Some Structural Complexity Aspects of Neural Computation

José L. Balcázar* Ricard Gavaldà*

Department of Software (LSI)
Universitat Politècnica de Catalunya
Barcelona 08028, Spain

Hava T. Siegelmann† Eduardo D. Sontag†

Department of Computer Science Department of Mathematics
Rutgers University
New Brunswick, NJ 08903

E-mail: balqui@lsi.upc.es, gavalda@lsi.upc.es,
siegelma@yoko.rutgers.edu, sontag@control.rutgers.edu

Abstract

Recent work by Siegelmann and Sontag has demonstrated that polynomial time on linear saturated recurrent neural networks equals polynomial time on standard computational models: Turing machines if the weights of the net are rationals, and nonuniform circuits if the weights are reals. Here we develop further connections between the languages recognized by such neural nets and other complexity classes. We present connections to space-bounded classes, simulation of parallel computational models such as Vector Machines, and a discussion of the characterizations of various nonuniform classes in terms of Kolmogorov complexity.

1 Introduction

Among the many research issues suggested by neural computational models, the problem of precisely knowing the power of the different models under different resource bounds is clearly worth attention. Like for other computational models, the analysis of the resources necessary to complete a computation is a practically important, theoretically profound, and difficult consideration. This paper characterizes the computational power of certain resource-bounded neural models in terms of some familiar complexity classes of decisional problems.

A number of such relationships are already known, mostly for nets with threshold activation functions. Threshold nets can be thought of as a model of discrete computation, since at each moment the state of each neuron is a binary value. The model we treat here is analog, in the sense that the states of the neurons are real numbers obtained through a continuous activation function. Therefore the relationships we obtain are quite different in kind, and are based on different techniques.

More precisely, neural nets in which each neuron computes a threshold function lead to characterizations in terms of circuit classes and other known computational models, and actually the simplest widely known model, the finite automaton, was initially suggested as a characterization of the power of finite neural nets with threshold behavior [9]. Since in this case a constant number of neurons can only yield regular languages, nets of nonconstant size are considered. By bounding in various manners the growth of the neural net with respect to the input length, characterizations can be found in terms of boolean circuits. The excelent surveys [10] and [11] provide a precise account of these characteri-

*Research supported in part by the ESPRIT Basic Research Actions Program of the EC under contract No. 7141 (project ALCOM II).

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

zations. Most of them correspond to acyclic neural nets. Some of them characterize cyclic nets with time bounds by "unwinding" them into acyclic nets. Our results correspond to essentially cyclic nets in the sense that the proof techniques in no case rely on any unwinding process.

A quite ample repertory of functions has been proposed for the action of each computation unit in neural models. We focus on neurons whose real-valued states are computed by combining, in an affine or polynomial way, the inputs obtained from preceding neurons, and then filtering the result through a sort of approximation to a sigmoid. More precisely, our approximation is known as "linear saturated response": it is zero for negative arguments, the identity for arguments between zero and one, and stays constant at one for larger arguments. This behavior is essentially different from the threshold function case.

Actually thresholds present a problematic discontinuity since they require to sharply distinguish between -2^{-k} and 2^{-k} for no matter how large a k. As linear saturation is continuous, such objection does not arise. Still the discontinuity of the derivative at the saturation points makes it somewhat objectable in the grounds of implementations on physical systems, and make preferable a standard smooth sigmoid. However, linear saturation is clearly reasonable as an approximation that still allows for study without resorting to computability and complexity in the real field [5], and therefore admitting characterizations in terms of standard complexity classes based on the boolean semiring.

The starting point of the work reported here is the result by Siegelmann and Sontag [13] that proves that bounded size, linear saturated, cyclic neural nets with rational weights (and therefore rational states) are equivalent in power to Turing machines, with polynomial time overhead in both directions. Actually, it was proved there that the simulation of a Turing machine by a neural net can be done in linear time. A particularly noteworthy consequence is that, the proof being completely constructive, it allows one to compute an actual constant bound on the size of a universal neural net, based on a universal Turing machine with small tape alphabet and state set: 1058 neurons suffice to decide in time $T(n)$ any language Turing-decidable in time $T(n)$.

Here we extend these results in several directions. One is to classes defined by space bounds on Turing machines. As a resource in neural nets corresponding to memory space, we identify the size of binary descriptions of the rational states of the neurons during the computation. A number of technical considerations are required due to the input convention of the neural net, and will be discussed in the text; in particular, the simulation of certain online machines requires a more efficient simulation than that of [13]. Indeed, a neural net can simulate a Turing machine in real time (although the proof of this fact is deferred to the complete version of this paper).

Similarly, we consider classes defined by parallel time bounds. Actually neural nets are considered a very appropriate model of parallel computation, due to the fact that the net result embodies the activity of a large number of neurons (the so-called Parallel Distributed Processing). We find rather interesting the fact that our model of neural nets can achieve exactly the power of parallel machines of the Second Machine Class (see [3] or [15]) *even with a bounded number of neurons.* To characterize parallel time, we follow an intuition familiar to the complexity theorist: to allow the model to manipulate large objects in short time. More precisely, although there is no difference (modulo a polynomial) in the power of our cyclic neural nets if polynomials instead of affine combinations are used to compute the argument fed into the sigmoid, we prove that second class power is obtained if they can use rational functions (i.e. division) and bitwise AND, and obey an exponential precision bound.

We also consider the case of real-valued weights and states, studying again both the affine or polynomial case, and the case of second class power. The following interesting result was proved in [14]: with real weights and states, bounded size, linear saturated, cyclic neural nets simulate (nonuniform) boolean circuits so that neural net time and circuit size are polynomially related. Thus, for instance, in polynomial time these neural nets accept exactly the languages in P/poly, and in exponential time they can accept *any* arbitrary set. We relate this fact to the preceding ones regarding parallel time classes: the use of division and bitwise AND in this case provides exactly the power of nonuniform parallel computation, so that time corresponds to nonuniform (bounded fan-in) circuit depth; in particular, any arbitrary set can be decided in linear time by nets with real weights, provided that division and bitwise AND are available. This corresponds to writing arbitrary boolean functions as sum of minterms in linear depth. So, essentially real weights add the characteristic of nonuniformity to both the sequential and the parallel models. Thus in a sense the technical merit of this result is that of [14].

A natural question regarding nonuniform classes is the possibility of bounding the amount of advice corresponding to the class. We also study how such bounds are reflected in the neural model. It can be argued that, if some nets with real weights are computationally feasible to implement, then short descriptions must exist for their real-valued weights. It is therefore interesting to have characterizations of the accepted languages in terms of the amount of information and resources required to construct these reals.

Thus we set bounds on the resource-bounded Kolmogorov complexity of the reals used as weights in the neural nets, and prove that such bounds correspond precisely to the amount of advice allowed to nonuniform classes between P and P/poly, as studied previously in [4]. It is known that P/poly and some subclasses can be characterized by polynomial time with tally oracles: we show that the complexity of the reals in the net corresponds also with the Kolmogorov complexity of these tally oracles. Using such Kolmogorov complexity arguments, we prove that there exists a proper hierarchy of complexity classes defined by neural nets whose weights have increasing Kolmogorov complexity. All this is proved by combining the contributions of [14] with some structural constructions taking care of the Kolmogorov complexity conditions.

2 Preliminaries

2.1 Structural Complexity

The concepts from Complexity Theory mentioned through this paper are all standard; see [2] for undefined notions.

Complexity classes are sets of formal languages. A formal language is a set of words over the alphabet $\{0,1\}$. By standard encoding methods, any other finite, fixed alphabet could be assumed if necessary provided that it has at least two different symbols. We denote by $w_{1:k}$ the word consisting of the first k symbols of w; this is valid too when w is an infinite sequence. The length of a word w is denoted $|w|$, and overloading the notation we denote by $|A|$ the cardinality of the finite set A.

For any alphabet Σ, Σ^* is the set of all words over Σ; $\Sigma^{\leq n}$ is the set of all words of length at most n, and $A^{\leq n} = A \cap \Sigma^{\leq n}$; similarly we have $\Sigma^{=n}$ and $A^{=n}$. Here we will use in particular the alphabets $\Sigma = \{0,1\}$ and $\Sigma = \{0\}$. A tally set is a set of words over this single letter alphabet $\{0\}$. The strings of Σ^* are ordered by lengths and lexicographically within each length.

If A is a set of words, $\chi_A \in \{0,1\}^\infty$ is the characteristic sequence of A, defined in the standard way: the i^{th} bit of the sequence is 1 if and only if the i^{th} word of Σ^* is in A. Similarly, $\chi_{A^{\leq n}}$ is the characteristic sequence of $A^{\leq n}$ relative to $\Sigma^{\leq n}$. In both cases Σ is taken as the smallest alphabet containing all the symbols occurring in words of A, so that for a tally set T, χ_T denotes the characteristic sequence of T relative to $\{0\}^*$.

Throughout this paper, $\log n$ means the function $\max(1, \lceil \log_2 n \rceil)$.

We will mention complexity classes defined by computational models; these can either be sequential or exhibit unbounded parallelism in some guise. The sequential classes can be defined in a completely standard way by time-bounded or space-bounded multitape Turing machines, possibly nondeterministic, e.g. classes like P, PSPACE, or NP. Relativizations of these classes are also used; the oracle machine model used for defining them is standard. All these classes are invariant under changes of the machine model, provided that it stays within the so-called First Machine Class [15]: they simulate and are simulated by multitape Turing machines within a polynomial time overhead and a linear space overhead.

Parallel models have in principle more power than the First Class. Many models exist, and not all of them are equivalent. Our parallel models are taken from the so-called Second Machine Class [15]. This class captures a very frequently observed species of parallelism, characterized by the Parallel Computation Thesis: time on these models corresponds, modulo polynomial overheads, to space on First Class models. Prominent members of the Second Machine Class are the alternating Turing machines and the Vector Machines ([12], see also [3]).

The notion of advice function was introduced in [6] to provide connections between uniform computation models such as resource-bounded Turing machines and nonuniform computation models such as bounded-size boolean circuits.

Definition 1 Given a class of sets C and a class of bounding functions F, the class C/F is formed by the sets A such that

$$\forall n \; \exists w \; (|w| \leq h(n)) \; \forall x \; (|x| = n) \\ x \in A \iff \langle x, w \rangle \in B$$

where $B \in C$ and $h \in F$. □

The words w mentioned in the definition are frequently called "advice words". The corresponding Skolem function mapping each n into an appropriate advice w_n for length n is called "advice function". C is usually a uniform complexity class, most frequently P, whereas the class poly $= \{n^k \mid k \in \mathbb{N}\}$ of polynomials and the class log $= \{k \cdot \log n \mid k \in \mathbb{N}\} = O(\log n)$ of logarithms are the most frequent bounding functions.

The class P/poly is known to have a number of interesting characterizations; the most relevant two of them are $\bigcup_T \mathrm{P}(T)$ where T is a tally set and the class of sets A such that for all n the set $A^{=n}$ can be decided by a circuit of size polynomial in n. Several variants corresponding to logarithmic advice can be defined; see [4] and the references there.

Later on in section 5 we introduce additional structural material regarding Kolmogorov complexity.

2.2 Neural Networks

In this work, a *neural network* is a processor network consisting of a finite number of processors, or neurons, each of which has a state whose value at integer times t that can be characterized by a real number. We assume that there are N processors and M external input signals. The state values, or "activations," are updated by equations such as

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

for $i = 1, \ldots, N$. Here $x_i(t)$ and $u_j(t)$ denote the state of neuron i and the value of input line j at time t, respectively. The elements a_{ij}, etc, are called the "weights" of the network, and $\sigma(x) = \max\{\min\{x, 1\}, 0\}$. In vector form, this reads

$$x^+ = \sigma(Ax + Bu + c) \tag{2}$$

where "$x^+(t)$" stands for "$x(t+1)$" and we drop time arguments t. We are letting σ denote the application of σ to each coordinate of x; note that now c is an N-vector, A and B are real matrices of sizes $N \times N$ and $N \times M$ respectively. Given as part of the definition is also a set of indices $i_1, \ldots, i_p$. We think of the processors $x_{i_1}, \ldots, x_{i_p}$ as output processors. For each input sequence $u = u(1), u(2), \ldots$ and initial state $x(1) = \mathbf{0}$, recursively solving equations (2) gives us the state $x(t)$ at time t. Restricting attention to the output processors one gets a corresponding sequence of output values, which we refer to as the output produced by the input u. We assume that $\mathbf{0}$ is an equilibrium state, which amounts to:

$$\sigma(A\mathbf{0} + B\mathbf{0} + c) = \mathbf{0} \; .$$

We now restrict, as in [13], to networks with *two* binary input and output lines. In each case, the first one is a *data line* that carries a binary signal (defaults to zero if there is no signal), and the second one is a *validation line*, used to indicate when the data line is active. The validation signal is "1" while the input is present, and "0" otherwise. Thus we can write $u(t) = (D(t), V(t)) \in \{0,1\}^2$, and similarly for outputs $(O_d(t), O_v(t))$.

We use the following convention to deal with language recognition. We start by encoding each word $a = a_1 \cdots a_k \in \{0,1\}^+$ into an input signal of the form described above, namely: Let

$$u_a(t) = (D_a(t), V_a(t)) \; , \quad t = 1, \ldots \; ,$$

where $V_a(t)$ is 1 if $t = 1, \ldots, k$ and is 0 otherwise, while $D_a(t)$ equals a_k for $t = 1, \ldots, k$ and is 0 otherwise. We say that a word a is *classified in time* τ if the output sequence $y(t) = (O_d(t), O_v(t))$ produced by u_a is of the special type:

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

where η_a is binary.

A language $L \subseteq \{0,1\}^+$ is said to be *accepted in time* T by the network $\mathcal{N}$ if each $a \in \{0,1\}^+$ is classified in time $\tau \leq T(|a|)$, and η_a equals 1 when $a \in L$ and equals 0 otherwise.

The definition given here corresponds to the so-called first-order neural networks, since the computation of each processor is an affine function. Second-order nets are obtained if polynomials (equivalently, multiplication) are allowed to take place in the processors. Time in second-order nets is polynomially related to time in first-order nets [13].

3 Space Classes

This section discusses rational-valued neural nets on which a bound is set on the precision available for the computations. It should be observed that any simulation of a neural net computation, e.g. by implementing a simulation program on a more or less standard computer, will have to obey such a bound. Indeed, efficient implementations of the arithmetic require dedicated hardware, able to handle "reals" of a limited precision seldom larger than 64 bits (and quite frequently smaller). When larger precision is necessary, for instance to process longer inputs, one must resort to a software implementation of real arithmetic (sometimes provided by the compiler), and even in this case a physical limitation on the length of the mantissa of each state of a network under simulation is imposed by the amount of available memory. It is thus important to know the computational consequences of these limitations.

This very same observation suggests that some connection can be traced between the space requirements needed to solve a problem and the precision required on the states of the neural networks that solve them.

Definition 2 A rational neural net works within precision $S(n)$ if and only if all the weights, and all the rational values of the states of the neurons through a computation on an input of length n, can be represented in binary within $O(S(n))$ bits.

We observe here the following:

Theorem 1 *Let $S(n) \geq n$ be a space-constructible function. Then the following are equivalent:*

1. *the set L is accepted by a Turing machine within space $O(S(n))$;*
2. *the set L is accepted by a neural net within precision $O(S(n))$.*

The proof is not difficult along the lines of [13]. However, that proof relies on a preliminary phase through which the input is completely loaded into the state of a specific neuron, before proceeding to the actual computation. This is the reason why we need the condition $S(n) \geq n$, since the precision needed for that neuron will be at least linear. Actually, the proof of theorem 2 below can be used as well to prove this theorem, taking into account that the restrictions imposed there become trivial for at least linear space.

It is quite interesting to see what happens under sublinear precision bounds. The point is that the input convention we have described for neural nets makes available each input symbol only once; moreover, it is available for only a single step, since the next iteration brings a new symbol in.

Thus, nets will correspond weakly to restricted variants of Turing machines, the on-line machines and a still more restricted model called here lr-machines: they move left to right the input head one symbol per each step, and cannot backtrack nor even stay at a symbol more than one step. However, they are allowed to continue working without further reading after exhausting the input. This last period of work uses only the information gathered in the worktapes during the reading. Clearly this restricted model is equivalent to the standard model for at least linear space bounds.

Theorem 2 *Let $S(n)$ be any space-constructible function.*

1. *If a set L is accepted by an lr-machine within space $O(S(n))$, then L is accepted by a neural net within precision $O(S(n))$.*
2. *If a set L is accepted by a neural net within precision $O(S(n))$, then L is accepted by an on-line machine within space $O(S(n))$.*

Note that, unlike the previous and next theorems, we don't have to impose any lower bound on $S(n)$ here. Essentially this corresponds to proving that the intermediate step of loading the input into a single neuron state, as done in [13], is not necessary; but this does not suffice since there the net needs four steps to simulate each step of the Turing machine. A different procedure is necessary to prove that the simulation can be done in real time, i.e. spending only one step of the neural net to simulate each step of the Turing machine; otherwise, input characters would be lost. This new simulation will be described in the full paper, together with some consequences such as a better bound on the size of the smallest universal neural net.

On the other hand, the second part is quite simple, since it consists of a straightforward simulation of the computation of the neural net. The state of each of the fixed number of neurons is kept in worktape, where it fits due to the precision bound. Since the network receives its input in real time, there is never the need of backtracking the input head during the simulation. Observe however that the simulating machine is not an lr-machine since each step of the net requires a nontrivial number of Turing machine steps due to the arithmetic operations to be done.

Off-line space-bounded machines can be proven equivalent to precision-bounded neural nets under a different input convention.

Definition 3 A neural net with cyclic input receives the input w through two input lines as follows: the data line brings in the bits of the input w repeatedly, w^∞, while the validation line brings in $(10^{|w|-1})^\infty$. □

So, the data line brings in $wwwwww\ldots$ and the validation one, instead of marking the end of the whole input, marks the beginning of each cycle. This (admittedly somewhat artificial) input convention gives:

Theorem 3 *Let $S(n) \geq \log n$ be a space constructible function. Then the following are equivalent:*

1. *the set L is accepted by an off-line Turing machine within space $O(S(n))$;*
2. *the set L is accepted by a neural net with cyclic input within precision $O(S(n))$.*

Here we only sketch the proof.

Proof. 1 $\Rightarrow$ 2) The network $\mathcal{N}$ simulating the Turing machine M is built conceptually out of two subnetworks: In a manner similar to that of [13], we construct a constant size subnet that receives as input the bit currently scanned by the input-tape head of the M and the state of M, and returns a new state and the direction to move the input-tape head, right or left. Another neuron keeps a rational that, interpreted as an integer value, indicates the current position of the input-tape head. The value is incremented or decremented depending on the direction of movement. Then another subnet, triggered by the 1 that marks the beginning of each cycle, counts up to the position of the input-tape head to catch the input symbol necessary for the simulation of the next step. With some precomputation time, it is possible to do the counting in real time using only logarithmic precision.

2 $\Rightarrow$ 1) For the backward implication, use the same simulation as for the on-line case. When reaching the right end of the input, stop the simulation, reset the input tape head, and resume it; when the simulating machine is reading the first symbol of the input, it simulates a 1 on the input validation line. ∎

The fact that time-bounded rational nets correspond modulo polynomial-time simulations to time-bounded Turing machines [13], taken together with theorem 1 here, allows us to close this section by pointing out a remark on the "linear precision suffices" lemma of [14]. There it is proved that for a neural net running in time $T(n)$, the net obtained truncating all states to $O(T(n))$ bits is equivalent to it. Their proof is valid for real states; but if we consider its restriction to the simpler rational case, then we can see an interesting intuitive analogy. Through the equivalences with the Turing model, we see that this result corresponds in some sense to the basic theorems relating time-bounded and space-bounded classes, and in particular to the by now elementary result that everything done in time $T(n)$ is done in space $T(n)$ as well. The "linear precision lemma", restricted to the rational case, would be essentially the neural net analog of this result.

4 Parallel Time Classes

It was proven in [14] that second-order nets can be simulated with a polynomial overhead in time by first-order nets. That is, allowing neurons that compute polynomials does not increase the computational power of nets (up to polynomials). In this section we show that, for nets with rational weights, adding both division and bitwise AND makes an enormous difference: that from sequential to *parallel* time.

Thus the nets we consider in this section have processors with either an update equation of the form

$$x_i(t+1) = \sigma\left(\frac{P_i(x_1(t), \ldots, x_N(t))}{Q_i(x_1(t), \ldots, x_N(t))}\right)$$

where P_i and Q_i are polynomials with rational coefficients, or of the form

$$x_i(t+1) = x_{j_1}(t) \wedge \ldots \wedge x_{j_k}(t).$$

where $\wedge$ denotes bitwise AND of binary representations (note that adding σ does not make any difference in this case). We assume that the binary expansion of a non-periodic rational always ends in an infinite sequence of zeros. That is, $1/2$ is represented as $0.10000\ldots$, not as $0.01111\ldots$

We say that a net *works within precision* $p(n)$ if the binary expansion of all weights, and of any state appearing during the computation on an input of length n, is identically zero after the first $p(n)$ digits. Let NN–TIME(t, p) be the set of languages accepted by nets with division and bitwise AND in time $O(t(n))$ and precision $O(p(n))$ simultaneously.

Intuitively, the extra power we get by using division can be demonstrated by the following example. By repeated multiplication a net can build in time $O(t)$ rationals as small as 2^{-2^t}. To recover the first 1-bit of these numbers, a net without division can only multiply at each step by some (constant) weight, and thus needs $2^{\Omega(t)}$ steps. However, a single division can turn this digit into the most significant one.

We use this power of division, and bitwise AND, to simulate a model of unbounded parallelism introduced by Pratt and Stockmeyer, the vector machines ([12], see also [3, 7]).

Vector machines are machines that can make boolean operations and left and right shifts on their potentially infinite registers; these capabilities give them the power of parallel machines. More precisely, a *vector machine* is a processor together with a fixed number of vector registers $V_1, V_2, \ldots, V_r$, each containing bit vectors. These bit vectors are ultimately constant sequences of bits written from right to left, and infinite to the left. The length of a vector register is the length of its nonconstant part. Vectors that are ultimately 0 and ultimately 1 represent non-negative and negative integers respectively. The input is given to the machine in register V_1, and the output is in V_1 when the machine halts. The program for the vector machine can contain the following instructions, assumed to have unit cost:

- $V_i := x$: Load the constant x into V_i.
- $V_i :=$ **not** V_i: Bitwise negate all of V_i.
- $V_i := V_j \wedge V_k$: Bitwise AND V_j and V_k.
- $V_i := V_i \uparrow V_j$: If V_j contains a positive number, shift V_i to the left by V_j positions; new positions are filled with zeros. Otherwise, do nothing.
- $V_i := V_i \downarrow V_j$: If V_j contains a positive number, shift V_i to the right by V_j positions; rightmost bits are discarded. Otherwise do nothing.
- if $V_i = \mathbf{0}$ go to *label.*
- *accept, reject.*

To make vector machines equivalent in power to other Second Class models, we have to impose the following restriction: no register is ever shifted by more than $2^{O(t(n))}$ positions in a single shift instruction, where $t(n)$ is the machine's running time. In other words, arguments V_j in shift instructions always have values $O(t(n))$. We call machines with this property *restricted.* Let VECTOR–TIME(t) be the class of languages accepted by restricted vector machines in time $O(t(n))$.

We now show that, up to polynomials, the classes VECTOR–TIME(t) and NN–TIME$(t, 2^t)$ are equal. To our view, the restriction of shifts in vector machines can be compared to the restriction of precision in the nets.

Theorem 4 *For any* $t(n) \geq n$*,* VECTOR–TIME(t) $\subseteq$ NN–TIME$(t^{O(1)}, 2^{O(t)})$. ∎

Proof. Let M be a restricted vector machine running in time $t(n)$. For a given n, let s be the minimum power of two such that the length of M's registers is always less than s, during the computation on an input of length n. It is easy to prove that $s = 2^{O(t(n))}$ (the restriction on the shifts is necessary here).

To simulate M by means of a neural net, we encode the contents of each register V_i of M as the activation value of a net processor v_i. More precisely, if V_i contains the vector $\ldots 000 b_\ell b_{\ell-1} \ldots b_2 b_1$, then $v_i = 0.\underbrace{000\ldots000 b_\ell b_{\ell-1}\ldots b_2 b_1}_{s} 000\ldots$, and if V_i contains $\ldots 111 b_\ell b_{\ell-1} \ldots b_2 b_1$, then $v_i = 0.\underbrace{111\ldots111 b_\ell b_{\ell-1}\ldots b_2 b_1}_{s} 000\ldots$ Note that $0 \leq v_i < 1$, and that $v_i \geq 1/2$ if and only if $V_i < 0$.

Initially, the net reads the input and stores it as the state of v_1, as described in [13]. For the actual computation, we divide the proof in two parts: First, we show that the effect of each vector instruction can be simulated by rational functions and bitwise ANDs in time polylogarithmic in $s = 2^{O(t(n))}$, i.e., polynomial in $t(n)$. Then, we show that these sequences of operations, as well as the finite control of the vector machine, can be programmed in a neural net.

We simulate each vector instruction as follows:

- $V_i := x$: The constant x is built into the network as a weight, and this instruction sets $v_i := x$.
- $V_i :=$ **not** V_i: Build the rational 2^{-s} as described below for the shift instructions. Then set $v_i := 1 - 2^{-s} - v_i$.
- $V_i := V_j \wedge V_k$: This is simulated by a bitwise AND of v_j and v_k.

```
/* compute y = 2^{-x}, where x = x_{s-1}...x_0 is given as a real x' = 0.x_{s-1}...x_0 */
p := 1/2;
for i := 1 to log s do
        p := p^2;
        /* p = 2^{-s} here; recall that s is a power of 2 */
y := 1; z := 1/2;
while x'/p >= 1 do begin                /* digits left in x' */
        /* ∃i  (p = 2^{-s+i} ∧ z = 2^{-2^i} ∧ y = 2^{-(x_{i-1}...x_0)}) */
        if (x' ∧ p)/p = 1 then           /* x_i = 1 */
                y := y * z;
        p := 2 * p;
        z := z^2
end
```

Figure 1: Computing 2^{-x}

- $V_i := V_i \uparrow V_j$. This is simulated as:

  if $v_j < 1/2$ (i.e., $V_j \geq 0$) **then begin**
  build $y = 2^{-v_j}$;
  $v_i := v_i/y$
  end

  Testing condition "$v_j < 1/2$" is knowing whether $\sigma(4(v_j - 1/2))$ is 0 or 1. To compute 2^{-v_j} we use the algorithm given in the figure, which works in time $O(\log|V_j|)$. Because M is restricted, $|V_j|$ is $2^{O(t(n))}$ and thus the computation takes time $O(t(n))$.

- $V_i := V_i \downarrow V_j$. Similar to left shift using product times 2^{-v_j} instead of division.

- if $V_i = \mathbf{0}$ go to *label:* Compute 2^{-s} as above and then test whether $\sigma(v_i/2^{-s}) = 0$. Note that $\sigma(v_i/2^{-s}) = 1$ for all possible contents of V_i except for 0.

- *accept, reject:* To simulate these instructions, the net sets to 1 the output validation line and to 1 or 0, respectively, the output data line.

It remains to show that sequencing all these instructions can be hardwired into a network. Here we only provide an example: a subnet that implements the computation of $p = 2^{-s}$ following the first loop of the algorithm in the figure.

This network is triggered by the input a; it outputs its data via the neuron p and validates the output via the neuron v.

- The binary input a is 0 except for once. When 1, it triggers the network described below.

- The output validation neuron v is set to 1 $\log(s)$ steps after a triggers.

- The output data neuron p contains the value 2^{-s} when $v = 1$.

The internal neuron c counts the time. We assume that some neuron ℓ in the rest of the net provides the reciprocal of s, the maximum length that a register can have. For example, if $s = 32$, ℓ contains binary 0.00001 (recall that s is a power of two).

The update equations of the processors are:

$p^+ := \sigma(\ a/2 + (1-a)p^2\ )$
/* $a = 1$ resets p to $1/2$, $a = 0$ squares it */
$c^+ := \sigma(\ a \cdot \ell + (1-a) \cdot 2c\ )$
/* $a = 1$ resets counter to $1/s$ */
$v^+ := \sigma(\ 4c - 1\ )$
/* $v^+ = 0$ for $c \leq 1/4$, $v^+ = 1$ for $c \geq 1/2$ */

∎

Theorem 5 *For any $t(n) \geq n$,* NN–TIME$(t, 2^t) \subseteq$ VECTOR–TIME$(t^{O(1)})$.

Proof. Consider a net running in time $t(n)$ and within precision $2^{t(n)}$. To simulate the net by a vector machine, we keep the state of each processor in a vector register of length $2^{t(n)}$. Remember that addition, product, division, and bitwise AND of m-bit numbers can be computed in parallel machines in time $(\log m)^{O(1)}$ and $m^{O(1)}$ memory (see for example [7]). Thus, updating the state of each processor at each simulated step needs $t(n)^{O(1)}$ time and $2^{O(t(n))}$ memory on the vector machine. ∎

In fact, the simulations show that amount of memory in vector machines is polynomially related to net precision. The theorems were stated for at least linear running time, as the networks need linear time to read the input. However, the simulations work even for sublinear running times $t(n) \geq \log n$, if we adopt an alternative convention that the input is given to the net as the initial state of one of the processors, as in theorem 2 of [13]. Then we can characterize NC, the class of sets accepted by Second Class machines in polylog time and polynomial space, as NN–TIME($\log^{O(1)} n, n^{O(1)}$).

Time for both models is still polynomially related in the presence of nonuniformity, that is, when vector machines are nonuniform and nets have real instead of rational weights. We discuss this in more detail in section 5.

5 Nonuniform Classes

5.1 Real weights and circuit depth

In section 4 we have considered nets whose processors can compute rational functions and bitwise ANDs on their inputs, and shown that time in these nets is equivalent to time on parallel machines. If we allow real instead of rational weights, their power changes accordingly: we obtain nonuniform parallel time, or, equivalently, nonuniform circuit depth. For example, one can obtain the following analog of the fact that nonuniform circuits of bounded fan-in and linear depth can decide any set.

Theorem 6 *Every language is decided in linear time by a net with real weights whose processors compute rational functions and bitwise ANDs.*

Proof. The net contains a real weight whose binary expansion is the characteristic sequence of the language to decide. On an input that has lexicographical number i, the net computes the real $x = 2^{-i}$ using multiplication; it can do this as the input is entering. When the input ends, the net ANDs x with the real encoding the set, and divides the result by x, thus determining whether the input is in the set or not. ∎

Note that, in fact, the net has the answer two steps after the input has been read.

5.2 Kolmogorov Weights: Between P and P/poly

As said in the introduction, in [13] and [14] Siegelmann and Sontag showed that the computational power of neural networks depends on the type of numbers utilized as weights. They investigated the computational power of networks in which either rationals or reals are involved. When the networks compute in polynomial time, the computational power of these networks happens to coincide with the classes P and P/poly, respectively.

Here, we concentrate on weights from various classes of computable numbers, characterized in an information-theoretic manner. We discover an infinite hierarchy of computational classes of networks with such weights — while still complying with the polynomial computation time constraint. This result is maybe surprising as different neural network models were traditionally considered as equivalent to finite automata, Turing machines, or unlimitedly powerful models.

We define different classes of computable numbers [1] by considering different time constraints and amounts of information in their construction. Our definition of Kolmogorov complexity of infinite sequences is a time-bounded analog of that in [8].

Definition 4 Fix a universal Turing machine U. Let f be a nondecreasing function, g a time-constructible function, and $\alpha \in \{0,1\}^\infty$. We say that $\alpha \in K[f(n), g(n)]$ if there exists $\beta \in \{0,1\}^\infty$ such that, for every n, the universal machine U outputs $\alpha_{1:n}$ in time $g(n)$, when given $\beta_{1:f(n)}$ and n as inputs. If no condition is imposed on the running time, we say $\alpha \in K[f(n)]$.

Observe that here the length of the output is provided for free to the universal machine; so our definition corresponds to the usually named complexity relative to the length. The reason is that we want simple numbers (e.g. rationals) to have extremely low complexity (e.g. constant), and the information contained in the length of a string could be higher. However, the definitions are equivalent (modulo constants) for complexities at least logarithmic.

Generally, $K[\mathcal{F}, \mathcal{G}]$ is the set of all infinite binary sequences taken from $K[f,g]$ where $f \in \mathcal{F}$ and $g \in \mathcal{G}$ For example, a sequence is in $K[$*log,poly*$]$ if its prefixes are computable from logarithmically long prefixes of some other sequence in polynomial time.

In the following, we denote by $\{0,1\}^\#$ the set of both finite and infinite binary strings.

Define a function

$$\delta_4 : \{0,1\}^\# \to [0,1]$$

by the formula

$$\delta_4(\epsilon) = 0 \qquad \delta_4(\alpha) = \sum_{i=1}^{|\alpha|} \frac{2\alpha_i + 1}{4^i} \ .$$

Here ϵ is the empty string; $|\alpha|$ is the length of the string α, and can be either a finite value or ∞; α_i is the ith bit of the string α. Let Δ_4 be the range of this function. That is,

$$\Delta_4 \equiv \left\{ \sum_{i=1}^{n} \frac{\beta_i}{4^i} \right\}_{n \in \mathbb{N} \cup \{0\} \cup \{\infty\}} , \qquad \beta_i \in \{1,3\} \ .$$

The map δ_4 is injective in $\{0,1\}^\# \mapsto \Delta_4$ and δ_4^{-1} is well defined there. Thus, it can be used to define the Kolmogorov complexity of numbers in Δ_4: A number $\omega \in \Delta_4$ is said to be in $K[f(n), g(n)]$ iff $\delta^{-1}(\omega) \in K[f(n), g(n)]$.

The main contribution of this section is to show that the Kolmogorov complexity of the weights of a net is also related to a structural notion: the amount of advice for nonuniform classes. Important consequences follow; for instance, we can prove the following "hierarchy" theorem:

Theorem 7 *Let $\mathcal{F}, \mathcal{G}$ be function classes such that $\exists s \in \mathcal{G}, s \in o(n)$ such that $\forall p \in$ poly, $\forall r \in \mathcal{F}, r(p(n)) \in o(s(n))$. Let $\mathcal{N}_{K[\mathcal{F},poly]}$ be the set of networks that compute in polynomial time, and each of which uses weights from $K[\mathcal{F}, poly] \bigcup \mathbb{Q}$. Let $\mathcal{L}(\mathcal{N}_{K[\mathcal{F},poly]})$ be the class of languages accepted by $\mathcal{N}_{K[\mathcal{F},poly]}$. Then:*

$$\mathcal{L}(\mathcal{N}_{K[\mathcal{F},poly]}) \neq \mathcal{L}(\mathcal{N}_{K[\mathcal{G},poly]}) \ .$$

∎

In subsection 5.3, we prove the equivalence between an infinite subset of oracle TMs and networks. We show in subsection 5.4 the hierarchy in the discussed subset of oracle TMs, thus concluding theorem 7.

5.3 Equivalence of TMs with Tally Oracles and NNs

Definition 5 Let $S \subseteq \{0,1\}^\#$. S is *closed under mixing* if for any finite number $k \in \mathbb{N}$ and for any k strings from S,

$$\begin{aligned} \alpha^1 &= \alpha_1^1 \alpha_2^1 \alpha_3^1 \cdots , \\ \alpha^2 &= \alpha_1^2 \alpha_2^2 \alpha_3^2 \cdots , \\ \cdots &, \\ \alpha^k &= \alpha_1^k \alpha_2^k \alpha_3^k \cdots \end{aligned}$$

the shuffled string

$$\alpha_1^1 \alpha_1^2 \alpha_1^3 \cdots \alpha_1^k \, \alpha_2^1 \alpha_2^2 \alpha_2^3 \cdots \alpha_2^k \, \alpha_3^1 \alpha_3^2 \cdots$$

is again an element of S.

Definition 6 Let $S \subseteq \{0,1\}^\#$. We define the *fraction set* of S to be

$$\tilde{S}_4 = \{\omega \in \Delta_4 \mid \exists \alpha, \quad \omega = \delta_4(\alpha) \text{ and } \alpha \in S\}$$

Definition 7 Let $T \subseteq \{0,1\}^\star$. We define the *characteristic number* of T as

$$T_4 = \delta_4(\chi_T) \in \Delta_4 \ ,$$

where χ_T is the characteristic string of T.

The main theorem of this subsection is as follows:

Theorem 8 *Let $S \subseteq \{0,1\}^\infty$ be closed under mixing and $\mathcal{T}$ the class of tally sets $\mathcal{T} = \{ T : \chi_T \in S \}$. Time in the following models is polynomially related:*

1. *Oracle Turing machines that consult oracles in $\mathcal{T}$.*
2. *Neural networks that have all weights in the set $\tilde{S}_4 \bigcup \mathbb{Q}$.*

Before proving this theorem, we look at some consequences for polynomial time machines. The following classes Pref-C/H were defined in [4]: Given a class of sets C and a class of bounding functions H, the class Pref-C/H is formed by the sets A such that

$$\begin{aligned} &\forall n \, \exists \omega_n \, (|\omega_n| \leq h(n)) \, \forall x \, (|x| \leq n) \\ &x \in A \iff <x, \omega_n> \in B \end{aligned}$$

where $B \in C$ and $h \in H$ and, for all $n \leq m$, ω_n is a prefix of ω_m. Observe that advice w_n must be correct for all strings of length up to n, not only those of length n. Note also that Pref-P/poly = P/poly, but that a similar equality may not hold for smaller function classes.

We also say that a class $\mathcal{F}$ of functions is *closed under $O(.)$* if for every f, g, if $g \in O(f)$ and $f \in \mathcal{F}$, then $g \in \mathcal{F}$.

Corollary 8 Let $\mathcal{F}$ be a class of nondecreasing functions closed under $O(.)$, and $\mathcal{L}(\mathcal{N}_{K[\mathcal{F},\mathrm{poly}]})$ be the class of languages accepted by networks with weights in $K[\mathcal{F},poly] \cup \mathbb{Q}$. Then,

$$\text{Pref-P}/\mathcal{F} = \mathcal{L}(\mathcal{N}_{K[\mathcal{F},\mathrm{poly}]}) \ .$$

Some interesting special cases arise when considering various natural bounds for the Kolmogorov complexity:

- $S = K[n, \mathrm{poly}]$, that is, arbitrary strings. The class of languages accepted in this case is P/poly: this is the main result of [14].
- $S = K[1, \mathrm{poly}]$, that is, the sets of strings computable in polynomial time. The class of languages accepted in this case is P.
- $S = K[\log, \mathrm{poly}]$. In this case, the class of languages accepted is Full-P/log, described in [4].

The next two subsubsections prove theorem 8.

5.3.1 Proof: $1 \subseteq 2$

Definition 9 An oracle neural network (ONN) is a network $\mathcal{N}$ with additional three special oracle neurons Q, A, W — called query, answer, and wait neurons — and a particular *oracle number* Y. The above receive their values in

$$Q \in \Delta_4, \quad A \in \{0,1\}, \quad W \in \{0,1\}, \quad Y \in \Delta_4 \ .$$

- The network operates regularly as long as $W = 0$. When $W = 1$, the activations in the network $\mathcal{N}$ are not being changed.
- The network can set W to 1 but cannot reset it.
- When $W = 1$, the three oracle neurons change as follows:

  $$\begin{aligned} &A \leftarrow (\delta_4^{-1}(Y))_{\mathrm{lex}(Q)} \\ &Q \leftarrow 0 \\ &W \leftarrow 0 \end{aligned}$$

  where $\mathrm{lex}(Q)$ is the lexicographic index of Q in Δ_4. Other neurons of $\mathcal{N}$ do not change.

Setting $W = 1$ is like invoking a subroutine for solving a membership query. Y can be thought of as the characteristic number of an oracle set Y', and the subroutine tests whether $\delta_4^{-1}(Q) \in Y'$. However, we assume that this oracle subroutine answers in unit time.

Lemma 10 Let T be a tally set. Time in the following models is polynomially related.

- Oracle TM that consults the tally set T.
- Oracle NN with all weights in $\mathbb{Q}$ and oracle number T_4.

The proof of this lemma is very similar to the proof of the main result in [13], and is not included here.

Lemma 11 For each number $T_4 \in \Delta_4$, there exists a network of five neurons and two inputs — u_1, u_2 — that started from the zero initial value, and given the input signals

$$\begin{aligned} u_1 &= [\sum_{i=1}^{n} (\frac{1}{4})^i] \ 0 \ 0 \ 0 \ \cdots \\ u_2 &= 1 \ 0 \ 0 \ 0 \ \cdots \end{aligned}$$

the network outputs

$$\underbrace{0\,0 \cdots 0}_{n+1} \ b \ 0 \ 0 \ \cdots$$

where b is the truth value of $\delta_4^{-1}(u_1(1)) \in T$, for the set T that has T_4 as its characteristic number.

Proof. We use T_4 as one of the weights of the network.

Notice that

$$\begin{aligned} u_1(1) &= .\underbrace{11\cdots1}_{n} && \text{in base 4} \ , \\ T_4 &= .3133113\cdots && \text{in base 4} \end{aligned}$$

and the nth digit of T_4 in base 4 expansion is the decision of whether $\delta_4^{-1}(\sum_{i=1}^{n}(\frac{1}{4})^i) \in T$.

The network simultaneously scans the value given in T_4 [in x_1 and x_2] and the value of $u_1(1)$ [in x_3 and x_4]. When it reaches the last digit '1' of $u_1(1)$, the network returns the currently scanned digit in the base 4 expansion of T_4.

$$\begin{aligned} x_1^+ &= \sigma(u_2(1+T_4) + 4x_1 - 2x_2 - 1) \\ x_2^+ &= \sigma(4u_2(1+T_4) + 16x_1 - 8x_2 - 6) \\ x_3^+ &= \sigma(u_1 + 4x_3 - 1 + u_2) \\ x_4^+ &= \sigma(4u_1 + 4x_4 - 1) \\ x_5^+ &= \sigma(4x_3 - 1 + x_2 - 16x_4) \ . \end{aligned}$$

∎

Using the above two lemmas, we can prove the inclusion $1 \subseteq 2$ as follows: let M be an OTM that uses a tally set T as an oracle, where $\chi_T \in S$. We construct a network $\mathcal{N}$ that accepts the same language and has all weights in $\tilde{S}_4 \bigcup \mathbb{Q}$. The network $\mathcal{N}$ consists of two subnetworks: $\mathcal{N}_1$ is an oracle network that consults the number T_4, and $\mathcal{N}_2$ is the retrieval network of T_4 as described in lemma 11. By lemma 10, $\mathcal{N}_1$ has only rational weights, and by lemma 11, $\mathcal{N}_2$ has both rational weights and the weight $T_4 \in \tilde{S}_4$. $\mathcal{N}_1$ simulates M in linear time [13], while $\mathcal{N}_2$ has a total computation time bounded by $O(\sum |\text{queries}|)$ — which is bounded by the computation time of M. Thus, given a OTM with an oracle in S, there is a corresponding neural network whose weights are either rationals or in the set $\tilde{S}_4$ that computes the same as the TM with no more than linear slowdown in the computation.

(To couple the OTM network with the retrieval network:

We add the neurons

$$\begin{aligned} t_1 &= \sigma(Q+W-1) \\ t_1' &= \sigma(t_1) \\ t_1'' &= \sigma(t_1 - t_1') \\ t_2 &= \sigma(W - t_1) \\ t_3 &= \sigma(4x_3 - 16x_4)\ , \end{aligned}$$

where t_1'', t_2 are used as the input u_1 and u_2 of the retrieval network. The neuron t_3 is used to update the dynamics of the oracle neurons.

$$\begin{aligned} W &\leftarrow \sigma(\ldots - C_1 t_3) \\ A &\leftarrow \sigma(\ldots + C_2(2x_5 + t_3 - 2)) \\ Q &\leftarrow \sigma(\ldots - C_3 t_3)\ , \end{aligned}$$

where "..." represents the previously used values of the neurons, and C_1, C_2, C_3 are constants.)

5.3.2 Proof: $2 \subseteq 1$

Given a network $\mathcal{N}$ with weights in $\tilde{S}_4 \bigcup \mathbb{Q}$. The network has a fixed number $k' \in \mathbb{N}$ of weights, which can be written in base four expansion as:

$$\begin{aligned} &\omega^1 = .\omega_1^1 \omega_2^1 \omega_3^1 \cdots\ , \\ &\omega^2 = .\omega_1^2 \omega_2^2 \omega_3^2 \cdots\ , \\ &\ldots\ , \\ &\omega^{k'} = .\omega_1^{k'} \omega_2^{k'} \omega_3^{k'} \cdots\ . \end{aligned}$$

Assume w.l.o.g. that the first k of them are in $\tilde{S}_4$, that is, $\omega_j^i \in \{1,3\}$ for such weights i. (The weights $\omega^{k+1}, \ldots, \omega^{k'}$ are rationals.) As S is closed under mixing, the string

$$\alpha = \tfrac{\omega_1^1-1}{2}\tfrac{\omega_1^2-1}{2}\tfrac{\omega_1^3-1}{2}\ldots\tfrac{\omega_1^k-1}{2}\tfrac{\omega_2^1-1}{2}\tfrac{\omega_2^2-1}{2}\tfrac{\omega_2^3-1}{2}\ldots \\ \ldots\tfrac{\omega_2^k-1}{2}\tfrac{\omega_3^1-1}{2}\tfrac{\omega_3^2-1}{2}\tfrac{\omega_3^3-1}{2}\ \ldots$$

is again an element of S.

We show an oracle TM M that consults a tally set with characteristic string $\chi_T = \alpha$, and simulates the network $\mathcal{N}$ while keeping the polynomial time constraint.

1. M receives the input string x.
2. M computes the running time $B(|x|)$ of $\mathcal{N}$.
3. For a certain constant C, M executes:

   For $i = 1$ to $kCB(|x|)$
   query the ith word of α.

   Now, M has the weights of $\mathcal{N}$ up to a precision $CB(|x|)$. C is a constant such that this precision suffices. The existence of such a C was proved in [14]. (The $(k'-k)$ rational weights are encoded in the machine M.)
4. M simulates $\mathcal{N}$ step by step in polynomial time.

5.4 Hierarchy of TMs That Consult Tally Oracles

Theorem 9 *Let $\mathcal{F}, \mathcal{G}$ be function classes such that $\exists s \in \mathcal{G}, s \in o(n)$, and for every polynomial p and every $r \in \mathcal{F}$, $r(p(n)) \in o(s(n))$. Let $\bigcup_T P(T, \mathcal{F})$ be a class of TMs that compute in polynomial time, where each TM consults a tally set T such that $\chi_T \in K[f, poly]$, $f \in \mathcal{F}$. Define $\mathcal{L}(\bigcup_T P(T, \mathcal{F}))$ as the class of languages computed by these TMs. Then: $\mathcal{L}(\bigcup_T P(T, \mathcal{F}))$ and $\mathcal{L}(\bigcup_T P(T, \mathcal{G}))$ are different.*

Proof. We define a set $\mathcal{A} \in \mathcal{L}(\bigcup_T P(T, \mathcal{G}))$ but not in $\mathcal{L}(\bigcup_T P(T, \mathcal{F}))$. Let $s(n)$ be as in the theorem. Choose an infinite sequence $\gamma \notin K[n/2]$. For every n define string β_n as $\beta_n = \gamma_{1:s(n)/2} \cdot 0^{n-s(n)/2}$ if $n \geq s(n)/2$, and $\beta_n = 0^n$ otherwise.

Let $\mathcal{A}$ be the tally set with characteristic string $\beta_1\beta_2\beta_3\ldots$ Given $\gamma_{1:s(n)/2}$ it is easy to build $\chi_{\mathcal{A}\leq n}$, so $\chi_{\mathcal{A}} \in K[s(n)/2 + c, q(n)] \subseteq K[s(n), q(n)]$, for some constant c and polynomial q. Hence, $\mathcal{A} \in \mathcal{L}(\bigcup_T P(T, \mathcal{G}))$.

However, $\mathcal{A} \notin \mathcal{L}(\bigcup_T P(T, \mathcal{F}))$. Assume otherwise, then there is some machine that prints $\gamma_{1:s(n)/2}$ in time $p_1(n)$, querying at most the first $p_1(n)$ elements of a tally set T, with $\chi_T \in K[r(n), p_2(n)]$, p_1, p_2 polynomials, and $r \in \mathcal{G}$. Then

$\gamma_{1:s(n)/2}$ is obtained from the first $r(p_1(n))+O(1) < s(n)/4$ bits of χ_T, in fact in time $O(p_2(p_1(n)))$. This contradicts the choice of γ. ∎

The combination of theorem 8 and theorem 9 proves theorem 7.

References

1. Aberth O., *Computable Analysis,* McGraw-Hill, 1980.
2. Balcázar J.L., J. Díaz, and J. Gabarró, *Structural Complexity I*, Springer-Verlag EATCS Monographs vol. 11, 1988.
3. Balcázar J.L., J. Díaz, and J. Gabarró, *Structural Complexity II*, Springer-Verlag EATCS Monographs vol. 22, 1990.
4. Balcázar J.L., M. Hermo, and E. Mayordomo, "Characterizations of logarithmic advice complexity classes", in *Information Processing 92*, vol. I, IFIP Transactions A-12, North-Holland, 1992, 315–321.
5. 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.
6. Karp R.M. and R.J. Lipton, "Some connections between uniform and nonuniform complexity classes", in *Proc. 12th ACM Symp. on Theory of Computing*, 1980, 302–309.
7. Karp R.M. and V. Ramachandran, "Parallel algorithms for shared-memory machines", in *Handbook of Theoretical Computer Science*, vol. A, MIT/Elsevier, 1990, 869–941.
8. Kobayashi K., "On compressibility of infinite sequences", Research Report C-34, Department of Information Sciences, Tokyo Institute of Technology, 1981.
9. McCulloch W.S. and W. Pitts, "A logical calculus of the ideas immanent in nervous activity", *Bull. Math. Biophys.* 5, 1943, 115–133.
10. Orponen P., "Neural networks and complexity theory", in *Proc. 17th Symposium on Mathematical Foundations of Computer Science*, 1992, 50–61.
11. Parberry I., "A primer on the complexity theory of neural networks", in *Formal Techniques in Artificial Intelligence: a Sourcebook*, North-Holland, 1990, 217–268.
12. Pratt V.R. and L.J. Stockmeyer, "A characterization of the power of vector machines", *Journal of Computer and System Sciences* **12**, 198–221 (1976).
13. Siegelmann H.T. and E.D. Sontag, "On the computational power of neural nets," in *Proc. Fifth ACM Workshop on Computational Learning Theory*, Pittsburgh, July 1992, 440-449.
14. Siegelmann H.T. and E.D. Sontag, "Neural networks with real weights: analog computational complexity," Report SYCON 92-05, Rutgers Center for Systems and Control, September 1992. Submitted for publication.
15. Van Emde Boas P., "Machine models and simulations", in *Handbook of Theoretical Computer Science*, vol. A, MIT/Elsevier, 1990, 1–66.