1. Let D be a generalized processor network

$$D \,:\, x^+ \;=\; \psi(\pi(x,u))$$

as above. Let Q be a positive integer. The *Q-truncation* of D, denoted

$$Q\text{-Truncation }(D)\,,$$

is the network with dynamics defined by

$$x^+ \;=\; Q\text{-Truncation }[\psi(\pi(x,u))]\,,$$

where "Q-Truncation" represents the operation of truncating after Q bits. The *Q-chop* of D is the network with dynamics defined by

$$x^+ \;=\; Q\text{-Chop }[\psi(\pi(x,u))] \equiv\; Q\text{-Truncation }[\psi(\tilde{\pi}_Q(x,u))]\,,$$

where $\tilde{\pi}_Q$ is the same polynomial π but with coefficients truncated after Q bits.

The next observations insure that round-off errors due to truncation or chopping are not too large.

Let $T : \mathbb{N} \mapsto \mathbb{N}$ be a function from integers into integers. We say that a generalized processor network D *computes in time* T if for every input of size $n \in \mathbb{N}$, D completes its output in no more than $T(n)$ steps.

Lemma 7.2 Assume D computes in time T, with decision thresholds α, β. Then, there is a constant c such that the function

$$q(n) = cT(n)$$

satisfies the following property. For each positive integer n, let $Q = q(n)$. Then, Q-Truncation(D) computes the same function as D on inputs of length at most n, with decision thresholds

$$\alpha' = \alpha + \frac{\beta-\alpha}{3} \;\text{ and }\; \beta' = \beta - \frac{\beta-\alpha}{3}\,.$$

Proof. Let D be a generalized processor network satisfying the above conditions, and let $\tilde{D} = Q$-truncation(D), with Q still to be decided upon. Let δ be the error due to truncating after Q bits, that is, $\delta = c_1 2^{-Q}$ for some constant c_1. Finally, let ϵ_t be the largest accumulated error in all the processors by time t. The following estimates are obtained using the Lipschitz property of f:

$$\begin{aligned}
\epsilon_o &= 0 \\
\epsilon_1 &= \delta \\
\epsilon_t &= \delta \sum_{i=o}^{t-1} C^i \;= \delta \frac{C^t - 1}{C - 1}\,,
\end{aligned}$$

where C is the Lipschitz constant of f for $\rho = 1$ (c.f. Remark 7.1). To bound the error with the amount $\gamma = \frac{\beta-\alpha}{3}$, we require

$$\epsilon_t \le \gamma\,.$$

That is,

$$\delta \leq \frac{\gamma(C-1)}{C^t - 1} \leq \tilde{C}^{-t} ,$$

for some constant $\tilde{C}$. This requirement is met when δ is the truncation error corresponding to

$$Q = (\log_2((\frac{1}{c_1}\tilde{C}))T ,$$

so we can take $q(n) = \log_2(\frac{1}{c_1}\tilde{C})T(n)$. ∎

As a corollary of Lemma 4.1, and using an argument exactly as in the proof of Lemma 7.2, we conclude:

Lemma 7.3 Lemma 7.2 holds for the Q-chop network as well.

Definition 7.4 Given a vector function $f = \psi \circ \pi$ as above, we say that f is *approximable in time* $A_f(n)$, if there is a Turing Machine M that computes n-Truncation(f) in time $A_f(n)$ on each input having total bit size n.

Example 7.5 If $f = \psi \circ \pi$, ψ is approximable, and π has rational coefficients, then f is approximable. (As π is approximable at this case.) □

Theorem 5 *Let $L(T)$ be the class of languages recognized by generalized processor networks in time T, for which the function f is approximable in time A_f, and the function T is computable in time $M(n)$. Then, $L(T)$ is included in the class of languages recognized by Turing Machines in time $O(M(n) + T(n)A_f(T(n)))$.*

Proof. Assume given a generalized processor network D satisfying the above assumptions. A Turing Machine which approximates it can be built as follows. The machine receives an input string of length n. As a first step, it computes the function $T(n)$, and it estimates the required precision $Q = q(n)$ as in the previous Lemma. Finally, it simulates the generalized processor network step by step, forgetting all information but the first Q required bits. This Turing Machine computes the required function in the stated time. ∎

Corollary 7.6 Let D be a network which computes in polynomial time T, and so that f is approximable is polynomial time. Then the language recognized by D is in P.

Definition 7.7 Given a vector function $f = \psi \circ \pi$ as above, we say that f is *non-uniformly approximable in time* $A_f(n)$, if there is a Turing Machine M that using a polynomial advice function –c.f. [3], volume I, pg 99-115— computes n-Chop(f) in time $A_f(n)$ on each input having total bit size n.

Example 7.8 A polynomial π with general real coefficients is non-uniformly computable: For each input of size n, the machine receives the first $O(T(n))$ bits of each coefficient as an advice sequence, and then computes the polynomial. □

Lemma 7.9 Let $\tilde{L}(T)$ be the class of languages recognized by generalized processor networks in time T, for which the function f is non-uniformly approximable in time A_f, and the function T is computable in time $M(n)$. Then, $\tilde{L}(T)$ is included in the class of languages recognized by Turing Machines with advice sequence of length $O(T)$ in time $O(M(n) + T(n)A_f(T(n)))$.

Corollary 7.10 Let D be a network which computes in polynomial time T and so that f is non-uniformly approximable in polynomial time. Then the language recognized by D is in P/poly.

We can summarize our conclusion as a general theorem, as follows:

Theorem 6 *Let D be a generalized processor network which computes via a function $f = \psi \circ \pi$, where the function ψ is either approximable or non-uniformly approximable in polynomial time. Then there exists a neural network $\mathcal{N}_D$ which recognizes the same language as D and which does so with at most a polynomial time slowdown. Furtheremore, the weights utilized by $\mathcal{N}_D$ are of the same type as the coefficients of the polynomial π (rational or real, respectively).*

Corollary 7.11 Adding flexibility to the neural network model, described in Equation (1), does not add power to the model, except for a possible polynomial time speed up. This flexibility includes:

- Using a high order polynomial π rather than an affine function.
- Using other ψ functions rather than the saturation we used earlier, including the possibility of having different functions in different neurons.
- Allowing for the output to be "soft binary" rather than pure binary.

Note that networks with high order polynomials have appeared especially in the language recognition literature (see e.g. [8] and references there). We emphasize the relationship between these models: Let N_1 be neural network (of any order), which recognizes a language L in polynomial time. Then there is a first order network N_2 which recognizes the same language L in polynomial time.

Remark 7.12 The networks that we consider are mildly "robust to noise and to implementation error" in the sense that small enough perturbations in weights or the sigmoid activation function do not affect the computation, as long as "soft binary" outputs are considered. Given any time T, there is some ϵ_T so that an error of ϵ_T would not affect the computation up to a time T. □

8 Comments on Analog and non-Turing "Computation"

In the recent, very popular –and very controversial– book [15], Penrose has argued that the standard model of computing is not appropriate for modeling true biological intelligence. The author argues that physical processes, evolving at a quantum level, may result in computations which cannot be incorporated in Church's Thesis. It is interesting to point out that the work that we report here does allow for such non-Turing power, while keeping track of computational constraints –and thus embedding a possible answer to Penrose's challenge in more classical computer science. Note that Parberry, in [14], also insists that possible non-Turing theories should take account of such constraints, though he suggests a different approach, namely the use of probabilistic computations within the theory of circuit complexity.

Finally, we remark that human cognition seems to be clearly based on "subsymbolic" or "analog" components and modes of operation. As pointed out by many authors, in particular in the work of [12], the issue of understanding how macroscopic symbolic behavior arises from such a substrate is one of the most challenging ones in science. Perhaps our work, with its implicit use of infinite precision for internal computations, is not at all relevant to this understanding, because neurons are often taken to be low-precision devices. On the other hand, it is also possible that the precision issue should be understood solely in terms of limitations on observers and more generally interactions with the environment, and in that respect, our model is not deficient, since input and output data are binary.

9 Acknowledgement

We wish to thank Richard Beigel for suggesting that we study the generalization of our results to the high-order net case treated in section 7 .

References

[1] Alspector J., R.B. Allen, "A neuromorphic VLSI learning system," in *Advanced Research in VLSI: Proceedings of the 1987 Stanford Conference*, (P. Loseleben ed.,) MIT Press, Cambridge, MA, 1987: 313-349.

[2] Atkinson K.E., *An Introduction to Numerical Analysis*, Wiley, New York, 1989.

[3] BalCazar J.L., J. Diaz, J. Gabarro, *Structural Complexity*, Springer-Verlag, Berlin, 1988.

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

[5] Chandra A.K., L. Stockmeyer, U. Vishkin, "Constant depth reducibility," *SIAM J. Computing* **13**(1984): 423-439.

[6] Eberhardt S.P., T. Daud, D. A. Kerns, T. X. Brown, and A. P. Thakoor, "Competitive neural architecture for hardware solution to the assignment problem," *Neural Networks* **4**(1989): 431-442.

[7] Furst M., J.B.Saxe, M. Sipser "Parity, circuits, and the polynomial-time hierarchy," *Proc. 22nd IEEE Symp. Foundations of Comp. Sci.*, 1981: 260-270.

[8] Giles, C.L., C.B. Miller, D. Chen, H.H. Chen, G.Z. Sun, and Y.C. Lee, "Grammatical Inference Using Second- Order Recurrent Neural Networks," *Proceedings of the International Joint Conference on Neural Networks*, Seattle, Washington, IEEE Publication, vol. 2 (1991), p. 357.

[9] Hong J.W., "On Connectionist Models," *Comm. on Pure and Applied Mathematics* **41**(1988): 1039-1050.

[10] Karp R.M., R. Lipton, "Turing Machines that take advice," *Enseign. Math.* **28**(1982): 191-209.

[11] Maass W., G. Schnitger, and E.D. Sontag, "On the computational power of sigmoid versus Boolean threshold circuits," *Proc. 32nd IEEE Symp. Foundations of Comp. Sci.*, 1991: 767-776.

[12] MacLennan B.J., "Continuous symbol systems: The logic of connectionism," in D.S. Levine and M. Aparicio IV (eds.), *Neural Networks for Knowledge Representation and Inference*, Lawrence Erlbaum, Hillsdale, NJ, 1992.

[13] Muller D.E., "Complexity in electronic switching circuits," *IRE Trans. Electronic Comp.* **5**(1956): 15-19.

[14] Parberry I., "Knowledge, understanding, and computational complexity," Technical Report CRPDC-92-2, Center for Research in Parallel and Distributed Computing, Department of Computer Sciences, University of North Texas, Feb. 1992.

[15] Penrose R., *The Emperor's New Mind*, Oxford University Press, Oxford, 1989.

[16] Savage J.E. *The Complexity of Computing*, New Tork, Wiley, 1976.

[17] Siegelmann H.T., E.D. Sontag, "On the computational power of neural nets," in *Proc. Fifth ACM Workshop on Computational Learning Theory*, Pittsburgh, July 1992: 440-449.

[18] Vergis A., K. Steiglitz, B. Dickinson, "The complexity of analog computation," in *Math. and Computers in Simulation* **28**(1986): 91-113.

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