Discontinuities in Recurrent Neural Networks

Ricard Gavaldà
Department of Software (LSI)
Universitat Politècnica de Catalunya
Barcelona 08034, Spain
E-mail: gavalda@lsi.upc.es

Hava T. Siegelmann
Faculty of Industrial Engineering and Management
Technion, Haifa 32000, Israel
E-mail: iehava@ie.technion.ac.il

Revised, June 9th, 1998

Abstract

This paper studies the computational power of various discontinuous real computational models that are based on the classical analog recurrent neural network (ARNN). This ARNN consists of finite number of neurons; each neuron computes a polynomial net function and a sigmoid-like continuous activation function.

We introduce "arithmetic networks" as ARNN augmented with a few simple discontinuous (e.g., threshold or zero test) neurons. We argue that even with weights restricted to polynomial-time computable reals, arithmetic networks are able to compute arbitrarily complex recursive functions. We identify many types of neural networks that are at least as powerful as arithmetic nets, some of which are not in fact discontinuous but they boost other arithmetic operations in the net function, e.g. neurons that can use divisions and polynomial net functions inside sigmoid-like continuous activation functions. These arithmetic networks are equivalent to the Blum-Shub-Smale (BSS) model, when the latter is restricted to a bounded number of registers.

With respect to implementation on digital computers, we show that arithmetic networks with rational weights can be simulated with exponential precision; but even with polynomial-time computable real weights arithmetic networks are not subject to any fixed precision bounds. This is in contrast with the ARNN that are known to demand only precision that is linear in the computation time.

When nontrivial periodic functions (e.g. fractional part, sine, tangent) are added to arithmetic networks, the resulting networks are computationally equivalent to a

massively parallel machine. Thus, these highly discontinuous networks can solve the presumably intractable class of PSPACE-complete problems in polynomial time.

1 Introduction

Models of computation are in the heart of all algorithms because they specify the primitive operators which are in use. Choosing an appropriate model of computation is of great importance, but it presents us with a challenge. The model should capture the essential realistic features, while still being mathematically tractable.

In models of real number computation, one thinks of real numbers as the atomic data items. This is in contrast with models of discrete computation which handle binary digits. In real-valued models, one assumes infinite precision registers rather than bit registers, and a collection of operations on real numbers that are executed in unit time.

There are two main fields where formal models of computation with real numbers are necessary. The first is the study of biological, or biologically inspired, computations. Here, one admits that some natural systems update according to the values of their real parameters rather that their base 2 representation. Second, in areas such as computational geometry or numerical analysis, algorithms are naturally expressed in terms of real numbers. This double origin is the reason why two types of real models have been proposed: continuous and discontinuous ones.

Continuous systems allow for continuous functionality only, which is believed to better describe most of biologically motivated computations. Among the best studied continuous models are most neural networks with continuous/analog activation functions [9, 10, 12, 7], in particular those with recurrent interconnection pattern.

Real computational models with discontinuities usually include infinite-precision tests of equality and inequality, which are discontinuous by definition. Although such tests with infinite precision are often considered physically implausible, they are routinely used in algorithms in computational geometry, numerical analysis, and algebra. Two well-established models of this kind are the real RAM of Preparata and Shamos [20] and the real Turing machine suggested by Blum, Shub, and Smale [5], now usually called the BSS model. Moore [16] has recently proposed still another model (in fact, a family of models) for real-time analog computation.

Neural networks constitute a particular type of real-valued models. In this field as well we are faced with continuous neurons such as sigmoidal ones, as well as discontinuous neurons such as McCulloch-Pitts neurons. In this paper we ask what difference does it make to the computational model if our neurons are all continuous or if discontinuous neurons are incorporated as well. We choose as a starting point the continuous model called analog recurrent neural network (ARNN), typically used to analyze computational capabilities of neural networks, and consider several discontinuous extensions.

The ARNN model suggested by Siegelmann and Sontag [25, 26] consists of a fixed number of neurons in a general interconnection pattern. Each neuron is updated by

$$x_i(t+1) = \phi\left(\nu(\omega, x, u)\right) \ , \quad i = 1, \ldots, N \tag{1}$$

where the net function ν is a polynomial combination of its input (formed by the external input u and input from other neurons x; ω denotes the vector of constant coefficients or weights). It filters the result through the *linear-saturated* (ramp) activation function $\phi = \sigma$:

$$\sigma(x) = \begin{cases} 1 & \text{if } x \geq 1 \\ x & \text{if } 0 \leq x \leq 1 \\ 0 & \text{if } x \leq 0. \end{cases}$$

Such networks are sometimes classified as *first-order* and *high-order*, according to the degree of the polynomial constituting the net-function. It was previously proven that high order and first-order networks are computationally equivalent, even if other sigmoid-like, continuous, and Lipschitz activation functions ϕ are allowed besides σ [26]. Here we will consider high-order networks only. In this model, input appears to the network as a string of digits that enters a subset of the neurons, output is generated as a string as well (an equivalent model considers initial and final states and no inputs and outputs). This model is equivalent in power to Turing machines for rational weights (constants), and becomes of a nonuniform (above Turing) power when the weights are reals.

As a first stage of adding discontinuities to the analog networks we introduce in Section 3 the class of arithmetic networks. The simplest expression of this class is obtained by incorporating threshold neurons

$$\sigma_H(x) = \begin{cases} 1 & \text{if } x \geq 0 \\ 0 & \text{if } x < 0 \end{cases}$$

into the finite interconnection of analog neurons constituting the ARNN. We show in two different ways that arithmetic networks are computationally stronger than high-order (continuous) networks. For this we concentrate on networks whose weights belong to a very simple and small subset of real numbers called polynomial-time computable reals.

A real number r is called *polynomial-time computable* if there is a polynomial p and a Turing machine M such that M on input n will produce the first n digits of the fractional part of r in time $p(n)$. All algebraic numbers, constants such as π and e, and many others are polynomial-time computable. To emphasize how small this class is, we note that there are no more polynomial-time computable real numbers than Turing machines, hence there are countably many of them. Furthermore it can be shown [3] that, when used as constants in ARNN, networks still compute the class P only, just like in the case where all constants are rational numbers.

As the first evidence of the arithmetic networks' superiority, we prove that arithmetic networks can recognize some recursive functions arbitrarily faster than Turing machines and ARNN; they recognize arbitrarily complex recursive functions in linear time. The second evidence concerns the amount of precision required to implement arithmetic networks on digital computers. We show that no fixed precision function is enough to simulate all arithmetic nets running in linear time. This contrasts with ARNN even with arbitrary real weights (where linear precision in the computation time suffices) and arithmetic nets with rational weights (where exponential precision suffices).

Hence, we obtain an interesting computational class of neural networks that is potentially more powerful than the Siegelmann and Sontag's nets [26, 24]. Both multiplications and discontinuities seem necessary to obtain this class: high-order nets with only continuous, Lipschitz activation functions have at most the power of first-order nets – they are actually equivalent to them for the saturated-linear function [26]. And it follows from a more general result of Koiran [13] that adding the threshold function to first-order nets does not increase their power, either.

If we consider nets running in polynomial time, this complexity class of arithmetic nets lies between the classes P and PSPACE (P $\subseteq$ PSPACE). The first corresponds to the power of so-called *first class* serial machine models, of which the Turing machine is a prime example. The latter corresponds to *second class* models, with the power of massively parallel computers, in which time is polynomially equivalent to Turing-machine (first class) space (see Section 2 for definitions of these classes, and [29] for an exposition of first and second-class models). For all we know our class could coincide with P, PSPACE, both, or form a third intermediate class. Yet, if we show that adding threshold strictly increases the power of networks we have actually shown that P $\neq$ PSPACE. Recall, however, that the conjecture P $\neq$ PSPACE, although widely believed, is a long-standing and notoriously difficult open problem.

We show in Section 4 that many other networks share the same properties. We first notice that the threshold gates can be substituted with the gates computing the exact zero-test gates

$$\sigma_{=}(x) = \begin{cases} 1 & \text{if } x = 0 \\ 0 & \text{if } x \neq 0 \ . \end{cases}$$

There is a wide family of activation functions which gives at least the same (and possibly more) power as threshold or zero-test gates. We show that this holds for any function containing what we call "jump discontinuities". Another family is that of "launching functions", which throw values that are close to zero exponentially far away; an example is the square root. An alternative way is to stay with the saturated linear activation function in all neurons, and increase the computational capabilities of the network by enlarging the set of operators in the net function. One case is to allow the net function to compute divisions in addition to polynomials. In fact, we prove that nets with division or square root are equivalent in computational power to threshold or zero-test (up to polynomials in the running time).

In Section 5, we show that networks with thresholds (or divisions) and some pretty natural periodic functions — such as fractional part, sine, or tangent — compute up to the upper bound: PSPACE. Such periodic functions, combined with the threshold (or division), provide infinitely many periodic discontinuities as opposed to the single discontinuity of the threshold. Our proof relies strongly on the theorem by Bertoni, Mauri, and Sabadini stating that unit-cost arithmetic RAMs can solve all of PSPACE [4].

This result can be considered as complexity-theoretic evidence that it is unrealistic to assume periodic and discontinuous functions together with infinite precision. Of course, the assumption of infinite precision is physically unrealistic anyway. So far, however, there is no evidence (such as a PSPACE-hardness or an NP-hardness result) that infinite precision by itself is more helpful than polynomial precision, even in a theoretical sense.

It is interesting to compare this theorem with a recent of Moore [16] which also demonstrates, in another context, the computation power added by periodic functions. He exhibits a language that can be recognized in real time with dynamical systems with sinusoidal activation functions but cannot be recognized in real time, for example, by polynomial or sigmoidal functions.

Some of our results are proved for nets with arbitrary real weights, while others apply only to nets with rational weights only. Invariably, the restriction to rational numbers appears where our proof technique requires a reasonable bound on the smallest real number that can appear during the computation in a net. This bound is easy to obtain for rational weights, but as we showed in Section 3 it is not possible to find such a bound for general real weights. This does not necessarily imply that our missing results for real numbers are false, but shows at least that very different proof techniques will be necessary.

Before starting the technical part of the paper, let us discuss the relationship with biological neuron networks. One popular argument of discrediting the significance of computational complexity to biological modeling claims that: Not only are the artificial models far removed from nature, they also emphasize functions which require a lengthy response. In contrast, nature is likely to respond in real or at least linear time. Being endowed with the feature of arbitrary speedup in some cases, and combining analog functioning with discontinuities, our model is perhaps somewhat attractive for computational modeling of neuron networks. However our network carries a feature which is very unlikely to exist in biology: it allows for no robustness. This we termed as the lack of precision bound as opposed to the linear bound existing in the analog models. We leave as an open question the existence of network that has the desirable feature of speedup while still being subject to precision bounds.

2 Preliminaries: Computational Models

In this section we provide the preliminaries from the field of computational complexity that are required to understand the previous results as well as our new ones. We also present some known results on the computational power two real-valued models: the ARNN and the BSS model.

2.1 Alphabets, Strings, Languages

In classical computation theory, inputs are encoded as finite *strings* over a finite alphabet Σ. Most of the times we assume that $\Sigma = \{0, 1\}$, although any other alphabet with at least two letters could be used. The set $\Sigma^\star$ is the set of all finite strings over Σ. For a string $x \in \Sigma^\star$, we use $|x|$ to denote the *length* (or number of letters) of x.

We identify often natural numbers and strings via an easy isomorphism. Also, we assume the existence of an easily computable and invertible *pairing function* $\langle .,. \rangle : \Sigma^\star \times \Sigma^\star \mapsto \Sigma^\star$ encoding uniquely two strings into a third string. For example, we can encode binary strings x and y by first duplicating every bit of x, then appending $01y$. Thus, $\langle 101, 0010 \rangle = 110011010010$. This function is extended to more than two arguments by composition: $\langle x, y, z \rangle = \langle x, \langle y, z \rangle \rangle$.

In any computation model taking strings as input, resources are usually measured as a function of the length of the input string. For example, we say that the running time of any device is $t(n)$, or simply t, if the device makes at most $t(n)$ steps on any input string whose length is n.

Computational complexity theory has a technical name for the functions $t(n)$ that are at all interesting to measure running times of algorithms. These are called *time-constructible functions,* although in this paper we call them simply *time bounds.* A function $t(n) \geq 2n$ is time-constructible if there is a Turing machine that, given n, computes $t(n)$ in time $O(t(n))$. All functions that the reader may think of using as time bounds for an algorithm are time-constructible, including $n \log n$, all polynomials, and all exponentials. See [1, 11, 18] for more details and motivation.

A *formal language* L is any subset of $\Sigma^\star$. Equivalently, a language can be seen as a function from $\Sigma^\star$ to {true,false} or $\{0, 1\}$, indicating membership in L.

Languages and functions are classified in *complexity classes* according to the resources, such as running time or memory space, necessary to decide or compute them. Thus, the classes P and PSPACE is the class of all languages decided by a Turing machine in polynomial time and polynomial memory space, respectively. It is easy to argue that P is a subclass of PSPACE, but whether they are actually different is an open problem. Let us recall that the well-known class NP falls in between P and PSPACE, and that it is also unknown whether it differs or coincides with either one.

All logarithms in this paper are taken in base 2.

2.2 The Power of Real-Valued Models

In principle, analog recurrent neural networks can compute functions over the real numbers, We concentrate only on networks with discrete input/output and, more precisely, recognizing formal languages as defined above over the alphabet $\Sigma = \{0, 1\}$. For this to make sense, we must first define an encoding scheme for input and output. There are several, equivalent, ways of defining this encoding, discussed for example in [26]. We explain only one for definiteness.

A network has two input lines. The first of these is a *data line*, used to carry a binary input stream of signals; 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. Two output neurons, that take binary values only, are taken to represent the data and validation of the output. Then the computation time of a neural network is well defined and it makes sense to compare them with other real-valued models such as the BSS.

For this discussion, let us consider only polynomial running time. When all the constants are rational numbers, the computational power of ARNN is known to be exactly equal to P. For the BSS machine, the computational power is known to be somewhere between P and PSPACE, but not exactly determined. Even for the bounded-memory BSS, i.e., machines using only a constant number of registers, the exact power is not known.

When the constants are reals, the power of both models becomes non-uniform: P/poly for the ARNN, and somewhere between P/poly and PSPACE/poly for the BSS, these classes are defined, for example, in [1, 18].

We will later use the fact that ARNN can implement most of the usual constructs in programming languages, such as arithmetic on integer variables, assignments, conditional statements, and loops, the most important exception being equality and inequality tests on real variables. Some examples of ARNN programming can be found in [23].

3 The Arithmetic Networks

From now on, we will define several generalizations of the ARNN model defined in the Introduction. Each generalization can be specified by a pair (ν, ϕ), where ν is the set of net functions allowed and ϕ is the set of activation functions allowed.

Let "$\mathbb{Q}$-poly" and "$\mathbb{R}$-poly" be the set of all multivariate polynomials with rational and real coefficients, respectively. By "poly" we mean either $\mathbb{Q}$-poly or $\mathbb{R}$-poly, and we use this notation when the choice is either clear or irrelevant for the discussion.

We define *high-order networks* as these with $(\nu, \phi) = (\text{poly}, \sigma)$, and *arithmetic networks* (or threshold networks) as these computing with $(\nu, \phi) = (\text{poly}, \{\sigma, \sigma_H\})$. For discrete input, arithmetic networks are polynomial-time equivalent to BSS machines in which only a constant number of registers are used. The proof is not difficult and we omit it in order to keep focused on neuron-based models.

In many cases, real weights are much more powerful than rational ones. For example, polynomial-time high-order nets with rational weights accept only languages in P while those with real weights accept all of P/poly, which contains even non-recursive languages.

At first sight, one might think that this is due exclusively to the fact that there are uncountably many real weights, so most of them are highly non-computable, while all rational weights are easily computable in any reasonable sense. In this section we show that, when we move from first-order to higher-order threshold nets, or arithmetic nets, this simple explanation is wrong.

Indeed, we show that taking polynomial-time computable real numbers as weights increases the computational complexity of arithmetic nets in at least two ways. Note that the results in this section are absolute, not depending on any unproven conjecture such as P $\neq$ PSPACE.

Recall that it was shown in [26] that, for first-order nets, "linear precision $O(t(n))$ suffices", meaning that it is enough to have the first $O(t(n))$ bits of the real weights and activation values to achieve a correct result after $t(n)$ steps. Similarly, we will show in Lemma 4.2 that "precision $2^{t^2(n)}$ suffices" to simulate all ($\mathbb{Q}$-poly,$\{\sigma, \sigma_H\}$) nets running in time $t(n)$. As an evidence of the power of discontinuity, we show that no result of this kind is possible for arithmetic nets with even very simple weights.

Theorem 3.1 There is no computable precision function $r(n)$ such that "precision $O(r(t(n)))$ suffices" to simulate all ($\mathbb{R}$-poly,$\{\sigma, \sigma_H\}$) nets running in time $t(n)$. This is true even if only polynomial-time computable weights are used.

This theorem speaks of precision functions depending on the input size n only. It is clear that, for each set of weights, there is some amount of precision *depending on the weights* that suffices to simulate any net having these particular weights and discrete input.

As a second evidence, we show that arithmetic nets, even with simple weights, can recognize some recursive languages arbitrarily faster than Turing machines.

Theorem 3.2 There are ($\mathbb{R}$-poly,$\{\sigma, \sigma_H\}$) nets that run in polynomial time, have polynomial-time computable weights, and yet they accept recursive languages of arbitrarily high time complexity (in the Turing machine sense).

Again, this is in contrast with the first-order case and the rational-weight case. First-order nets with polynomial-time computable weights accept only languages in P [3], and arithmetic nets with rational weights can be simulated in PSPACE, so in exponential time.

Theorems 3.1 and 3.2 are both consequences of the following theorem.

Theorem 3.3 For every time-constructible function $t(n)$ there is a net $\mathcal{N}$ in ($\mathbb{R}$-poly,$\{\sigma, \sigma_H\}$) such that:

1. The weights in $\mathcal{N}$ are computable in time $O(n)$;
2. $\mathcal{N}$ runs in time $2n$;
3. The language T accepted by $\mathcal{N}$ is recursive but not decidable in time $O(t(n))$ by any Turing machine.
4. Precision $O(t(n))$ does not suffice to simulate $\mathcal{N}$, that is, if $\mathcal{N}$ is simulated with precision $O(t(n))$ a language different from T is accepted, even in the soft acceptance sense.

Proof. We first give a rough idea of how $\mathcal{N}$ is built. We take a recursive but hard language $T \subseteq 1^\star$ where "hard" means that it cannot be decided in time close to $t(n)$. We build a weight w in a way that the predicate "$1^i \in T$" is equivalent to "the $r(i)$-th bit of w is 1", where $r(i)$ is function sufficiently larger than $t(i)$. Under some additional conditions on the set T, the $r(i)$-th bit of w is computable in time $O(r(i))$ to satisfy part (1) of the theorem. Under the same conditions, $\mathcal{N}$ can access this bit using the threshold in time $O(i)$, hence it can decide T in linear time to satisfy conditions 2 and 3. On the other hand, if $\mathcal{N}$ is simulated with precision $O(t(n)) \ll r(n)$, then there is no time to access the $r(i)$-th bit of w. Then, the net cannot correctly decide whether $1^i \in T$, unless we contradict the assumption that T is not decidable in time close to $t(n)$.

Now we provide the details. For a real number $a \in [0,1]$ with binary expansion $0.a_1a_2a_3\ldots$, we denote by a_j the jth bit in its binary expansion, and by $a \downarrow j$ the number $0.\underbrace{00\ldots00}_{j-1}a_ja_{j+1}\ldots$.

Given function $t(n)$, define functions $s(n)$ and $r(n)$ as:

$$s(1) = 1 \qquad r(i) = t^5(s(i)) \qquad s(i+1) = r^2(i).$$

(Here, for example, $t^5(n)$ denotes the fifth power of t, not t iterated 5 times.) It is routine to check that s and r are time constructible if t is. We assume w.l.o.g. that $r(i+1) > r(i)+1$. Now we take the hard set T mentioned above:

Claim. There is a set T with the following properties:

1. T contains only strings of the form $1^{s(i)}$.

2. T is decidable by some Turing machine in time $t^5(n)$ but is not decidable by any Turing machine in time $O(t^4(n))$.

The existence of this T follows from a basic theorem in computational complexity theory called the Time Hierarchy Theorem. See for example [1, 11, 18] for expositions of this theorem.

Now define a pair of weights $u, w \in [0,1]$. Weight w is an encoded version of T and u is a support weight useful to find the encoding bits:

$$u_j = \begin{cases} 1 & \text{if, for some } i,\ j = r(i) \\ 0 & \text{otherwise} \end{cases}$$

and

$$w_j = \begin{cases} 1 & \text{if, for some } i,\ j = r(i) \text{ and } 1^{s(i)} \in T \\ 0 & \text{otherwise.} \end{cases}$$

Observe that, for every i, $1^{s(i)} \in T$ if and only if $w_{r(i)} = 1$, and we claim that this happens if and only if $w \downarrow r(i) \geq (u/2) \downarrow r(i)$. This is so because all bits before the $r(i)$-th are the same in both $w \downarrow r(i)$ and $(u/2) \downarrow r(i)$ (namely, 0). Furthermore, $(u/2)_{r(i)+1} = u_{r(i)} = 1$ for sure, and, because $r(i+1) > r(i) + 1$, $w_{r(i)+1} = 0$, and similarly $(u/2)_{r(i)} = 0$. So the bit $w_{r(i)}$ decides which of the two numbers is larger.

But all the bits of u and w in between $r(i-1)+1$ and $r(i)$ are 0, so this is equivalent to $w \downarrow (r(i-1)+1) \geq (u/2) \downarrow (r(i-1)+1)$. This property can be used to decide T if weights $u/2$ and w are available.

More precisely, the net $\mathcal{N}$ decides T as follows:

1. input 1^n;

2. check that $n = s(i)$ for some i, and compute $j = r(i-1)+1$;

3. from the weights w and $u/2$, compute $w' = w \downarrow j$ and $u' = (u/2) \downarrow j$;

4. output $\sigma_H(w' - u')$.

This net accepts T by the observation above, so it satisfies part (3) of the theorem. Getting the input takes time n. Note that $r(i-1)$ is $o(s(i))$ by definition of $r(n)$. Then, computing i and j takes time $o(s(i))$ by the time-constructibility of r, and obtaining u' and w' can be done in time $O(j) = o(s(i))$ with essentially the net in Lemma 4.1. This says that $\mathcal{N}$ works in time $s(i) + o(s(i)) \leq 2n$, as stated in the theorem, part (2).

To see part (1) of the theorem, see that all the weights in $\mathcal{N}$ are the rationals used for controlling the execution flow, u, and w. For u, note that checking whether $u_j = 1$ is deciding whether $j = r(i)$ for some i, which can be done in time $O(j)$ by definition of r; deciding whether $w_j = 1$, for $j = r(i)$ is possible because T is decidable in time $t^5(n)$, so deciding $1^{s(i)} \in T$ takes time $t^5(s(i)) = r(i) = j$.

Finally, for part (4) we have to show that if $\mathcal{N}$ is simulated with precision $O(t(n))$ then the language accepted is not T anymore. We argue by contradiction. If precision $O(t(n))$ suffices, we can decide T with a Turing machine as follows: given an input 1^n, with $n = s(i)$, compute weights u and w with precision $O(t(n))$; this takes time $O(t(n))$ by the time-computability of u and w. Then simulate $\mathcal{N}$ with precision $O(t(n))$ for its running time, which is at most $2n$. Arithmetic operations $+$ and $*$ with precision p can be implemented on a Turing machine in time $O(p^3)$, so the simulation can be done in time $O(n \cdot t^3(n)) \leq O(t^4(n))$. If the simulation still accepts T correctly, we contradict the fact that T is not decidable in time $O(t^4(n))$. ∎

4 Basic and Simple Discontinuities

In this section we investigate other classes of nets equivalent to arithmetic ones. We consider first the hard threshold σ_H and zero-test $\sigma_=$ functions, since they look like the "simplest" discontinuous functions in an intuitive sense. Our main result is that, indeed, they are the simplest ones in a computational sense.

In addition, we call *division networks* these defined by $(\nu, \phi) = (\{\text{poly,division}\}, \sigma)$. We prove that threshold networks are computationally equivalent to division networks. Later in Subsections 4.1 and 4.2 we consider two richer classes of simple discontinuous functions which, if included in high-order networks, form at least as strong a network as the arithmetic one. The class of *jump discontinuous* functions will do for networks with real weights, while the class of the *launching* functions is sufficient for networks with rational weights.

On the equivalence of $\sigma_=$ and σ_H, note that the presence of the saturated-linear function is essential here. In most arithmetic models, testing for zero is believed to be much easier than testing sign. For example, in arithmetic RAMs, arithmetic circuits, and straight-line programs, if only "=" instructions or gates are used, they can be simulated probabilistically or nondeterministically in polynomial time ([13], Theorem 9; [22], Theorems 4 and 5; [27], Theorem 3); for "<" gates, no easiness result of this kind is known.

Concerning the equivalence of division and σ_H, note that it is well known that division operations do not add any power to the BSS model: they can also be simulated with "<" tests. Curiously enough, in our proof the σ_H functions are not so much used to simulate divisions but, rather, to simulate the effect of the saturations of σ over the divisions; this effect has no clear parallel in the BSS model.

An important tool that we use in the construction is the Cantor-4 set encoding (as was introduced, for example, in [25]): Let $\omega = \omega_1\omega_2\omega_3\cdots$ be a finite or infinite binary string. We encode this string into the number that we call $\delta_4(\omega)$,

$$\delta_4(\omega) = \sum_{i=1}^{n} \frac{2\omega_i + 1}{4^i} ,$$

where n is the length of ω if ω is finite, and ∞ if it is infinite. If the string starts with the value 1, then the associated number has a value of at least $\frac{3}{4}$, and if it starts with 0, the value is in the range $[\frac{1}{4}, \frac{1}{2})$. The empty string is encoded into the value 0. The next bit restricts the possible value further. The set of possible values is not continuous and has "holes," it is a Cantor set. Its self-similar structure means that bit shifts preserve the "holes." The advantage of this encoding is that there is never a need to distinguish between two very close numbers in order to read the most significant digit in the base-4 representation.

Using this encoding, one can prove that:

Lemma 4.1 There is a first-order neural net that, given any real number r in Cantor-4 format, $0 \leq r \leq 1$, and a real of the form 2^{-i}, outputs the ith bit in the binary expansion of r in time linear in i.

Another tool is Lemma 4.2 stated below, which is an analog of the so-called "Linear precision suffices" Lemma 4.2 in [26] proved there for first-order networks. It states that in arithmetic networks having rational weights, the precision required in both the neurons and as the weights is at most exponentially larger than in the first-order case.

Still another term we use is *soft acceptance* [26]. In the usual model of recognizing languages by neural nets, the values of the output neurons are always binary. In the soft acceptance the output is of "soft binary values." That is, there exist two constants α, β, satisfying $\alpha < \beta$ and called the *decision thresholds*, so that each output neuron 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:

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

It is easy to transform any net accepting in the soft sense into another one accepting in the standard binary sense. We are now ready to state the lemma.

Lemma 4.2 (Exponential Precision Suffices) Let $\mathcal{N}$ be a $(\{\mathbb{Q}\text{-poly,division}\}, \{\sigma, \sigma_H\})$ net computing in $t(n)$ time and accepting a language $L \subseteq \{0,1\}^*$. Then there are constants c and d such that

1. At any time $t \leq t(n)$, the state of a neural-processor is either 0 or else greater than $2^{-2^{ct}}$.
2. If all computations of $\mathcal{N}$ are performed with precision $2^{-2^{dt(n)}}$ instead of infinite precision, $\mathcal{N}$ still accepts L, though in the "soft acceptance" sense.

Part (1) is easily proved by induction. Part (2) follows from (1): given a bound on the smallest number that can appear in a processor, it is possible to make an analysis of how the error introduced by using finite precision accumulates over time; this gives a bound on the precision needed for the output of the computation to be correct in the soft sense. This is similar to the proof in [25] for high-order nets, and is omitted.

Notes:

1. For numbers in $[0,1]$, computing with precision $2^{-2^{dt(n)}}$ is equivalent to using $2^{dt(n)}$ bits for the computation. Hence the name of the lemma.

2. The lemma may still work if we add other functions to the net, provided they cannot be used to produce small positive numbers much faster than polynomials do. In particular, this is true when any 0/1-valued functions are added. This will be used later on.

3. We showed in Section 3 that no lemma like this works for the real case: no fixed amount of precision is enough to guarantee correctness of the result when real weights are used, i.e. in a $(\{\mathbb{R}\text{-poly,division}\}, \{\sigma, \sigma_H\})$ network.

Given the lemmas above, we can state and prove the main theorem of this section, namely, that the addition of either division, threshold, or test-for-zero to high-order networks is computationally equivalent.

Theorem 4.3 For $W \in \{\mathbb{Q}, \mathbb{R}\}$, time in the following models is polynomially related:

1. Networks $(\nu, \phi) = (W\text{-poly}, \{\sigma, \sigma_=\})$.
2. Networks $(\nu, \phi) = (W\text{-poly}, \{\sigma, \sigma_H\})$.
3. Networks $(\nu, \phi) = (\{W\text{-poly,division}\}, \sigma)$.

Proof. We show that these models simulate each other with no more than polynomial overhead.

1 is equivalent to **2**. It is easy to verify that $\sigma_H(x) = \sigma_=(\sigma(-x))$ and that $\sigma_=(x) = \sigma_H(x) + \sigma_H(-x) - 1$.

2 simulates **3**. Let $\mathcal{N}$ be an division net of item 3 with N neurons. Without loss of generality, we can assume that each neuron has an update equation of one of the two forms:

$$\begin{array}{lcll} x_i^+ & := & \sigma(\ P_i(x_1, x_2, \ldots, x_N)\ ), & \text{with } P_i \text{ a polynomial} \\ x_i^+ & := & \sigma(\ x_j/x_k\ ) & \end{array}$$

We describe a neural net $\mathcal{N}'$ with additional σ_H (of item 2), that computes the same function as $\mathcal{N}$ using update equations of the form

$$\begin{array}{lcll} x_i^+ & := & \sigma(\ P_i(x_1, x_2, \ldots, x_N)\ ), & \text{with } P_i \text{ a polynomial} \\ x_i^+ & := & \sigma_H(\ x_j\ ) & \end{array}$$

Each neuron $x_i \in \mathcal{N}$ is associated with two neurons in $\mathcal{N}'$: y_i^u and y_i^d, so that at all times

$$x_i = \frac{y_i^u}{y_i^d}$$

and the values $y_i^u, y_i^d \in [0, c]$ for a constant $0 < c < 1$, to be further bounded below.

We next describe how $\mathcal{N}'$ updates each pair (y_i^u, y_i^d). We describe the simulation in three steps.

1. For each neuron, define the following polynomials z^u and z^d:

(a) For a neuron computing $\sigma(\ P_i(x_1, x_2, \ldots, x_N)\ )$, let Q_i and R_i be two polynomials such that

$$P_i(x_1, x_2, \ldots, x_N) = \frac{Q_i(y_1^u, y_1^d, \ldots, y_N^u, y_N^d)}{R_i(y_1^d, \ldots,, y_N^d)}$$

then define

$$(z^u, z^d) = (Q_i(y_1^u, y_1^d, \ldots, y_N^u, y_N^d), R_i(y_1^d, \ldots,, y_N^d))$$

(b) For a neuron computing $\sigma(x_i/x_j)$, define

$$(z^u, z^d) = (y_i^u y_j^d,\ y_i^d y_j^u)\ .$$

The constant c is chosen such that for all neurons $|z^u|, |z^d| < 1$ whenever the arguments to z^u and z^d are in $[0, c]$. This c always exists because we consider only a finite number of polynomials. Note also that, for the time being, we are not applying σ to z^u and z^d, so they may well take negative values.

2. We then normalize the values for different y's.

Case

$$\begin{array}{lll}
B_1: & (z^u = 0) \vee (z^u z^d < 0): & \\
 & \quad \text{saturate to } 0 & (y^u, y^d)^+ = (0, c) \\
B_2: & (z^u, z^d < 0) \wedge (z^u \leq z^d): & \\
 & \quad \text{saturate to } 1 & (y^u, y^d)^+ = (c, c) \\
B_3: & (z^u, z^d < 0) \wedge (z^u < -c \vee z^d < -c): & \\
 & \quad \text{back to range} & (y^u, y^d)^+ = (-cz^u, -cz^d) \\
B_4: & (z^u, z^d > 0) \wedge (z^u \geq z^d): & \\
 & \quad \text{saturate to } 1 & (y^u, y^d)^+ = (c, c) \\
B_5: & (z^u, z^d > 0) \wedge (z^u > c \vee z^d > c): & \\
 & \quad \text{back to range} & (y^u, y^d)^+ = (cz^u, cz^d)
\end{array}$$

3. We next show how to encode the algorithm as a network. First we realize that the conditions $B_1 \ldots B_5$ can be specified as

$$\begin{aligned}
B_1 &\equiv \sigma_H[\,\sigma_H(z^u)\,\sigma_H(-z^u) + \sigma_H(-z^u z^d)\,] \\
B_2 &\equiv \sigma_H[\,\sigma_H(-z^u)\,\sigma_H(z^d - z^u)\,] \\
B_3 &\equiv \sigma_H[\,\sigma_H(-z^u)\,\sigma_H(-c - z^u) + \sigma_H(-z^d)\,\sigma_H(-c - z^d)\,] \\
B_4 &\equiv \sigma_H[\,\sigma_H(z^d)\,\sigma_H(z^u - z^d)\,] \\
B_5 &\equiv \sigma_H[\,\sigma_H(z^u)\,\sigma_H(-c + z^u) + \sigma_H(z^d)\,\sigma_H(-c + z^d)\,]\ .
\end{aligned}$$

Then the update equations of the y's are given by

$$\begin{aligned}
(y^u)^+ &= \sigma((B_2 + B_4)c + B_3(-cz^u) + B_5 cz^u + (1 - \sum_{i=1}^{5} B_i)z^u) \\
(y^d)^+ &= \sigma((B_1 + B_2 + B_4)c + B_3(-cz^d) + B_5 cz^d + (1 - \sum_{i=1}^{5} B_i)z^d)\ .
\end{aligned}$$

Since z_u and z_d are polynomials, these are finite combinations of polynomials, σ, and σ_H.

3 simulates **2**. Let $\mathcal{N}$ be a neural net of item 2 with N neurons. Without loss of generality, we can assume that each neuron has an update equation of one of the two forms:

$$\begin{array}{lcll} x_i^+ & := & \sigma(\ P_i(x_1, x_2, \ldots, x_N)\ ), & \text{with } P_i \text{ a polynomial} \\ x_i^+ & := & \sigma_H(\ x_j\ ) & \end{array}$$

We describe a neural net $\mathcal{N}'$ of item 3 that computes the same function as $\mathcal{N}$ using update equations of the form

$$\begin{array}{lcll} x_i^+ & := & \sigma(\ P_i(x_1, x_2, \ldots, x_N)\ ), & \text{with } P_i \text{ a polynomial} \\ x_i^+ & := & \sigma(\ x_j/x_k\ )\ . & \end{array}$$

Neurons in $\mathcal{N}$ computing polynomials are left unchanged in $\mathcal{N}'$.

To simulate the neurons that compute hard thresholds, $\mathcal{N}'$ computes first a positive real number that is smaller than the activation value of any neuron during the computation of $\mathcal{N}$, except possibly 0. This pre-computed value is stored in a particular neuron x_{small}. That is, at any step t, if $x_j \neq 0$, then $0 < x_{small} < x_j$. Then the neuron with update equation

$$x_i^+ \ := \ \sigma_H(\ x_j\ )$$

is replaced by the equivalent one

$$x_i^+ \ := \ \sigma(\ x_j/x_{small}\ )$$

So the problem is reduced to computing this x_{small}. Consider first the case where all weights in $\mathcal{N}$ are rational. Let c be the constant provided by Lemma 4.2, part (1), for $\mathcal{N}$. At any time t, the state of a neuron is either 0 or else greater than $2^{-2^{ct}}$. Then, to compute x_{small}, $\mathcal{N}$ only has to set a neuron to $1/2$ and square its contents ct times.

When $\mathcal{N}$ contains arbitrary real weights, it is not possible to bound by any function of n the smallest activation value that can appear in the computation. In this case, however, we build into $\mathcal{N}'$ a new real weight telling how to compute such a number on-line.

Let ϵ_n be the smallest positive activation value of a neuron in a computation of $\mathcal{N}$, minimized over all neurons, computation steps, and inputs in $\{0,1\}^n$. This smallest value is defined because all computations are terminating, so there are only a finite number of choices. Assume ϵ_n appears in neuron number i_* at computation step t_* on an input $w_* \in \{0,1\}^n$.

Let $t(n)$ be the running time of $\mathcal{N}$, and define the following $t(n) \times N$ matrix M_n with entries in $\{0,1\}^2$.

$$M_n[t,k] = \begin{cases} 00 & \text{if } x_k \text{ is saturated to 0 at step } t \text{ in the computation of } \mathcal{N}(w_*) \\ 01 & \text{if } x_k \text{ is saturated to 1 at step } t \text{ in the computation of } \mathcal{N}(w_*) \\ 10 & \text{otherwise} \end{cases}$$

The "saturation" here comes from σ or σ_H, depending on k. Note that M can be seen as a binary string of length $2 \cdot N \cdot t(n)$.

Let α_n be the string $\langle w_*, t_*, i_*, M\rangle$, which has length linear in $n + N \cdot t(n)$. Let α be the infinite sequence $\alpha_0 \cdot \alpha_1 \cdot \alpha_2 \ldots$, and define $R = \delta_4(\alpha)$. Net $\mathcal{N}'$ has the real number R as a weight and, given n, obtains ϵ_n as follows:

1. Decode α_n out of R.

2. Decode w_*, t_*, i_*, and M out of α_n.

3. Simulate t_* steps of $\mathcal{N}(w_*)$ as follows: to update neuron k at step t, read the contents of $M[t,k]$; if it is 00, set x_k^+ to 0; if it is 01, set x_k^+ to 1; otherwise, set x_k^+ to $P_k(x_1,\ldots,x_N)$.

4. After step t_*, read ϵ_n from the current state of x_{i_*}, and store it in x_{small}.

Using the net in Lemma 4.1 and some neural net programming, each of the steps above takes time polynomial in $n + N \cdot t(n)$. And once x_{small} has been computed, $\mathcal{N}'$ simulates $\mathcal{N}$ in real time. Hence, the total simulation time is a polynomial of n and $t(n)$. ∎

Let us note a couple of points in these proofs. The simulation of threshold by division obtains a definite 0-1 value, the exact result of the threshold; hence, it remains valid if we introduce other operations in the net. In the converse simulation, however, the result of a division is obtained as a pair of numbers. It is not clear that the simulation goes through if we add further operations to the net, because we may need to use the number that results from the division.

Second, note that the simulation of threshold by division is not really constructive in the $\mathbb{R}$ case: the new network contains a new real weight with a lot of precoded information, and this weight depends not only on the original weights but also on how the old net uses these weights. It is of a certain interest to give a constructive proof of this theorem. Observe also that the proof needs that only inputs in $\{0,1\}^\star$ are used.

4.1 Other Jump Discontinuities

Not only the activation functions $\sigma_=$ and σ_H extend networks in this manner. We can show that many other discontinuous functions have at least the same power. We require functions that have some clear "jump" at the discontinuity, formally:

Definition 4.4 A *jump discontinuous function* f is one for which there exist real numbers a, ϵ, δ, with $\epsilon, \delta > 0$, such that for all $x \in (a, a+\epsilon]$ (or equivalently $x \in [a-\epsilon, a)$), the formula $|f(x) - f(a)| > \delta$ holds. □

Theorem 4.5 Neural nets of the type ({$\mathbb{R}$-poly,division}, σ) and ($\mathbb{R}$-poly,$\{\sigma, \sigma_H\}$) can be simulated by neural nets of the type ($\mathbb{R}$-poly, $\{\sigma, f\}$), where f is any jump discontinuous function.

Proof. We show how to simulate the function σ_H using σ and f, and the result for nets with division follows by Theorem 4.3. Let a, ϵ, and δ be as in Definition 4.4.

Let x be a number in a bounded range, $x \in [-B, B]$, for which the threshold at zero has to be implemented. There is such a B for every ($\mathbb{R}$-poly,$\{\sigma, \sigma_H\}$) net. We define

$$z(x) = a + \epsilon\sigma(\frac{x}{B})$$

such that the range $(0, B]$ is linearly mapped onto $(a, a+\epsilon]$ and the range $[-B, 0]$ is mapped to a. Now, $z \in [a, a+\epsilon]$, and we then have to simulate the threshold at a (rather than at 0) on this range. We now define

$$v(z) = \frac{1}{\delta}[f(z) - f(a)]$$

so that the range is

$$v(z) = \begin{cases} \geq 1 & f(a) < f(z) \\ \leq -1 & f(a) > f(z) \\ = 0 & z = a \end{cases}$$

and we are to simulate any function that computes 1 for the first two cases and 0 for the last case. We choose a particular function

$$k(v) = \sigma(2v - 1) + \sigma(-2v - 1)$$

which computes as required.

To summarize, the threshold at 0 can be simulated by a neural network having both σ and f activation functions, using the equation:

$$k(v(z(x))) = \sigma\{\frac{2}{\delta}[f(a + \epsilon\sigma(\frac{x}{B})) - f(a)] - 1\} + \sigma\{\frac{2}{\delta}[f(a) - f(a + \epsilon\sigma(\frac{x}{B}))] - 1\} .$$

∎

4.2 Launching Parts Simulate Discontinuities

As mentioned in the Introduction, it is known that a very large class of net functions and activation functions are equivalent to high-order networks [26]. That theorem applies to all activation functions which are bounded and Lipschitz.

Recall that f is *Lipschitz* if for every ϵ there is a c such that, for all x and y satisfying $|x-y| \leq \epsilon$, it holds $|f(x)-f(y)| \leq c \cdot |x-y|$. The Lipschitz condition, on a compact domain, is stronger than being continuous and is weaker than having derivatives.

A non-Lipschitz function f is similar to a discontinuous one in the following sense: at some parts of the function, a small change in x may produce a large change in $f(x)$. These very fast changes are precisely what makes discontinuous functions hard to compute by first-order nets.

We show an example of non-Lipschitz function, the square root, for which this similarity can be made precise: adding square root activation functions makes high-order networks computationally equivalent to threshold networks. Later we sketch how similar results can be proved for many other non-Lipschitz functions.

Theorem 4.6 For nets that use only rational weights, time in the following models is polynomially related:

1. Networks $(\nu, \phi) = (\{\mathbb{Q}\text{-poly}, \sqrt{\cdot}\}, \sigma)$.

2. Networks $(\nu, \phi) = (\mathbb{Q}\text{-poly}, \{\sigma, \sigma_H\})$.

Proof. **2** simulates **1**. Fix a $(\{\text{poly}, \sqrt{\cdot}\}, \sigma)$-net $\mathcal{N}$ that runs in time t and contains only rational weights. We can show that such a net only requires 2^{ct} bits of precision, for some constant c. This follows by an analysis of the accumulated numerical error, similar to that in the proof of Lemma 4.2, part (2).

We obtain an equivalent net $\mathcal{N}'$ replacing each processor computing $\sqrt{a}$ by a subnet running in time $t^{O(1)}$ which computes an approximation to $\sqrt{a}$ correct up to 2^{ct} bits. The subnet approximates $\sqrt{a}$ by the Newton-Raphson method: to find a solution to $x^2 - a = 0$, iterate the mapping

$$x^+ := x - \frac{x^2 - a}{2x}$$

which converges to $x = \sqrt{a}$. The following well-known fact ensures that converge is fast enough (see, e.g., [6, 14], for proofs). Here $x^{(i)}$ stands for the number that results from iterating i times the mapping starting from x.

Proposition 4.7 Let f be a real function, and $[a, b]$ an interval such that f is infinitely differentiable in $[a, b]$, $f(a) \cdot f(b) < 0$, and f' and f'' do not change sign in $[a, b]$. Then Newton-Raphson converges quadratically inside $[a, b]$, this is, there is a constant C for f such that $|f(x^{(i)})| \leq C \cdot |f(x^{(i-1)})|^2$.

Then, inductively, at least 2^t correct bits are obtained in $O(t)$ iterations. This subnet uses division, so $\mathcal{N}'$ does. But by Theorem 4.3, there is a net equivalent to $\mathcal{N}'$ using σ_H instead of division.

1 simulates **2**. By Theorem 4.3, we only have to show how to simulate $(\text{poly}, \{\sigma, \sigma_=\})$ nets. Fix one such net, and assume it runs in time t. Let c be the constant given by Lemma 4.2, part (1), this is, all activation values of this net are either 0 or else greater than $2^{-2^{ct}}$.

Replace each processor computing $\sigma_=(a)$ by a subnet that does the following: Square a to make sure $a \geq 0$; note that $\sigma_=(a) = \sigma_=(a^2)$. Set $x^{(0)} := a$. Then, iterate ct times the mapping $x^{(i)} := \sigma(\sqrt{x^{(i-1)}})$, so that $x^{(ct)} = a^{2^{-ct}}$.

If $a = 0$, then $x^{(i)} = 0$ for every i. Otherwise, $a > 2^{-2^{ct}}$, and then $x^{(ct)} > \left(2^{-2^{ct}}\right)^{2^{-ct}} = 1/2$. All in all, we obtain $\sigma_=(x)$ as $\sigma(2 \cdot x^{(ct)})$. ∎

Generalizing the second part of this proof, one can see that the square root operator in Theorem 4.6 can be substituted by any "launching" function. Say that a function f has *launching degree* α $(0 < \alpha < 1)$ if for every ϵ there is a constant c such that, for every x and y with $|x - y| < \epsilon$,

$$|f(x) - f(y)| > c \cdot |x - y|^\alpha$$

and α is the supremum of the values satisfying this property. The launching condition is opposite of the Hölder condition, where $>$ is substituted by$\leq$; it can be interpreted as being "strongly" non-Lipschitz. For the following preposition we can relax the launching condition to occur only for the fixed value $y = 0$ to get $|f(x)| > c|x|^\alpha$.

Proposition 4.8 Let f be a launching function, then for nets that use only rational weights, the networks $(\{\mathbb{Q}\text{-poly}, f\}, \sigma)$ simulate $(\mathbb{Q}\text{-poly}, \{\sigma, \sigma_H\})$ with at most polynomial slow-down.

5 Periodic Discontinuities

In this section we consider only networks that use rational numbers as weights and run in polynomial time. Consider again threshold networks. It is easy to see that these nets can compute at least all functions in P: they properly include high-order networks, that are known to compute in polynomial time exactly the class P [25]. It is also possible to show that threshold nets only compute functions included in PSPACE: for example, the unit-cost RAMs to be defined below can simulate threshold networks with a polynomial overhead, and it is known that unit-cost RAMs are at most as powerful as Turing machines working in polynomial space [22, 27]. Hence, the power of threshold networks, having a broad class of discontinuous activation functions, is located in between (or on) P and PSPACE. Recall that the inequality P $\neq$ PSPACE, although widely believed, is a long-standing open problem in the field of computer science.

We do not resolve the exact complexity of threshold nets, but we show that some activation functions sufficiently more complex than the threshold do increase the power of neural networks up to their upper bound, PSPACE. Hence, these *periodic networks* become so-called *second-class* computing models, those in which time is polynomially equivalent to Turing-machine space.

Second-class machines are usually introduced as models of massively parallel computation. Parallelism can be explicit, that is, the model explicitly uses exponentially many processors, or implicit, in that it sequentially executes operations involving exponentially large objects. The first happens, for example, with the Parallel RAM or PRAM model. The second case is true, for example, for the vector machines of Pratt and Stockmeyer [21]. See [29] for more information on second-class models.

In [2], it was shown that networks with polynomials, division, and bitwise-AND operations on rational numbers constitute a second-class machine. The proof consisted essentially of an efficient simulation of a vector machine by such a network, with the bitwise-AND used to simulate the boolean operations on vectors.

Bitwise-AND is admittedly an unnatural operation in the context of neural networks and, in general, of arithmetic models. We thus look for a computational equivalence which is more natural for this context. Bertoni, Mauri, and Sabadini [4] proved the surprising and nontrivial result that bitwise operations are not necessary to obtain second-class power. They used the following model of RAM operating on unbounded integers.

Definition 5.1 A *Random Access Machine* (RAM) consists of an infinite number of registers, R_0, R_1, R_2, ... Each register can contain any nonnegative integer number. Register R_0 is used as an accumulator, and contains the input at the start of the computation. The program of the RAM can contain the following operations:

- $R_0 := k$ /* constant load */
- $R_0 := R_i$ /* direct load */
- $R_0 := @(R_i)$ /* indirect load */

- $R_i := R_0$ /* direct store */
- $@(R_i) := R_0$ /* indirect store */
- ADD R_i /* add R_i to R_0 */
- SUB R_i /* subtract R_i from R_0; if $R_i > R_0$, set R_0 to 0 */
- MUL R_i /* multiply R_0 by R_i */
- DIV R_i /* integer divide R_0 by R_i */
- JZERO label /* jump if $R_0 = 0$ */
- HALT /* result is in R_0 */

In a *unit-cost* RAM, each instruction is executed in one unit of time, regardless of the size of the operands. The running time of a unit-cost RAM is thus the number of instructions it executes until it halts.

Bertoni, Mauri, and Sabadini proved that every problem in PSPACE is solved by a unit-cost RAM in polynomial time. In fact, their work, together with a padding argument, shows the following.

Theorem 5.2 [4] For any time bound $t(n) \geq n$, the following two models are equivalent:

1. Turing machines running in space poly$(t(n))$.
2. Unit-cost RAMs running in time poly$(t(n))$.

For our proofs, it is convenient to use RAMs that do not abuse the power of indirect addressing. We use the following folklore lemma.

Lemma 5.3 Let M be a unit-cost RAM working in time $t(n)$. Then there is an equivalent unit-cost RAM working in time $O(t(n) \log t(n))$ that only reads and writes registers with index numbers $O(t(n))$.

The idea of the proof is to organize the memory as a dictionary of pairs (i, v_i), where v_i is the last value written into R_i. When the original RAM tries to read from or write to R_i, first search the table looking for an entry with i, then read or update the value of v_i. If the dictionary is organized as a sequential table, each access costs time $O(t(n))$, as there are never more than $t(n)$ pairs in the table. Implementing the dictionary as, say, a balanced tree, the cost for each access is $O(\log t(n))$, and the memory overhead is a small multiplicative constant.

We next show two theorems. Theorem 5.4 states the second class power of periodic networks, i.e., those with polynomials, division, and the fractional part operation. Fractional part is used both to encode and decode a unit-cost RAM memory and to simulate integer division. Then in Theorem 5.6 we show that a large variety of other periodic functions, such as the sine, can simulate fractional part efficiently. So let $\sigma_F : \mathbb{R} \mapsto [0,1)$ denote fractional part.

Theorem 5.4 For time bounds $t(n) \geq n$, time in the following models is polynomially related:

1. Networks $(\nu, \phi) = (\{\mathbb{Q}\text{-poly,division}\}, \{\sigma, \sigma_F\})$.
2. Unit-cost RAMs.

Proof. To simulate 2 by 1, fix a unit-cost RAM program that runs in time $t(n)$. We describe a division net using also σ_F-neurons that simulates it in time $O(t^2(n))$. First we give some notation for a fixed input length n.

Let R be the number of registers used by M on inputs of length n. We can assume w.l.o.g. that $R = O(t(n))$ by Lemma 5.3. Fix any D such that 2^D is greater than the contents of any register of the RAM on any input of length n (we will give an explicit value for D in a moment).

For any integer m, let $code(m)$ be $m \cdot 2^{-D}$. Note that if m is stored in a register of the RAM, then $code(m) \in [0,1)$. We simulate the memory of the RAM in a fixed processor *Mem* of the net, such that at any moment:

$$Mem = \sum_{i=0}^{R} code(R_i) \cdot 2^{-iD}.$$

We can imagine each register of the RAM encoded in blocks of D binary digits inside *Mem*, something like

$$Mem = 0.\underbrace{code(R_0)}_{D \text{ bits}}\ \underbrace{code(R_1)}_{D \text{ bits}}\ \ldots \underbrace{code(R_R)}_{D \text{ bits}}$$

We describe now some basic operations of the net.

Computing 2^{-D}. It is easy to verify by induction that for every unit-cost RAM there is a constant c such that the numbers it builds in time t have value at most $(n+c)^{2^t}$. Let D be $\log(n+c)^{2^{t(n)}} = O(2^{t(n)} \log n)$. Then, the arithmetic net can compute 2^{-D} in time $O(t(n) + \log\log n) = O(t(n))$ by repeatedly squaring from $1/2$.

Extracting a field from Mem. Given i and the memory of the RAM encoded in *Mem*, we want to compute $code(R_i)$ to do some operation using R_i. Observe that

$$code(R_i) = \sigma[\ \sigma_F(\ Mem \cdot 2^{(i-1)D}\ ) - \sigma_F(\ Mem \cdot 2^{iD}\ ) \cdot 2^{-D}\ \ ]$$

The first fractional part gets rid of the code of registers $R_0 \ldots R_{i-1}$, then we subtract the code of registers $R_{i+1} \ldots R_R$, so we are left with the code of R_i. Clearly, an arithmetic net can compute this in constant time given 2^{-D}, if i is constant. (Note: numbers such as 2^{iD} cannot be stored in a processor; however, in expressions as above we write "$Mem \cdot 2^{iD}$" meaning "$Mem/2^{-iD}$" for clarity.)

Inserting a field in Mem. Given i, *Mem*, and a value $x = code(m)$, we want to update *Mem* so that $R_i = m$; i.e., we want to replace the current $code(R_i)$ with x. This is done as follows:

$$\begin{aligned} Mem^+ = \sigma[\quad & Mem - \sigma_F(\ Mem \cdot 2^{(i-1)D}\ ) \cdot 2^{-(i-1)D} \\ & + x \cdot 2^{-iD} \\ & + \sigma_F(\ Mem \cdot 2^{iD}\ ) \cdot 2^{-iD}\ \ ] \end{aligned}$$

The first line gives the codes of registers up to R_{i-1}, the second line adds x, the new code for R_i, and the third line adds the codes for registers R_{i+1} on.

With these two operations on fields, the net can simulate both direct and indirect access to register R_i. Indeed, we only have to compute numbers such as 2^{-iD}, and this can be done in time $O(t(n))$ given i, because we assume that the RAM never reads or writes registers R_i with indices $i > O(t(n))$.

Simulating arithmetic instructions. For natural numbers a and b,

$$\begin{aligned} code(a+b) &= code(a) + code(b) \\ code(a \cdot b) &= code(a) \cdot code(b) \cdot 2^D \\ code(a \text{ DIV } b) &= 2^{-D} \frac{code(a)}{code(b)} - 2^{-D}\, \sigma_F \left(\frac{code(a)}{code(b)} \right). \end{aligned}$$

Test for zero. The expression

$$\sigma(\ code(R_i) \cdot 2^D\ )$$

is 0 if $R_i = 0$, and 1 otherwise.

Putting it all together. With the building blocks above, each unit instruction of the unit-cost RAM can be simulated in time $O(t(n))$. Using some hardware to control the flow of the program, the arithmetic net: 1) reads the input in time $O(n)$; 2) computes 2^{-D} and related numbers in time $O(t(n))$; 3) simulates the program, each instruction adding a cost of $O(t(n))$; 4) when the RAM halts, the net outputs the contents of R_0. Hence, the running time is $O(n + t^2(n))$.

The converse simulation of an arithmetic net by a unit-cost RAM is much easier. As the net has only rational weights, all the states in the computation are rationals. The unit-cost RAM keeps the state of each processor as a pair (numerator, denominator), and this allows to simulate each step of the net in constant time in a straightforward manner. Note only that function σ_F is simulated by means of DIV. ∎

We next show that many other periodic functions can substitute σ_F in Theorem 5.4, together with division or threshold. One sufficient condition is the following.

Definition 5.5 Let f be a periodic function f with period P. We call f *weakly invertible* if there is a nonempty interval $[a, b) \subseteq [0, P)$ such that: i) f is infinitely differentiable in $[a, b]$; ii) for every $x \in [a, b)$, $f(x)$ has exactly one preimage in $[0, P)$.

Theorem 5.6 Let f be any weakly-invertible periodic function. Then, for time bounds $t(n) \geq n$, unit-cost RAMs are polynomially simulated by networks $(\nu, \phi) = (\{\mathbb{R}\text{-poly,division}\}, \{\sigma, f\})$ and by networks $(\nu, \phi) = (\mathbb{R}\text{-poly}, \{\sigma, \sigma_H, f\})$.

Note that the constants in the simulating networks are either rational or else constants depending on f only.

Proof. By Theorem 5.4, we only have to show how to compute σ_F using f. In fact, by the usual analysis of error propagation, it is enough if we can approximate σ_F with $2^{O(t(n))}$ bits of precision in time polynomial in $t(n)$.

Let $[a, b)$ be the interval given by the assumption that f is weakly invertible. Take a subinterval $[c, d]$ with the following properties:

- $a < c < d < b$.
- The period P is an integral multiple of $d - c$, i.e., for some natural number k we have $k \cdot (d - c) = P$.
- f, f' and f'' have constant sign inside $[c, d]$, and in particular they are not zero there. (This will be used to apply Newton-Raphson in the conditions of Proposition 4.7).

Note that if the interval $[c, d]$ cannot be chosen then, because of the third condition, every subinterval of $[a, b)$ must contain a zero of either f, f', or f''. By the assumption that f is infinitely differentiable, f has to be either constant or linear in $[a, b)$. If it is constant, then $[a, b)$ cannot witness that f is weakly invertible. If it is linear, the function h built as below is a linear transformation of σ_F so we are done with the proof. Hence we can assume for the argument that $[c, d]$ exists.

We now do some surgery on f so that it can be used to compute σ_F. See Figure 1 for an example.

((((Figure 1 here))))

Define function g by

$$g(x) = \begin{cases} f(x) & \text{if } f(x) \in [c, d] \\ 0 & \text{otherwise} \end{cases}$$

and then function h by

$$h(x) = \sum_{i=0}^{k-1} g(x + i \cdot (d - c)).$$

Now h has the following properties:

- it is a periodic function of period $d - c$ consisting of repeated copies of $f(c) \dots f(d)$;
- inside their period, neither h' nor h'' change sign, and they are never zero;
- it can be computed by a net of constant size containing f and σ_H processors; σ_H can be replaced with division as we saw in Theorem 4.3.

For simplicity, we assume from now on that h has period 1; it is enough to always divide the argument to h by its true period.

To compute $\sigma_F(z)$, do as follows:

1. Compute $y := h(z)$; observe that $h(\sigma_F(z)) = y$.
2. Solve the equation $h(x) = y$ in the interval $[c, d)$ with precision $2^{-2^{ct}}$ in x.
3. Output this x as an approximation to $\sigma_F(z)$.

To solve the equation $h(x) = y$, use Newton-Raphson method. By Proposition 4.7, the distance from x to the root after $O(t)$ Newton iterations is at most $2^{-2^{ct}}$, as we need.

Finally, to implement Newton's iteration

$$x^+ := x - \frac{h(x) - y}{h'(x)}$$

we compute a small ϵ and use $(h(x+\epsilon) - h(x))/\epsilon$ instead of $h'(x)$. We have to show that there is an ϵ computable in time polynomial in t such that the error introduced by this approximation of h' does not affect the overall result of the computation.

Assume for simplicity that $y = 0$ so we want to solve $h(x) = 0$. Let $\{x^{(i)}\}_i$ be the sequence obtained by iterating

$$x^{(i+1)} := x^{(i)} - \frac{h(x^{(i)})}{h'(x^{(i)})}$$

and $\{y^{(i)}\}_i$ be the one obtained by iterating

$$y^{(i+1)} := y^{(i)} - \frac{h(y^{(i)})}{\frac{h(y^{(i)}+\epsilon) - h(y^{(i)})}{\epsilon}}$$

from the same initial point $y^{(0)} = x^{(0)}$. We will set up a recurrence bounding $|x^{(i)} - y^{(i)}|$.

Since the initial point is the same, $|x^{(0)} - y^{(0)}| = 0$. In general,

$$|x^{(i+1)} - y^{(i+1)}| \le |x^{(i)} - y^{(i)}| + \left| \frac{h(x^{(i)})}{h'(x^{(i)})} - \frac{h(y^{(i)})}{\frac{h(y^{(i)}+\epsilon) - h(y^{(i)})}{\epsilon}} \right|.$$

To bound the second term we use that, for all u, v, s, and t,

$$\left|\frac{u}{v} - \frac{s}{t}\right| \le \max(|u|, |s|) \cdot \left|\frac{1}{v} - \frac{1}{t}\right| \le \frac{\max(|u|, |s|)}{\min^2(|v|, |t|)} \cdot |v - t|.$$

Here,

$$\left| h'(x^{(i)}) - \frac{h(y^{(i)}+\epsilon) - h(y^{(i)})}{\epsilon} \right| \le \left| h'(x^{(i)}) - h'(y^{(i)}) \right| + \left| h'(y^{(i)}) - \frac{h(y^{(i)}+\epsilon) - h(y^{(i)})}{\epsilon} \right|.$$

We have $|h'(x^{(i)}) - h'(y^{(i)})| \le k_1 \cdot |x^{(i)} - y^{(i)}|$ for a constant k_1 because h'' is bounded. Furthermore, by the mean value theorem, there is a $\varphi \in [0, \epsilon]$ such that

$$h'(y^{(i)} + \varphi) = \frac{h(y^{(i)}+\epsilon) - h(y^{(i)})}{\epsilon}. \qquad (*)$$

On the one hand, this implies that

$$\min\left(|h'(y^{(i)})|, \left| \frac{h(y^{(i)}+\epsilon) - h(y^{(i)})}{\epsilon} \right| \right) = \min\left(|h'(y^{(i)})|, |h'(y^{(i)} + \varphi)| \right)$$

is bounded from below by a constant, because h' is not zero in $[c, d]$. Therefore, as also h is bounded above by a constant,

$$\frac{\max(|h(x^{(i)})|, |h(y^{(i)})|)}{\min^2\left(|h'(y^{(i)})|, \left|\frac{h(y^{(i)}+\epsilon)-h(y^{(i)}))}{\epsilon}\right|\right)}$$

is bounded above by a constant k_2.

On the other hand, $(*)$ also implies that

$$\left|h'(y^{(i)}) - \frac{h(y^{(i)}+\epsilon) - h(y^{(i)})}{\epsilon}\right| = |h'(y^{(i)}) - h'(y^{(i)}+\varphi)| \leq k_3 \cdot \varphi \leq k_3 \cdot \epsilon,$$

for some constant k_3, because h'' is bounded. All in all,

$$\left|\frac{h(x^{(i)})}{h'(x^{(i)})} - \frac{h(y^{(i)})}{\frac{h(y^{(i)}+\epsilon)-h(y^{(i)})}{\epsilon}}\right| \leq k_2 \cdot (k_1 \cdot |x^{(i)} - y^{(i)}| + k_3 \cdot \epsilon).$$

The recurrence becomes

$$\begin{array}{l} |x^{(0)} - y^{(0)}| = 0 \\ |x^{(i+1)} - y^{(i+1)}| \leq |x^{(i)} - y^{(i)}| \cdot (1 + k_2\, k_1) + k_2\, k_3\, \epsilon \end{array}$$

that certainly satisfies

$$|x^{(i)} - y^{(i)}| \leq \epsilon \cdot (k_4)^i$$

for a constant k_4 defined from k_1, k_2, and k_3.

The analog of Lemma 4.2 works for $(\{\mathbb{Q}\text{-poly,division}\}, \{\sigma, \sigma_F\})$ nets, so we can tolerate an error in the approximation of σ_F of $2^{-2^{ct}}$, for c a constant. To have $|x^{(t)} - y^{(t)}| \leq 2^{-2^{ct}}$, it is enough to have $\epsilon \leq 2^{-2^{ct}} k_4^{-t}$, a number that can be computed in time $O(t)$ by repeated squaring.

It remains to show that we can use σ_H instead of division – recall that our proof of Theorem 4.3 did not show that this can always be done when arbitrary functions f are added. Note that division exists in the network given by Theorem 5.4, and that it is introduced in the preceding construction by Newton's method.

Because we start from a net using only rational numbers, we are now guaranteed that whenever we want to compute u/v, then $|v| > 2^{-2^{ct}}$ for some constant c. By some easy scaling we can also assume that $0 < v < 1$. Then u/v can be approximated very well as follows:

1. Compute the unique integer p such that $2^p \cdot v \in [1/2, 1)$, and define $z = 1 - 2^p \cdot v$. Since p must be in the interval $[0, 2^{ct}]$, it can be found by binary search in time $O(ct)$. Threshold is used to do the binary search.

2. Use the series

$$\frac{u}{v} = \frac{2^p\, u}{1-z} = 2^p\, u \cdot (1+z) \cdot (1+z^2) \cdot (1+z^4) \cdots (1+z^{2^i}) \cdots$$

Since $0 < z \leq 1/2$, it is enough to use $O(ct)$ terms of the series to approximate u/v with $2^{O(ct)}$ bits of precision. And by the same argument as before, this precision is enough for the whole simulation to be correct. ∎

Observe that σ_F satisfies Definition 4.4, so by Theorem 4.5, it can simulate σ_H. Hence, an immediate corollary to Theorem 5.6 is that division is not necessary in Theorem 5.4.

Corollary 5.7 For time bounds $t(n) \geq n$, time in the following models is polynomially related:

1. Networks $(\nu, \phi) = (\{\mathbb{Q}\text{-poly,division}\}, \{\sigma, \sigma_F\})$.
2. Networks $(\nu, \phi) = (\mathbb{Q}\text{-poly}, \{\sigma, \sigma_F\})$.

□

Functions such as σ_F and tangent are easily seen to be weakly invertible. Sine is not because all points in the range have two preimages in the period, except for $\pi/2$ and $3\pi/2$. But the following variant of sine is weakly invertible.

$$\text{half-sine}(x) = \begin{cases} \sin(x) & \text{if } \sin(x + \pi/2) \geq 0 \\ 0 & \text{otherwise.} \end{cases}$$

In words, half-sine filters out the parts of the sine with negative slope. Furthermore, it can be computed with a $<$ gate and a sine gate, and $<$ can be replaced by division with the technique in Theorem 4.3.

Note that all weakly invertible functions must be discontinuous, to have an injective part. If the discontinuity is of the "jump" type, we can apply Theorem 4.5 and get rid of σ_H. This is the case, for example, for the tangent function, because $\sigma(\tan)$ has a jump discontinuity. The fact that it is not defined at the discontinuity is not problematic: it is easy to ensure that the function is never evaluated at undefined points by offsetting its argument with a sufficiently small number.

All in all, we have for example the following corollaries:

Corollary 5.8 Unit-cost RAMs are polynomially simulated by

- $(\{\text{poly,division}\}, \{\sigma, \sin\})$ networks,
- $(\text{poly}, \{\sigma, \sigma_H, \sin\})$ networks, and
- $(\text{poly}, \{\sigma, \tan\})$ networks.

Therefore, these nets are second class machines, and in particular they can solve all PSPACE problems in polynomial time. □

Note that the trick used to obtain a weakly invertible function from sine is likely to work for many other natural functions, though we do not attempt to formalize when.

6 Conclusions

Our results seem to point out both theoretical advantages and inconveniences of discontinuous models. On the one hand, we have proved that discontinuities can speed up arbitrarily some computations. On the other hand, continuous models allow for precision bounds, or in other words they have some robustness to noise; discontinuities seem to ruin this property.

In summary, there is a trade-off between computational power and robustness to noise. This trade-off should perhaps be taken into account when modeling with neural networks. Obviously, no realistic modeling can use infinite precision neurons. It is an open problem whether discontinuous operators help in solving natural problems any faster, if we model now using neurons of a moderate precision.

Acknowledgments

We thank José L. Balcázar, Amir Ben-Amram, and Felipe Cucker for helpful comments and pointers to bibliography. We are grateful to the four anonymous referees for a thorough reading and many comments. We also thank Pekka Orponen for inviting us to visit the University of Helsinki, where part of this work was done.

The work was supported in part by the Israeli ministry of arts and sciences, by the U.S.-Israel Binational Science Foundation (BSF), by the Fund for Promotion of Research at the Technion, by the E.U. through the ESPRIT Working Group NeuroCOLT (nr. 8556) and Long Term Research Project ALCOM IT (nr. 20244), and by DGICYT under grant PB95-0787 (project KOALA).

References

[1] J.L. Balcázar, J. Díaz, and J. Gabarró, *Structural Complexity I.* EATCS Monographs on Theoretical Computer Science, vol. 11. Springer-Verlag, 1988.

[2] J.L. Balcázar, R. Gavaldà, H.T. Siegelmann, and E.D. Sontag, "Some structural complexity aspects of neural computation", in *Proc. 8th Annual IEEE Conf. on Structure in Complexity Theory* (1993), 253–265.

[3] J.L. Balcázar, R. Gavaldà, and H.T. Siegelmann, "Computational power of neural networks: A characterization in terms of Kolmogorov complexity". *IEEE Transactions on Information Theory* **43** (1997), 1175–1183.

[4] G. Bertoni, G. Mauri, and N. Sabadini, "Simulations among classes of random access machines and equivalence among numbers succinctly represented", *Ann. Discrete Mathematics* **25** (1985), 65–90. Preliminary version in *Proc. 13th ACM Symp. on the Theory of Computing* (1981), 168–176.

[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] L. Blum, F. Cucker, M. Shub, and S. Smale, *Complexity and Real Computation.* Springer-Verlag, 1998.

[7] P.S. Churchland and T. J. Sejnowski, *The Computational Brain*, MIT Press, Cambridge, 1992.

[8] F. Cucker and D. Grigoriev, "On the power of real Turing machines over binary inputs". *SIAM Journal on Computing* **26** (1997), 243–254.

[9] S. Haykin, *Neural Networks: A Comprehensive Foundation* IEEE Press, New York, 1994.

[10] J. Hertz, A. Krogh, and R. Palmer, *Introduction to the Theory of Neural Computation*, Addison-Wesley, Redwood City, 1991.

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

[12] J. Kilian and H. T. Siegelmann, "The dynamic universality of sigmoidal neural networks," *Information and Computation* **128** (1996), 48–56.

[13] P. Koiran, "A weak version of the Blum, Shub & Smale model." *Journal of Computer and System Sciences* **54** (1997), 177–189.

[14] S. Lang, *Undergraduate Analysis.* Springer-Verlag, 1983.

[15] W. Maass, G. Schnitger, and E.D. Sontag, "A comparison of the computational power of sigmoid and Boolean threshold circuits", in *Theoretical Advances in Neural Computation and Learning,* Kluwer Academic Publishers, 1994, 127–151. Preliminary version, "On the computational power of sigmoid versus Boolean threshold circuits," in *Proc. 32nd IEEE Symp. on Foundations of Computer Science* (1991), 767–776.

[16] C. Moore, "Dynamical recognizers: Real-time language recognition by analog computers". *Theoretical Computer Science* **201** (1998), 99–136.

[17] P. Orponen, "Neural networks and complexity theory," in *Proc. 17th Symposium on Mathematical Foundations of Computer Science,* Springer-Verlag Lecture Notes in Computer Science, vol. 629 (1992), 50–61.

[18] C. H. Papadimitriou, *Computational Complexity.* Addison-Wesley, 1994.

[19] I. Parberry, *Circuit Complexity and Neural Networks.* MIT Press, 1994.

[20] F.P. Preparata and M.I. Shamos, *Computational Geometry,* Springer-Verlag, 1985.

[21] V.R. Pratt and L.J. Stockmeyer, "A characterization of the power of vector machines", *Journal of Computer and System Sciences* **12** (1976), 198–221.

[22] A. Schönhage, "On the power of Random Access Machines". *Proc. 6th Intl. Colloquium on Automata, Languages, and Programming, ICALP'79.* Springer-Verlag Lecture Notes in Computer Science, vol. 71 (1979), 520–529.

[23] H. T. Siegelmann, "On NIL: The software constructor of neural networks," *Parallel Processing Letters* **6** (1996), 575–582.

[24] H. T. Siegelmann, "Computation beyond the Turing limit". *Science* **268** (1995), 545–548.

[25] H. T. Siegelmann and E. D. Sontag, "On the computational power of neural nets". *Journal of Computer and System Sciences* **50** (1995), 132–150.

[26] H. T. Siegelmann and E. D. Sontag, "Analog computation via neural networks". *Theoretical Computer Science* **131** (1994), 331–360.

[27] J. Simon, "Division is good". *Proc. 20th IEEE Symp. on Foundations of Computer Science* (1979), 411–420.

[28] K.Y. Siu, V.P. Roychowdhury, and T. Kailath, *Discrete Neural Computation: A Theoretical Foundation.* Prentice-Hall, 1994.

[29] P. Van Emde Boas, "Machine models and simulations", in *Handbook of Theoretical Computer Science*, vol. A, MIT/Elsevier, 1990, 1–66.

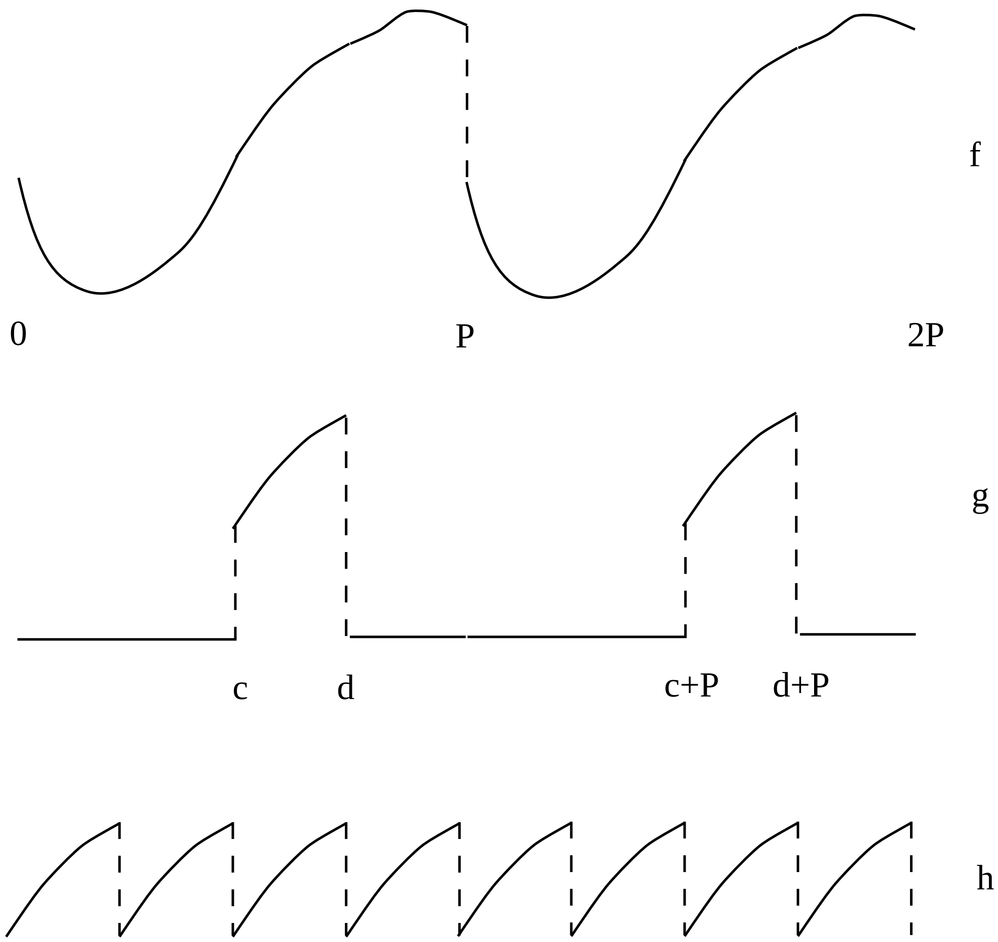

Figure 1: Transforming f to h, an example.