Some structural complexity aspects of neural computation

J. Balcázar
R. Gavaldá
H. Siegelmann
E. Sontag

Report LSI–92–32–R

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[†]
Department of Computer Science
Rutgers University, New Brunswick, NJ 08903

Eduardo D. Sontag[†]
Department of Mathematics
Rutgers University, New Brunswick, NJ 08903

Conference submission: Extended abstract with three technical appendices

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.

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

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 both a practically important and 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 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 [8]. 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 [9] and [10] provide a precise account of these characterizations. 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. Linear saturation being 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 [12] 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 the tape alphabet and state set of small universal Turing machines: 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. (This observation has been made also by Orponen.) 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 on-line machines require us to present a more efficient simulation than that of [12]. We prove here that a neural net can simulate a Turing machine in real time. Again, as a consequence of the new construction, we obtain a universal neural net with 886 neurons.

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 [14]) *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.

Let us discuss this a little further. A crucial lemma proved in [13] indicates that for affine or even polynomial computations inside the neurons, and even for real-valued weights and states, linear precision suffices in the sense that if all values are truncated after $O(T)$ bits, where T is the running time, then no computational power is lost. It is in this sense that the net cannot really take much advantage of manipulating infinite objets like the reals. An essential difference caused by division and bitwise AND is that linear precision no longer suffices, and the net can compute in constant time certain arithmetic operations on reals with exponentially large relevant bits. This is the source of second class power.

We also consider this 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 [13]: 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-bounded 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 [13].

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 [13] 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 alphabet $\Sigma = \{0,1\}$ 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 [14]: 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 [14]. 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 ([11], 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 2.1 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 $= \{n^k | k \in \mathbb{N}\}$ of polynomials and the class log $= \{k \cdot \log n | k \in \mathbb{N}\} = O(\log n)$ 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 as $\bigcup_T \mathrm{P}(T)$ where T is a tally set, and as 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.

Two variants corresponding to logarithmic advice can be defined. We are interested here in the only one of them closed under polynomial-time Turing reducibility; for facts about this class and its motivation, see [4] and the references there.

Definition 2.2 A set A is in Full-P/log if $\forall n\ \exists w\ (|w| \leq c \log n)\ \forall x\ (|x| \leq n)\ x \in A \iff \langle x, w\rangle \in B$ where $B \in \mathrm{P}$ and c is a constant. □

Note that now the advice corresponding to a length is valid also as an advice for all the smaller lengths.

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, 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

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

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 $x(1) = 0$, recursively solving the 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 0 is an equilibrium state, which amounts to:

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

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

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

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 is = 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. The computational power of second-order nets is polynomial time related to that of first-order nets [12].

3 Space-bounded 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 3.1 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 [12]. 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 precisiön 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, it 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, it is 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 these restricted models are equivalent to the standard model for at least linear space bounds.

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

1. *If a set L is accepted by a 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 $O(S(n))$.*

Essentially this corresponds to proving that the intermediate step of loading the input into a single neuron state, as done in [12], 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 is described in the Appendix 6, 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 a *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 3.2 A neural net with cyclic input receives the input w through two input lines as follows: one line brings in the bits of the input w repeatedly, w^∞, while the second one brings in $(10^{|w|-1})^\infty$. □

So the actual input line brings in $wwwww\ldots$ and the second 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 $S(n)$ 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. The cyclic input convention makes this simulation easy: each step of the machine is simulated by a constant size subnet in a manner similar to that of [12]. It also provides a rational value that, interpreted as an integer, indicates the position of the input tape head. Then another subnet, triggered by the 1 that marks the beginning of each cycle, counts up to that position to catch the input symbol necessary for the simulation of the next step.

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; a 1 is simulated on the auxiliary input line when reading the first symbol of the input. ∎

The fact that time-bounded rational nets correspond modulo polynomial time simulations to time-bounded Turing machines [12], taken together with theorem 1 here, allow us to close this section by pointing out a remark on the "linear precision suffices" lemma of [13]. 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. Actually, 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

It was proven in [13] 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 division and bitwise-ANDs makes an enormous difference: that from sequential to *parallel* time.

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 $O(t)$ reals 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-ANDs, to simulate a model of unbounded parallelism introduced by Pratt and Stockmeyer, the vector machines ([11], see also [3, 7]).

Vector machines are parallel machines that can make boolean operations and left and right shifts on their registers (a description can be found in Appendix 7). 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 $O(t(n))$ positions in a single shift instruction, where $t(n)$ is the machine's running time. We call machines with this property *restricted*. Let VECTOR-TIME(t) be the class of languages accepted by restricted vector machines in time $O(t(n))$.

Consider now neural nets with rational weights in which each neuron can compute rational functions and bitwise-ANDs on its inputs (see Appendix 7 for a precise definition). Let NN-TIME(t, p) be the set of languages accepted by these nets in time $O(t(n))$ where each neuron works within precision $O(p(n))$.

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

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

(A sketch of proof can be found in Appendix 7. The main idea is to use product and division to simulate left and right shifts, respectively.).

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

Proof. Consider a neural net $\mathcal{N}$ running in time $t(n)$ and whose neurons work within precision $2^{t(n)}$. To simulate the net by a vector machine, we keep each neuron state in a vector register of length $2^{t(n)}$. Remember that addition, product, division, and bitwise-AND of m-bit numbers can be done in parallel machines in time $(\log m)^{O(1)}$ and $m^{O(1)}$ memory (see for example [7]). Thus, updating the state of each neuron 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 neuron 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 neurons, as in theorem 2 of [12]. Then we can characterize NC, the class of sets accepted by parallel 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 neurons 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 neurons 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 [12] and [13] 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 happen 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 machine, or strong non-uniform models.

We define different classes of computable numbers [1] by considering different time constraints and allowing for advice strings of different sizes. We adapt the Kolmogorov complexity of strings used in [4]:

Definition 5.1 Fix a universal Turing machine U. Let f be a function, g is a time constructible function, and $s \in \{0,1\}^*$. We say that $s \in K[f(n), g(n)]$ if there exists $d \in \{0,1\}^*$, $|d| \leq f(|s|)$, such that, given string d and the size $|s|$, the universal machine U outputs s in time $g(|s|)$. If no condition is imposed on the running rime, we say $s \in K[f(n)]$.

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. A simple counting argument shows that if f is sublinear then not all words are in $K[f(n), g(n)]$.

Generally, $K[\mathcal{F},\mathcal{G}] = \{K[F,G] \mid F \in \mathcal{F}, G \in \mathcal{G}\}$ is a class of sets of strings. For example, $K[\log, \text{poly}]$ is a class of sets, where the strings of each set are computable with logarithmically long advice and polynomial time. More precisely, $A \in K[\log, \text{poly}]$ if there exist constants c and k such that $A \subseteq K[c \log n, n^k]$. We can generalize the definition to infinite strings as follows:

Definition 5.2 Denote by $\{0,1\}^\infty$ the set of infinite binary strings, and let $\alpha \in \{0,1\}^\infty$. We say that $\alpha \in K[f(n), g(n)]$ if $\forall n, \alpha_n \in K[f(n), g(n)]$ (where α_n is the prefix of length n of α).

In order to be able to talk of the Kolmogorov complexity of a real number, define an injective function $\delta_3 : \{0,1\}^\infty \to [0,1]$ by the formula

$$\delta_3(\alpha) = \sum_{i=1}^{\infty} \frac{2\alpha_i}{3^i} \ .$$

Then a number $\omega \in [0,1]$ 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 the 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 $K[F, poly]$. We let $\mathcal{L}(\mathcal{N}_{K[\mathcal{F},poly]})$ be the class of languages accepted by $\mathcal{N}_{K[\mathcal{F},poly]}$. Then:*

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

∎

The proof is given in Appendix 8. There it is proved also that, under a simple technical closure condition, the following relationship can be traced between neural nets whose weights have bounded Kolmogorov complexity and reducibility to tally sets (a more formal statement is given in Appendix 8, where it is proved):

Theorem 8 *Let $S \subseteq \{0,1\}^\infty$ closed under mixing. The following classes are polynomially time related:*

1. *The subset of $\bigcup_T B(T)$ consisting of those machines that consult tally sets T and having the property that, for each $n \in \mathbb{N}$, the characteristic string χ_T of the tally set satisfies that $\chi_{T \leq n} \in S$.*
2. *The class of neural networks whose weights are of complexity defined by S and that compute in time $B \in \mathcal{B}$.*

Some interesting special cases arise when considering polynomial time machines and various natural bounds for the Kolmogorov complexity:

- $S = K[n, \text{poly}]$, that is arbitrary strings. The class of languages accepted in this case is P/poly: this is the main result of [13].
- $S = K[1, \text{poly}]$, that is the sets of strings computable in polynomial time. The class of languages accepted in this case is P.
- $S = K[\log, \text{poly}]$. In this case, the class of languages accepted is Full-P/log, described in [4].

Appendix 8 contains additional material on structural complexity needed to complete the proof of these two theorems.

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, North-Holland IFIP Transactions A-12, 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.

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

[9] Orponen P., "Neural networks and complexity theory", in *Proc. 17th Symposium on Mathematical Foundations of Computer Science*, 1992, 50–61.

[10] Parberry I., "A primer on the complexity theory of neural networks", in *Formal Techniques in Artificial Intelligence: a Sourcebook*, North-Holland, 1990, 217–268.

[11] 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).

[12] 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.

[13] 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.

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

6 Appendix: Real-Time Turing Machine Simulation

We show how to simulate a TM in real-time by an RNN. We construct the network in three phases. First we show how to simulate a Turing machine with a two level NN. Then we modify the construction into two levels, where in one of the levels, neurons compute linear combinations of their input with no sigma function applied to the combinations. In the third phase, we show how to modify the last one into a NN of one level only.

6.1 Two Level RNN Simulates TM

In this section, we go over the proof in [12] that constructed a four-level NN that computed as a Turing machine. By careful checking, we see that indeed only two levels are necessary by this proof.

As the first stage of simulating *two stack machine* of s states, we constructed a dynamical system of $(s+2)$ coordinates. The first s represented the state of the machine in unary representation via binary values only. The last two coordinates represented the stacks. Using the notation $x_0 := 1 - \sum_{j=1}^{s} x_j$, the update equations were:

$$x_i^+ := \sigma\left[\sum_{j=0}^{s} \beta_{ij}(\zeta[q_1], \zeta[q_2], \tau[q_1], \tau[q_2])\, x_j\right] \tag{3}$$

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

$$\begin{aligned} q_i^+ := \quad & \\ \sigma\Bigg[& \left(\sum_{j=0}^{s} \gamma_{ij}^1(\zeta[q_1], \zeta[q_2], \tau[q_1], \tau[q_2])\, x_j\right) q_i + \\ & \left(\sum_{j=0}^{s} \gamma_{ij}^2(\zeta[q_1], \zeta[q_2], \tau[q_1], \tau[q_2])\, x_j\right)\left(\frac{1}{4}q_i + \frac{1}{4}\right) + \\ & \left(\sum_{j=0}^{s} \gamma_{ij}^3(\zeta[q_1], \zeta[q_2], \tau[q_1], \tau[q_2])\, x_j\right)\left(\frac{1}{4}q_i + \frac{3}{4}\right) + \\ & \left(\sum_{j=0}^{s} \gamma_{ij}^4(\zeta[q_1], \zeta[q_2], \tau[q_1], \tau[q_2])\, x_j\right)(4q_i - 2\zeta[q_i] - 1)\Bigg] \end{aligned} \tag{4}$$

for $i = 1, 2$. Note that the "σ" does not need to appear on the right hand side of both equations. It is used only for consistency in obtaining the desired result.

We proved the following lemma:

Lemma 6.1 For each function $\beta : \{0,1\}^4 \to \{0,1\}$ there exist vectors $v_1, v_2, \ldots, v_{16} \in \mathbb{Q}^6$ and scalars $c_1, c_2, \ldots, c_{16} \in \mathbb{Q}$ such that, for each $a, b, d, e, x \in \{0,1\}$ and each $q \in [0,1]$,

$$\beta(a,b,d,e)x = \sum_{i=1}^{16} c_i \sigma(v_i \cdot \mu)$$

and

$$\beta(a,b,d,e)xq = \sigma\left(q + \sum_{i=1}^{16} c_i\sigma(v_i \cdot \mu) - 1\right),$$

where we denote $\mu = (1, a, b, d, e, x)$ and "$\cdot$" = dot product in $\mathbb{Q}^6$.

In the case where the arguments are the top and empty functions of the stacks, the arguments are dependent and there is a need of only 9 terms in the summation, rather than 16. Indeed, if

$$(a,b,c,d) \equiv (\zeta[q_1], \zeta[q_2], \tau[q_1], \tau[q_2]) \ ,$$

then the information of the values

$$(1, a, b, c, d, ab, ad, bc, cd)$$

includes the information

$$(ac, bd, abc, abd, acd, bcd, abcd) \ .$$

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

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

where each q_{ij} is the jth term (row) in Equation (6.1). That is, q_{i1} may differ from 0 only if the last operation on stack i was "no-operation." Similarly, the components q_{i2}, q_{i3}, q_{i4} may differ from 0 only if the last operations of the ith stack were push0, push1, or pop , respectively.

Using Lemma 6.1. we can write

$$q_{ij} = \sigma\left(\text{next-q}_{ij} + \sum_{k=0}^{s} \gamma_{ik}^{j} x_k - 1\right) \ , \tag{5}$$

where

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

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

$$\begin{aligned} t_i &= \sum_{j=1}^{4} t_{ij} \ , \\ t_{ij} &= \sigma\left(4\left[\text{next-q}_{ij} + \sum_{k=0}^{s} \gamma_{ik}^{j} x_k - 1\right] - 2\right) \ , \end{aligned} \tag{6}$$

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

$$\begin{aligned} e_i &= \sum_{j=1}^{4} e_{ij} \ , \\ e_{ij} &= \sigma\left(4\left[\text{next-q}_{ij} + \sum_{k=0}^{s} \gamma_{ik}^{j} x_k - 1\right]\right) \ . \end{aligned} \tag{7}$$

Here, t_{ij} is the "top" of the element q_{ij}, and e_{ij} is the empty test of the same element. As three out of the four stack elements $\{q_{i1}, q_{i2}, q_{i3}, q_{i4}\}$ of each stack $i = 1, 2$ are 0, and the fourth has the value of the stack i, it makes sense to apply t and e to these four elements.

We construct a network in which the stacks and their readings are not kept explicitly in values q_i, t_i, e_i, but implicitly only via $q_{i1}, q_{i2}, q_{i3}, q_{i4}, t_{i1}, t_{i2}, t_{i3}, t_{i4}, e_{i1}, e_{i2}, e_{i3}, e_{i4}$, $i = 1, 2$. By Lemma 6.1, all update equations of

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

can be written as

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

that is, as a neural net with one hidden layer.

The main layer consists of the elements q_{ij}, t_{ij}, e_{ij}. In the hidden layer, we prepare all elements $\sigma(\cdots)$ required by the lemma to compute the functions β and γ. We showed that $9s + 2$ terms of this kind are required. We name these terms as "configuration detectors" as they provide the needed combination of states and stack readings. These terms are all that needed to compute x_k^+. We also keep in the hidden layer values of q_i, t_i to compute next-q_{ij}.

In summary:

- The main layer consists of:
  1. s neurons $x_k, k = 1, \ldots, s$ that represent the state of the system unarily.
  2. For each stack i, $i = 1, 2$ we have
     (a) four neurons $q_{ij}^1 \equiv q_{ij}, j = 1, 2, 3, 4$,
     (b) four neurons $t_{ij}^1 \equiv t_{ij}, j = 1, 2, 3, 4$,
     (c) four neurons $e_{ij}^1 \equiv e_{ij}, j = 1, 2, 3, 4$.
- The hidden layer consists of:
  1. $9s + 2$ neurons for configuration detecting. (The additional two are for the case of s_0.)
  2. For each stack i, $i = 1, 2$ we have
     (a) a neuron $q_i^2 \equiv q_i$,
     (b) a neuron $t_i^2 \equiv t_i$.

Remark 6.2 The number of neurons required to simulate a TM consisting of s states and p tapes:

$$\underbrace{s + 24}_{\text{main level}} + \underbrace{9^p s + 2 + 4p}_{\text{hidden level}}\ .$$

Hence, there is a simulation of a universal Turing Machine using 886 neurons.

6.2 Removing the Sigmoid From the Main Level

Here, we show how to construct an equivalent network to the above, in which the main level computes linear combinations only. This is the main part of the proof: The stack elements $\{q_{i1}, q_{i2}, q_{i3}, q_{i4}\}$ for each stack $i = 1, 2$, can receive not only values in $[0, 1]$ but also negative values. In case of negative values in the stack elements, we interpret them as 0. Top and isempty elements may also receive values out of the range $[0, 1]$.

To manage with only one level of sigma function, we represent the stack with large enough gaps between its different readings. We choose base 40, and represent the binary values of the stacks by the letter 31 to represent 0, and 39 to represent 1.

The readings of stack i are

$$\begin{aligned} \text{Top}\ (q_i) &:= 5(40q_i - 38)\ , \\ \text{Emp}\ (q_i) &:= 40q_i - 30\ . \end{aligned}$$

The values of these functions are

$$\text{Top}\ (q) \in \begin{cases} [5, 10] & \text{when the top is "1"} \\ [-35, -30] & \text{when the top is "0"} \\ [-\infty, -190] & \text{for an empty stack} \end{cases},$$

$$\text{Emp}\ (q) \in \begin{cases} [9, 10] & \text{when the top is "1"} \\ [1, 2] & \text{when the top is "0"} \\ [-\infty, -30] & \text{for an empty stack} \end{cases}.$$

Note that

Attribute 3.0 Any negative value of the functions Top and Emp has an absolute value of at least three times than any positive value of them.

The large negative numbers operate as inhibitors. We will see later how this attribute assists in constructing the network.

To verify the construction, we will generalize lemma 6.1 proved in [12]. There we used the variables a, b, c, d specified in the lemma as [top of stack1, top of stack2, isempty stack1, isempty stack2]. Here we are going to need more variables: $[a_1, \ldots, a_4]$ will be [top of element1 of stack1, ... ,top of element4 of stack1]. We also have $[b_1, \ldots, b_4]$, $[d_1, \ldots, d_4]$, $[e_1, \ldots, e_4]$, which will represent the different elements of top of stack2, isempty stack1 and isempty stack2, respectively.

Lemma 6.3 Let C be a range of values.

$$C = [1, 10] \cup [-\infty, -30]\ .$$

For each function $\beta : C^{16} \rightarrow \{0, 1\}$, there exist vectors $v_1, v_2, \ldots, v_{625} \in \mathbb{Q}^6$ and scalars $c_1, c_2, \ldots, c_{625} \in \mathbb{Q}$ such that, for each

$$a_j, b_k, d_m, e_p \in C \quad j, k, m, p = 1, 2, 3, 4\ ,$$

$x \in \{0, 1\}$, and $q \in [-\infty, 1]$, that satisfy the following condition:

Condition 6.4 For any two indices $j, k \in \{1, 2, 3, 4\}$:

$$\sigma(a_j)\sigma(a_k) = 0\ , \sigma(b_j)\sigma(b_k) = 0\ , \sigma(d_j)\sigma(d_k) = 0\ , \sigma(e_j)\sigma(e_k) = 0\ ,$$

(i.e. not more than one positive value to all as. The same holds for b, c, d.) we can write

$$\beta(a_1, a_2, \ldots, e_3, e_4)x = \sum_{i=1}^{625} c_i \sigma(v_i \cdot \mu_i) \ ,$$

and

$$q + \sum_{i=1}^{625} c_i \sigma(v_i \cdot \mu_i) - 1 = \begin{cases} q & \text{if } \beta(a_1, a_2, \ldots, e_3, e_4)x = 1 \\ q-1 & \text{if } \beta(a_1, a_2, \ldots, e_3, e_4)x = 0 \end{cases} ,$$

where we denote

$$\mu_i \in \{\mu_{jkmp} = (1, a_j, b_k, d_m, e_p, x) \ \ j, k, m, p = 1, 2, 3, 4\} \ ,$$

and "$\cdot$" = dot product in $\mathbb{Q}^6$. □

Proof. Write β as a polynomial

$$\begin{aligned} \beta(a_1, a_2, \ldots, e_3, e_4) \ = \ & c_1 + c_2\sigma(a_1) + c_3\sigma(a_2) + \cdots + c_{17}\sigma(e_4) + \\ & c_{18}\sigma(a_1)\sigma(b_1) + c_{19}\sigma(a_1)\sigma(b_2) + \cdots c_{113}\sigma(d_4)\sigma(e_4) \\ & c_{114}\sigma(a_1)\sigma(b_1)\sigma(d_1) + c_{115}\sigma(a_1)\sigma(b_1)\sigma(d_2) + \cdots + c_{369}\sigma(b_4)\sigma(d_4)\sigma(e_4) \\ & c_{370}\sigma(a_1)\sigma(b_1)\sigma(d_1)\sigma(e_1) + \cdots + c_{625}\sigma(a_4)\sigma(b_4)\sigma(d_4)\sigma(e_4) \ . \end{aligned}$$

Observe that for any sequence $l_1, \ldots, l_k$ of ($k \leq 4$) elements in C and $x \in \{0, 1\}$, one has $\sigma(l_1) \cdots \sigma(l_k)x = \sigma(l_1 + \cdots + l_k + 10kx - 10k)$. This is due to the fact that the sum of k, $k \leq 4$ elements of C is negative when at least one of the elements is negative. This stems fron attribute (3.0).

Expand the product $\beta(a_1, a_2, \ldots, e_3, e_4)x$, using the above observation and the fact $x = \sigma(x)$. This gives that

$$\begin{aligned} \beta(a_1, a_2, \ldots, e_3, e_4)x \ &= \ c_1\sigma(x) + c_2\sigma(a_1 + 10x - 10) + \cdots + c_{625}\sigma(a_4 + b_4 + d_4 + e_4 + 40x - 40) \\ &= \ \sum_{i=1}^{625} c_i \sigma(v_i \cdot \mu_i) \ . \end{aligned}$$

for suitable c_i's and v_i's, where $\mu_i \in \{\mu_{jkmp} \ \ j, k, m, p = 1, 2, 3, 4\}$. On the other hand, for each $\tau \in \{0, 1\}$ and each $q \in [-\infty, 1]$ it holds that

$$(q + \tau - 1) = \begin{cases} q & \text{if } \tau = 1 \\ q-1 & \text{if } \tau = 0 \ . \end{cases}$$

So, substituting the above formula with $\tau = \beta(a_1, a_2, \ldots, e_3, e_4)x$ gives the desired result. ∎

In the case where the arguments are the top and empty functions of the stacks, the arguments are dependent and there is a need of only 81 terms in the summation rather than 625.

Network Description

The network consists of two levels. The main level consists of s state neurons that are updated as

beforehand, the stack neurons and readings $q^1_{ij}, t^1_{ij}, e^1_{ij}$, $i = 1,2$ $j = 1,\ldots,4$, for stack elements, top elements, and isempty elements. The update equations are

$$\begin{aligned} q^1_{ij} &= \text{next-q}_{ij} + \sum_{k=0}^{s} \gamma^j_{ik} x_k - 1 \ , \\ t^1_{ij} &= 5\left[40(\text{next-q}_{ij} + \sum_{k=0}^{s} \gamma^j_{ik} x_k - 1) - 38\right] \ , \\ e^1_{ij} &= 40(\text{next-q}_{ij} + \sum_{k=0}^{s} \gamma^j_{ik} x_k - 1) - 30 \ , \end{aligned}$$

where

$$\text{next-q}_{ij} = \begin{cases} q_i & \text{if } j = 1 \\ \frac{1}{40} q_i + \frac{31}{40} & \text{if } j = 2 \\ \frac{1}{40} q_i + \frac{39}{40} & \text{if } j = 3 \\ 40 q_i - 8\zeta[q_i] - 31 & \text{if } j = 4 \end{cases} .$$

Using Lemma 6.3, the expressions γx can be written as linear combinations of terms like $\sigma(\cdots)$, which are the configuration detectors.

Second level consists of both $(81s + 2)$ configuration detectors neurons —as proved in Lemma 6.3— and the stack and top neurons:

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

which are updated by the equations

$$\begin{array}{lll} q^2_{ij} := \sigma(q^1_{ij}) & & [q_i = \sum_{j=1}^{4} q^2_{ij}] \ , \\ t^2_{ij} := \sigma(t^1_{ij}) & i = 1.2 \quad j = 1,\ldots,4 & [t_i = \sum_{j=1}^{4} t^2_{ij}] \ . \end{array}$$

6.3 One Level RNN Simulates TM

Consider the above network. Remove the main level and leave the hidden level only, while letting each neuron there compute the information that received beforehand from a neuron of the main level. (Except for the I/O neurons, a TM of s states and t binary tapes is simulated by $[81^t s + 16t + 2]$ neurons.)

7 Appendix: Simulation of Vector Machines

This section recalls the definition of vector machines and sketches a proof of theorem 4.

A *vector machine* is a processor together with vector registers V_1, V_2, ..., V_k that contain 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. Input is given to the machine in register V_1, and 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.
- $V_i := V_i \wedge V_j$: Bitwise AND V_i and V_j.
- $V_i := V_i \uparrow V_j$: 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 $V_i = 0$ go to *label*.

A vector machine is *restricted* if arguments V_j in shift instructions always have values $O(t(n))$, where $t(n)$ is the running time of the machine.

The neurons in the nets we consider in this section have update equations of the form

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

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 non-periodic rationals always ends in an infinite sequence of zeros. That is, 1/2 is represented as $0.10000\ldots$, not as $0.01111\ldots$

One such 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 constantly zero after the first $p(n)$ digits.

Proof of theorem 4. Let M be a restricted vector machine running in time $t(n)$. Let s denote the maximum length that a register of M can reach on an input of length n. It is easy to prove by induction that $s = 2^{O(t(n))}$ (the restriction on the shifts is necessary here). We assume for simplicity that s is a power of two.

To simulate M by means of a neural net, we encode the whole memory of M at every moment in a single real number r, $0 \leq r < 1$. That is, if the registers contain c_1, c_2, ... c_k, (each c_i a bit vector of length exactly s), r will be $0.c_1c_2c_3\ldots c_k0000\ldots$ Thus, the decimal expansion of r will be divided in a constant number of bit fields, each of length s.

Initially, the net reads the input and stores it as the state of one of the neurons, as described in [12]. Then it uses this state to build c_1 and lets all other c_i's in r be 0.

For the actual computation, we divide the proof in two parts: First, we show that the effect of each vector instruction on the real r can be simulated by rational functions and bitwise-AND's 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.

So, given the real r, simulate each vector instruction as follows:

- $V_i := x$. (In this abstract, we give a program for this instruction only, and describe all others informally).

```
/* constants during this program:
       i, x - from the instruction
       k - number of vector registers used in the program
       s - number of bits used per vector
*/
p := 1/2;
for j := 1 to log s do
       p := p^2
end;                          /* p = 2^{-s} = 0.000...000 1000... */
                                          (s-1)
m := 1 - p^{i-1} + p^i - p^k; /* m = 0.111...111 000...000 111...111 000... */
                                      (i-1)·s     s     (k-i)·s
q := r ∧ m;                   /* clear ith field with mask m */
r := q + x * p^{i+1};         /* insert x in ith field */
```

Thus, updating r this way requires $O(\log s)$ operations.

- $V_i :=$ not V_i: create a mask m that has 1 in all the bits of the ith field of r, and 0 in all others; AND it with r, thus obtaining a real

$$r_i = 0.\underbrace{000\ldots000}_{(i-1)\cdot s} c_i 000\ldots$$

where c_i has s bits. Subtract r_i from m, obtaining the negation of the register contents. Insert the result back into r, as described above.

- $V_i := V_i \wedge V_j$: Obtain two reals r_i and r_j encoding fields i and j in r, as in the case of negation. Multiply or divide r_j by a suitable power of two to align the important bits of r_i and r_j (i.e., the 2^s bits encoding the registers). AND them and insert the result back into r.

- $V_i := V_i \uparrow V_j$. Obtain r_j encoding the contents x of register j. If x is negative (i.e., its first bit is 1) do nothing. Otherwise, build the real 2^{-x}; since M is restricted, we know that x is $O(t(n))$; this allows one to compute 2^{-x} in time $O(t(n))$ by letting a neuron count up to x while another neuron divides by 2 at each step. Obtain r_i, multiply it by 2^{-x}, discard the bits that go beyond the limit of the ith field (by ANDing with a suitable mask), and insert the result back into r.

- $V_i := V_i \downarrow V_j$. Similar to left shift using division by 2^{-x}.

- **if $V_i = 0$ go to *label:* We implement the test "if $V_i = \ldots 1111$ go to *label*" instead (which is equivalent because we can first negate V_i). First, extract field i of r and move it next to the decimal point, obtaining $0.c_i000\ldots$ Then decide whether c_i is all ones or not, fast ANDing all its bits by recursive folding: Suppose that $c_i = b_1 b_2 \ldots b_{2k}$. Build $0.b_1 \ldots b_k 000\ldots$ and $0.b_{k+1} \ldots b_{2k} 000\ldots$ AND them. Iterate, each time halving the number of relevant bits. After $\log s$ iterations, we have obtained either $0.1000\ldots$ or $0.000\ldots$, depending on whether c_i was $111\ldots111$ or not.**

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 algorithm described above for the instruction $V_i := x$.

This net reads the states of three neurons r, ℓ, and a, and gives to the rest of the simulating net the states of two neurons u and v, with the following meanings:

- r contains the current value of the real r encoding the vector machine's memory.
- ℓ contains 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).
- a produces a 1 during a step to "activate" the subnet we are designing. Is 0 at all other times.
- v produces a 1 pulse $(\log s) + 3$ steps after a, and is 0 at all other times.
- u contains the updated value for r at the moment where v produces the pulse.

We also use neurons called p, m, q, p_1, p_2, and c. The functions computed by each neuron are:

$$
\begin{array}{ll}
p^+ := \sigma(\ a/2 + (1-a)p^2\ ) & /^*\ a = 1 \text{ resets } p \text{ to } 1/2,\ a = 0 \text{ squares it } {}^*/ \\
m^+ := \sigma(\ 1 - p^{i-1} + p^i - p^k\ ) & \\
q^+ := \sigma(\ r \wedge \dot{m}\ ) & \\
p_1^+ := \sigma(\ p\ ) & /^*\ p_1,\ p_2 \text{ delay } p \text{ two steps } {}^*/ \\
p_2^+ := \sigma(\ p_1\ ) & \\
u^+ := \sigma(\ q + x * p_2^{i+1}\ ) & \\
c^+ := \sigma(\ a \cdot \ell/8 + (1-a) \cdot 2c\ ) & /^*\ a = 1 \text{ resets counter to } 1/(8s)\ {}^*/ \\
v^+ := \sigma(\ 2c - 1\ ) & /^*\ v^+ = 0 \text{ for } u \leq 1/2,\ v^+ = 1 \text{ for } u \geq 1\ {}^*/
\end{array}
$$

The output of u should be used to update neuron r at the precise step when $v = 1$. ∎

8 Appendix: Kolmogorov Weights

In this appendix we prove theorems 7 and 8. The proofs consist of several steps. First, we observe the equivalence between the class of neural networks and the class of oracle Turing machines that consult tally sets. Then, we prove the hierarchy in the latter class. In both steps, we heavily rely upon the use of different types of computable numbers. The sketch of the proof is given in the next two subsections.

8.1 Equivalence of TMs with Tally Oracles and NNs

Definition 8.1 Let S be a set of infinite binary strings. S is *closed under mixing* if for any finite number $k \in \mathbb{N}$ and for any k strings from S,

$$\alpha^1 = \alpha_1^1\alpha_2^1\alpha_3^1\cdots \ , \ \alpha^2 = \alpha_1^2\alpha_2^2\alpha_3^2\cdots \ , \ \ldots, \ \alpha^k = \alpha_1^k\alpha_2^k\alpha_3^k\cdots \ ,$$

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 8.2 Let S be a set of infinite binary strings. We define the *fraction set* for S to be

$$\tilde{S}_3 = \{\omega \in [0,1] \mid \omega = \sum_{i=1}^{\infty} \frac{2\omega_i}{3^i} \text{ and } \omega_1\omega_2\cdots \in S\}$$

We can give now the precise statement of theorem 8. Let $\mathcal{B}$ be a class of time constructible functions, e.g., polynomials. Denote by $\bigcup_T \mathcal{B}(T)$ the class of Turing machines that compute in time $B \in \mathcal{B}$, where each machine consults an oracle which is a tally set.

Theorem 9 *et $S \subseteq \{0,1\}^\infty$ closed under mixing. The following classes are polynomially time related:*

1. *The subset of $\bigcup_T B(T)$ consisting of those machines that consult tally sets T and having the property that, for each $n \in \mathbb{N}$, the characteristic string χ_T of the tally set satisfies that $\chi_{T\leq n} \in S$.*
2. *The class of neural networks that have all weights in the set $\tilde{S}_3 \bigcup \mathbb{Q}$ and that compute in time $B \in \mathcal{B}$.*

The next two subsections prove this theorem.

8.1.1 Proof: $1 \subseteq 2$

Definition 8.3 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 set* Y so that

- 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 are being changed:**

$$A \leftarrow \begin{cases} 1 & Q \in Y \\ 0 & Q \notin Y \end{cases} , \qquad Q \leftarrow 0 , \qquad W \leftarrow 0 ;$$

the neurons of $\mathcal{N}$ do not change.

Setting $W = 1$ is like invoking a subroutine for solving a membership query in Y, except that we assume that this oracle subroutine operates in a unit time.

Define a function

$$\delta_4 : \{0,1\}^\infty \rightarrow [0,1]$$

by the formula

$$\delta_4(\alpha) = \sum_{i=1}^{\infty} \frac{2\alpha_i + 1}{4^i} .$$

Let Y be a set of strings, then the "fraction base-4" set of Y is defined as

$$\tilde{Y}_4 \equiv \{\delta_4(\alpha) \mid \alpha \in Y\} .$$

Note that if Y is a tally set, i.e., a subset of $\{1\}^*$, then $\tilde{Y}_4 \subseteq \{\sum_{i=1}^{n}(\frac{3}{4})^i\}_{n=1}^{\infty}$.

Lemma 8.4 Let T be a tally set. Then the following models are polynomially equivalent:

- Oracle TM that consults the tally set T and computes in time $B \in \mathcal{B}$.
- Oracle NN with the oracle $\tilde{T}_4$ that computes in time $B \in \mathcal{B}$.

The proof of this lemma is very similar to the proof of the main result in [12], and is not included here.

Lemma 8.5 For each set $\tilde{Y}_4 \subseteq \{\sum_{i=1}^{n}(\frac{3}{4})^i\}_{n=1}^{\infty}$, there exists a network of 5 neurons and two inputs, u_1, u_2, that starts from the zero initial value, and given the input signals

$$\begin{aligned} u_1 &= [\sum_{i=1}^{n}(\frac{3}{4})^i]\ 0\ 0\ 0\ \cdots \\ u_2 &= 1\ 0\ 0\ 0\ \cdots \end{aligned}$$

the network outputs

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

where b is the truth value of $u_1(1) \in \tilde{Y}_4$.

Proof. Encode $\tilde{Y}_4$ as a fraction

$$\mu = \sum_{n=1}^{\infty} \frac{2b_n}{3^n} ,$$

where

$$b_n = \begin{cases} 1 & (\sum_{i=1}^{n}(\frac{3}{4})^i) \in \tilde{Y}_4 \\ 0 & \text{otherwise.} \end{cases}$$

We use μ as one of the weights of the network.

Notice that

$$\begin{aligned} u_1(1) &= .\underbrace{33\cdots3}_{n} \quad \text{in base 4} \\ \mu &= .2022002\cdots \quad \text{in base 3}, \end{aligned}$$

and the nth digit of μ in the base 3 expansion is the decision of whether $(\sum_{i=1}^{n}(\frac{3}{4})^i) \in \tilde{Y}_4$.

The network simultaneously scans the value given in μ [in x_1 and x_2] and the value of $u_1(1)$ [in x_3 and x_4]. When reaches the last digit '3' of $u_1(1)$, the network returns the currently scanned b_i.

$$\begin{aligned} x_1^+ &= \sigma(u_2\mu + 3x_1 - 2x_2) \\ x_2^+ &= \sigma(3u_2\mu + 9x_1 - 6x_2 - 1) \\ x_3^+ &= \sigma(u_1 + 4x_3 - 3 + 3u_2) \\ x_4^+ &= \sigma(4u_1 + 4x_4 - 3)) \\ x_5^+ &= \sigma(4x_3 - 3 + x_2 - 16x_4)\,. \end{aligned}$$

■

Using the above two lemmas, we can prove the inclusion $1 \subseteq 2$ as follows: let M be an OTM that computes in time $B \in \mathcal{B}$ using a tally set T as an oracle. It is given that the characteristic string of the set T, $\chi_T \in S$. We consider an ONN with the oracle set in $\tilde{S}_3$. As T is a tally set, $\tilde{T}_4 \subseteq \{\sum_{i=1}^{n}(\frac{3}{4})^i\}_{n=1}^{\infty}$.

We couple the ONN $(\mathcal{N}, Q, A, W)$ with a retrieval network as described in lemma (8.5), and obtain in this manner a network that operated in time polynomial in B. In [12], we showed that for any TM, $\exists$ a NN of Rational weights that simulates it in linear time. Hence, and from the above description, it follows that given a TM with an oracle in S, there is a corresponding neural network whose weights are either Rationals or in the set $\tilde{S}_3$.

(To couple the OTM network with the retrieval network:

We add the neurons

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

where t_1, t_2 will be used as the input u_1 and u_2 of the retrieval network. The neuron t_3 is used the 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.)

8.1.2 Proof: $2 \subseteq 1$

Given a network $\mathcal{N}$ with weights in $\tilde{S}_3 \bigcup \mathbb{Q}$. The network has a fixed number $k' \in \mathbb{N}$ of weights:

$$\omega^1 = \omega_1^1\omega_2^1\omega_3^1 \cdots \ , \ \omega^2 = \omega_1^2\omega_2^2\omega_3^2 \cdots \ , \ \ldots \ \omega^{k'} = \omega_1^{k'}\omega_2^{k'}\omega_3^{k'} \cdots$$

Assume k of them are in S and the rest are Rationals. As S is closed under mixing, the string

$$\alpha = \frac{\omega_1^1}{2}\frac{\omega_1^2}{2}\frac{\omega_1^3}{2} \cdots \frac{\omega_1^k}{2}\frac{\omega_2^1}{2}\frac{\omega_2^2}{2}\frac{\omega_2^3}{2} \cdots \frac{\omega_2^k}{2}\frac{\omega_3^1}{2}\frac{\omega_3^2}{2}\frac{\omega_3^3}{2} \cdots$$

is again an element of S.

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

1. M receives the input string α.
2. M computes the computational time $B(|\alpha|)$ of $\mathcal{N}$.
3. for a given constant C, M executes:

   For $i = 1$ to $kCB(|\alpha|)$
   query the ith word.

   Now, M has the weights of $\mathcal{N}$ up to a precision $CB(|\alpha|)$. C is a constant such that this precision suffices. The existence of such a C was proved in [13]. (The $(k' - k)$ Rational weights are encoded in the machine M.)
4. M simulates $\mathcal{N}$ step by step in polynomial time.

8.2 Hierarchy of TMs That Consult Tally Oracles

Theorem 10 *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 so that the characteristic string of the set T is $\chi_{T^{\leq n}} \in K[F, poly]$, $F \in \mathcal{F}$. We 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 $s^{-1}(m)$ denote $\max\{k \mid s(k) \leq m\}$. Choose an infinite sequence γ such that for every polynomial p and almost every m, $\gamma_{1:m} \notin K[m/2, p(s^{-1}(2m))]$. 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 $\chi_{\mathcal{A}^{\leq n}} \in K[s(n)/2 + c, q(n)] \subseteq K[s(n), q(n)]$, 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 $\chi_{T^{\leq n}} \in K[r(n), p_2(n)]$, with p_1, p_2 polynomials. Then $\gamma_{1:s(n)/2}$ is obtained from some string of length

$r(p_1(n)) + O(1) < s(n)/4$, in time $O(p_2(p_1(n)))$. Observing that $n \leq s^{-1}(s(n)) = s^{-1}(2|\gamma_{1:s(n)/2}|)$, this contradicts the choice of γ. ∎

On a more abstract level, we could generalize the theorem as follows:

Lemma 8.6 Let $\mathcal{F}, \mathcal{G}$ be function classes as above. $\mathcal{B}, \mathcal{D}$ are time constructible classes *closed under* $\mathcal{B}$. That is, $\forall B \in \mathcal{B}, D \in \mathcal{D} : \; mB(m)D(m) \in \mathcal{B}$.

We define $\mathcal{L}(\bigcup_T D(T, \mathcal{F}, \mathcal{B}))$ as the class of languages recognized by TMs, where each TM operates in time $D \in \mathcal{D}$, and consult a tally oracle whose characteristic string satisfies $\chi_{T \leq n} \in K[F, B]$, $F \in \mathcal{F}$, $B \in \mathcal{B}$.

Then: $\mathcal{L}(\bigcup_T D(T, \mathcal{F}, \mathcal{B}))$ and $\mathcal{L}(\bigcup_T D(T, \mathcal{G}, \mathcal{B}))$ are different.

The combination of theorem 8 and theorem 10 proves theorem 7.