The Computational Limits to the Cognitive Power of the Neuroidal Tabula Rasa

Wiedermann, Jiří
1999

INSTITUTE OF COMPUTER SCIENCE

ACADEMY OF SCIENCES OF THE CZECH REPUBLIC

The Computational Limits to the Cognitive Power of the Neuroidal Tabula Rasa

Jiří Wiedermann

Technical report No. 786

July, 1998

Institute of Computer Science, Academy of Sciences of the Czech Republic
Pod vodárenskou věží 2, 182 07 Prague 8, Czech Republic
phone: (+4202) 66 05 35 20 fax: (+4202) 85 85 789
e-mail: wieder@uivt.cas.cz

INSTITUTE OF COMPUTER SCIENCE

ACADEMY OF SCIENCES OF THE CZECH REPUBLIC

The Computational Limits to the Cognitive Power of the Neuroidal Tabula Rasa

Jiří Wiedermann [1]

Technical report No. 786
July, 1998

Abstract

The neuroidal tabula rasa (NTR) as a hypothetical device which is capable of performing tasks related to cognitive processes in the brain was introduced by L.G. Valiant in 1994. Neuroidal nets represent a computational model of the NTR. Their basic computational element is a kind of a programmable neuron called neuroid. Essentially it is a combination of a standard threshold element with a mechanism that allows modification of the neuroid's computational behaviour. This is done by changing its state and the settings of its weights and of threshold in the course of computation. The computational power of an NTR crucially depends both on the functional properties of the underlying update mechanism that allows changing of neuroidal parameters and on the universe of allowable weights. We will define instances of neuroids for which the computational power of the respective finite-size NTR ranges from that of finite automata, through Turing machines, upto that of a certain restricted type of BSS machines that possess super–Turing computational power. The latter two results are surprising since similar results were known to hold only for certain kinds of analog neural networks.

Keywords

[1] This research was supported by GA ČR Grant No. 201/98/0717

1 Introduction

Nowadays, we are witnessing a steadily increasing interest towards understanding the algorithmic principles of cognition. The respective branch of computer science has been recently appropriately named as cognitive computing. This notion, coined by L.G. Valiant [8], denotes any computation whose computational mechanism is based on our ideas about brain computational mechanisms and whose goal is to model cognitive abilities of living organisms. There is no surprise that most of the corresponding computational models are based on formal models of neural nets.

Numerous variants of neural nets have been proposed and studied. They differ in the computational properties of their basic building elements, viz. neurons. Usually, two basic kinds of neurons are distinguished: discrete ones that compute with Boolean values, and analog (or continuous) ones that compute with any real or rational number between 0 and 1.

As far as the computational power of the respective neural nets is concerned, it is known that the finite nets consisting of discrete neurons are computationally equivalent to finite automata (cf. [5]). On the other hand, finite nets of analog neurons with rational weights, computing in discrete steps with rational values, are computationally equivalent to Turing machines (cf. [3]). If weights and computations with real values are allowed then the respective analog nets possess even super–Turing computational abilities [4]. No types of finite discrete neural nets are known that would be more powerful than the finite automata.

An important aspect of all interesting cognitive computations is learning. Neural nets learn by adjusting the weights on neural interconnections according to a certain learning algorithm. This algorithm and the corresponding mechanism of weight adjustment are not considered as part of the network.

Inspired by real biological neurons, Valiant suggested in 1988 [6] a special kind of programmable discrete neurons, called neuroids, in order to make the learning mechanism a part of neural nets. Based on its current state and current excitation from firings of the neighboring neuroids, a neuroid can change in the next step all its computational parameters (i.e., can change its state, threshold, and weights). In his monograph [7] Valiant introduced the notion of a neuroidal tabula rasa (NTR). It is a hypothetical device which is capable of performing tasks related to cognitive processes. Neuroidal nets serve as a computational model of the NTR. Valiant described a number of neuroidal learning algorithms demonstrating a viability of neuroidal nets to model the NTR. Nevertheless, insufficient attention has been paid to the computational power of the respective nets. Without pursuing this idea any further Valiant merely mentioned that the computational power of neuroids depends on the restriction put upon their possibilities to self–modify their computational parameters.

It is clear that by identifying a computational power of any learning device we get an upper qualitative limit on its learning or cognitive abilities. Depending on this limit, we can make conclusions concerning the efficiency of the device at hand and those related to its appropriateness to serve as a realistic model of its real, biological counterpart.

In this paper we will study the computational power of the neuroidal tabula rasa

which is represented by neuroidal nets. The computational limits will be studied w.r.t the various restrictions on the update abilities of neuroidal computational parameters.

In Section 2 we will describe a broad class of neuroidal networks as introduced by Valiant in [7].

Next, in Section 3, three restricted classes of neuroidal nets will be introduced. They will include nets with a finite, infinite countable (i.e, integer), and uncountable (i.e., real) universe of weights, respectively.

Section 4 will briefly sketch the equivalence of the most restricted version of finite neuroidal nets — namely those with a finite set of parameters, with the finite automata.

In Section 5 we further show the computational equivalence of the latter neuroidal nets with the standard neural nets.

The next variant of neuroidal nets, viz. those with integer weights, will be considered in Section 6. We will prove that finite neuroidal nets with weights of size $S(n)$, which allow a simple arithmetic over their weights (i.e., adding or subtracting of the weights), are computationally equivalent to computations of any $S(n)$–space bounded Turing machine.

In Section 7 we will increase the computational power of the previously considered model of neuroidal nets by allowing their weights to be real numbers. The resulting model will turn to be computationally equivalent to the so–called additive BSS machine ([1]). This machine model is known for its ability to solve some undecidable problems.

Finally, in the conclusions we will discuss the merits of the results presented.

2 Neuroidal Nets

In what follows we will define neuroidal nets making use of the original Valiant's proposal [7], essentially including his notation.

Definition 2.1 A neuroidal net $\mathcal{N}$ *is a quintuple* $\mathcal{N} = (G, W, X, \delta, \lambda)$, *where*

• $G = (V, E)$ *is the directed graph describing the topology of the network;* V *is a finite set of* N *nodes called* neuroids *labeled by distinct integers* $1, 2, \ldots, N$, *and* E *is a set of* M *directed edges between the nodes. The edge* (i, j) *for* $i, j \in \{1, \ldots, N\}$ *is an edge directed from node* i *to node* j.

• W *is the set of numbers called* weights. *To each edge* $(i, j) \in E$ *there is a value* $w_{i,j} \in W$ *assigned at each instant of time.*

• X *is the finite set of the* modes *of neuroids which a neuroid can be in each instant. Each mode is specified as a pair* (q, p) *of values where* q *is the member of a finite set* Q *of states, and* p *is an integer from a finite set* T *called the set of* thresholds *of the neuroid.*

Q consists of two kinds of states called firing *and* quiescent *states.*

To each node i there is also a Boolean variable f_i having value one or zero depending on whether the node i is in a firing state or not.

• δ *is the recursive* mode update function *of form* $\delta : X \times W \to X$.

Let $w_i \in W$ *be the sum of those weights* w_{ki} *of neuroid* i *that are on edges* (k, i) *coming from neuroids which are currently firing, i.e., formally* $w_i = \sum_{\substack{k \text{ firing} \\ (k,i) \in E}} w_{ki} =$

$\sum_{\substack{j \\ (j,i)\in E}} f_j w_{ji}$. *The value of* w_i *is called the* excitation *of* i *at that time.*

The mode update function δ *defines for each combination* (s_i, w_i) *holding at time* t *the mode* $s' \in X$ *that neuroid* i *will transit to at time* $t+1$*:* $\delta(s_i, w_i) = s'$.

• λ *is the recursive* weight update function *of form* $\lambda : X \times W \times W \times \{0,1\} \to W$. *It defines for each weight* w_{ji} *at time* t *the weight* w'_{ji} *to which it will transit at time* $t+1$*, where the new weight can depend on the values of each* s_i*,* w_i*,* w_{ji}*, and* f_j *at time* t*:* $\lambda(s_i, w_i, w_{ji}, f_j) = w'_{ji}$

The elements of sets Q, T, W, and f_i's are called *parameters* of net $\mathcal{N}$.

A *configuration* of $\mathcal{N}$ at time t is a list of modes of all neurons followed by a list of weights of all edges in $\mathcal{N}$ at that time. The respective lists of parameters are pertinent to neuroids ordered by their labels and to edges ordered lexicographically w.r.t. the pair of labels (of neuroids) that identify the edge at hand. Thus at any time a configuration is an element from $X^N \times W^M$.

The *computation of a neuroidal network* is determined by the *initial conditions* and the *input sequence*. The initial conditions specify the initial values of weights and modes of the neuroids. These are represented by the initial configuration. The input sequence is an infinite sequence of inputs or *stimuli* which specifies for each $t = 0, 1, 2, \ldots$ a set of neuroids along with the states into which these neuroids are forced to enter (and hence forced to fire or prevented from firing) at that time by mechanisms outside the net (by peripherals).

Formally, each stimulus is an N–tuple from the set $\{Q \cup *\}^N$. If there is a symbol q at i–th position in the t–th N–tuple s_t, then this denotes the fact that the neuroid i is forced to enter state q at time t. The special symbol $*$ is used as don't–care symbol at positions which are not influenced by peripherals at that time.

A *computational step* of neuroidal net $\mathcal{N}$, which finds itself in a configuration c_t and receives its input s_t at time t, is performed as follows. First, neuroids are forced to enter into states as dictated by the current stimuli. Neurons not influenced by peripherals at that time retain their original state as in configuration c_t. In this way a new configuration c'_t is obtained. Excitation w_i is computed for this configuration now and the mode and weight updates are realized for each neuroid i in parallel, in accordance with the respective function δ and λ. In this way a new configuration c_{t+1} is entered.

The result of the computation after the t–th step is the N–tuple of states of all neuroids in c_{t+1}. This N–tuple is called the *action* at time t. Obviously, any action is an element in Q^N. Then the next computational step can begin.

The output of the whole computation can be seen as an infinite sequence of actions.

From the computational point of view any neuroidal net can be seen as a transducer which reads an infinite sequence of inputs (stimuli) and produces an infinite sequence of outputs (actions).

For more details about the model see [7].

3 Variants of neuroidal nets

In the previous definition of neuroidal nets we allowed set W to be any set of numbers and the weight and mode update functions to be arbitrary recursive functions. Intuitively it is clear that by restricting these conditions we will get variants of neural nets differing in their expressiveness as well as in their computing power. In his monograph Valiant [7] discusses this problem and suggests two extreme possibilities.

The first one considers such neuroidal nets where the set of weights of individual neuroids is finite. This is called a "simple complexity-theoretic model" in Valiant's terminology. We will also call the respective model of a neuroid as a "finite weight" neuroid. Note that in this case functions δ and λ can both be described by finite tables.

The next possibility we will study are neuroidal nets where the universe of allowable weights and thresholds is represented by the infinite set of all integers. In this case it is no longer possible to describe the weight update function by a finite table. What we rather need is a simple recursive function that will allow efficient weight modifications. Therefore we will consider a weight update function which allows setting a weight to some constant value, adding or subtracting the weights, and assigning existing weights to other inputs edges. Such a weight update function will be called a *simple–arithmetic update function.* The respective neuroid will be called an "integer weight" neuroid. The *size* of each weight will be given by the number of bits needed to specify the respective weight value. This is essentially a model that is considered in [7] as the counterpart of the previous model.

The final variant of neuroidal nets which we will investigate is the variant of the previously mentioned model with real weights. The resulting model will be called *an additive real neuroidal net.*

4 Finite weight neuroidal nets and finite automata

It is obvious that in the case of neuroidal nets with finite weights there is but a final number of different configurations a single neuroid can enter. Hence its computational activities like those of any finite neuroidal net, can be described by a single finite automaton (or more precisely: by a finite transducer). In order to get some insight into the relation between the sizes of the respective devices we will describe the construction of the respective transducer in more detail in the next theorem. In fact this transducer will be a Moore machine (i.e., the type of a finite automaton producing an output after each transition) since there is an output (action) produced by $\mathcal{N}$ after each computational move.

Theorem 4.1 *Let $\mathcal{N}$ be a finite neuroidal net consisting of N neuroids with a finite set of weights. Then there is a constant $c > 0$ and a finite Moore automaton $\mathcal{A}$ of size $\Theta(c^N)$ that simulates $\mathcal{N}$.*

Sketch of the proof: We will describe the construction of the Moore automaton $\mathcal{A} = (I, S, q_0, O, \Delta)$. Here I denotes the input alphabet whose elements are N-tuples of the stimuli: $I = \{Q \cup *\}^N$. Set S is a set of states consisting of all configurations

of $\mathcal{N}$, i.e., $S = X^N \times W^M$. State q_0 is the initial state and it is equal to the initial configuration of $\mathcal{N}$. Set O denotes a set of outputs of $\mathcal{A}$. It will consist of all possible actions of $\mathcal{N}$: $O = Q^N$.

The transition function $\Delta : I \times S \to S \times O$ is defined as follows: $\Delta(i, s_1) = (s_2, o)$ if and only if the neuroid $\mathcal{N}$ in configuration s_1 and with input i will enter configuration s_2 and produce output o in one computational move. It is clear that the input–output behaviour of both $\mathcal{N}$ and $\mathcal{A}$ is equivalent. □

Note that the size of the automaton is exponential w.r.t the size of the neuroidal net. In some cases such a size explosion seems to be unavoidable. For instance, a neuroidal net consisting of N neuroids can implement a binary counter that can count up to c^N, where $c \geq 2$ is a constant which depends on the number of states of the respective neuroids. The equivalent finite automaton would then require at least $\Omega(c^N)$ states. Thus the advantage of using neuroidal nets instead of finite automata seems to lie in the description economy of the former devices.

The reverse simulation of a finite automaton by a finite neuroidal net is trivial. In fact, a single neuroid, with a single input, is enough. During the simulation, this neuroid transits to the same states as the simulated automaton would. There is no need for a neuroid to make use of its threshold mechanism.

5 Simulating Neuroidal Nets by Neural Nets

Neural nets are a restricted kind of neuroidal networks in which the neuroids can modify neither their weights nor their thresholds. The respective set of neuroidal states consists of only two states — of a firing and quiescent state. Moreover, the neurons are forced to fire if and only if the excitation reaches the threshold value. The computational behaviour of neural networks is defined similarly as that of the neuroidal ones.

It has been observed by several authors that neural nets are also computationally equivalent to the finite automata (cf. [5]). Thus, we get the following consequence of the previous theorem:

Corollary 5.1 *The computational power of neuroidal nets with a finite set of weights is equivalent to that of standard non-programmable neural nets.*

In order to better appreciate the relationship between the sizes of the respective neuroidal and neural nets, we will investigate the direct simulation of finite neuroidal nets with finite weights by finite neural nets.

Theorem 5.1 *Let $\mathcal{N} = (G, W, X, \delta, \lambda)$ be a finite neuroidal net consisting of N neuroids and M edges. Let the set of weights of $\mathcal{N}$ be finite. Let $|L|$, and $|D|$, respectively, be the number of all different sets of arguments of the corresponding weight and mode update function. Let $|S|$ be the set of all possible excitation values, $|S| \leq 2^{|W|}$.*

Then $\mathcal{N}$ can be simulated by a neural network $\mathcal{N}'$ consisting of $O((|X| + |S| + |L| + |D|)N + |W|M)$ neurons.

Proof: It is enough to show that to any neuroid i of $\mathcal{N}$ an equivalent neural network C_i can be constructed. At any time the neuroid i is described by its "instantaneous description", viz. its mode and the corresponding set of weights. The idea of simulation is to construct a neural net for all combinations of parameters that represent a possible instantaneous description of i. The instantaneous value of each parameter will be represented by a special module. There will also be two extra modules to realize the mode and weight update functions. Instead of changing the parameters the simulating neural net will merely "switch" among the appropriate values representing the parameters of the instantaneous description of the simulated neuroid. The details are as follows.

C_i will consist of five different modules.

54.054.0100.056.0 10.078.0C_i 28.065.028.014.0δ–module 80.065.028.014.0λ–module 25.043.022.014.0Mo

Fig. 1: Data flow in module C_i simulating a single neuroid

First, there are three modules called *mode module, excitation module, and weight module*. The purpose of each of these modules is to represent a set of possible values of the respective quantity that represents, in order of their above enumeration, a possible mode of a neuroid, a possible value of the total excitation coming from the firings of adjacent neurons, and possible weights for all incoming edges.

Thus the mode module M_i consists of $|X|$ neurons. For each pair of form $(q, p) \in X$, with $q \in Q$ and $p \in T$, there is a corresponding neuron in M_i. Moreover, the neuron corresponding to the current mode of neuroid i is firing, while all the remaining neurons in M_i are in a quiescent state.

The weight module W_i consists of a two–dimensional array of neurons. To each incoming edge to i there is a row of $|W|$ neurons. Each row contains neurons corresponding to each possible value from the set W. If $(i, j) \in E$ is an incoming edge to i carrying the weight $w_{ij} \in W$, then the corresponding neuron in the corresponding row of W_i is firing.

The excitation module E_i consists of $O(|S|)$ neurons. Among them, at each time

only one neuron is firing, namely the one that corresponds to the current excitation w_i of i. Let $w_i = \sum_{\substack{k \text{ firing} \\ (k,i)\in E}} w_{k,i} = \sum_{\substack{j \\ (j,i)\in E}} f_j w_{ji}$ at that time. In order to compute w_i, we have to add only those weights that occur at the connections from currently firing neuroids. Therefore we shall first check all pairs of form $\{f_j; w_{ji}\}$ to see which weight value w_{ji} should participate in the computation of the total excitation. This will be done by dedicating special neurons t_{ijk} to this task, with k ranging over all weights in W. Each neuron t_{ijk} will receive 2 inputs. The first one from a neuron from the j-th row and k-th column in the weight module, which corresponds to some weight $w \in W$. This connection will carry weight w. The other connection will come from C_j and will carry the weight 1. Neuron t_{ijk} will fire iff j is firing and the current weight of connection $\{j, i\}$ is equal to w. In other words, t_{ijk} will fire iff its excitation equals exactly $w + 1$. This calls for implementing an equality test which requires the presence of some additional neurons, but we will skip the respective details. The outcomes from all t_{ijk} are then again summed and tested for equality against all possible excitation values. In this way the current value of w_i is determined eventually and the respective neurons serve as output neurons of the excitation module.

Besides these three modules there are two more modules that represent, and realize the transition functions δ and λ, respectively.

The δ–module contains one neuron for each set of arguments of the mode update function δ. Neuron d, responsible for the realization of the transition of form $\delta(s_i, w_i) = s'_i$, has its threshold equal to 2. Its incoming edges from each output neuron in M_i and from each output neuron from E_i carry the weight equal to 1. Clearly, d fires iff neurons corresponding to both quantities s_i and w_i fire. Firing of e will subsequently inhibit the firing of a neuron corresponding to s_i and excite the firing of a neuron corresponding to s'_i. Moreover, if the state corresponding to s'_i is a firing state of i then also a special neuron *out* in C_i is made to fire.

The λ–module is constructed n a similar way. It also contains one neuron per each set of arguments of the weight update function λ. The neuron ℓ responsible for the realization of the transition of form $\lambda(s_i, w_i, w_{ji}, f_j) = w'_{ji}$ has the threshold 4. Its incoming edges of weight 1 connect to it each output neuron in M_i, to each output neuron in W_i, to each neuron from the row corresponding to the j–th incoming edge of i, in E_i, and to the output from C_j. Clearly, ℓ fires iff neurons corresponding to all four quantities s_i, w_i, w_{ji}, and f_j fire. Firing of ℓ will subsequently inhibit the firing of a neuron corresponding to w_{ji} in W_i and excite the firing of a neuron corresponding to w'_{ji}, also in W_i.

Schematically, the topology of network C_i is sketched in Fig.1. For simplicity reasons only the flow of data is depicted by arrows.

The size of C_i is given by the sum of all sizes of all its modules. The whole net $\mathcal{N}'$ thus contains N mode–, excitation–, δ– and λ–modules, of size $|X|$, $|S|$, $|D|$, and $|L|$, respectively. Moreover, for each of M edges of $\mathcal{N}$ there is a complete row of $|W|$ neurons. This altogether leads to the size estimation as stated in the statement of the theorem.

□

From the previous theorem we can see that the size of a simulating neural network is larger than that of the original neuroidal network. It is linear in both the number of

neuroids and edges of the neuroidal network. The constant of proportionality depends linearly on the size of "program" of individual neuroids, and exponentially on the size of the universe of weights. However, note that the neural net constructed in the latter theorem is much smaller than that obtained via the direct simulation of the finite automaton corresponding to the simulated neuroidal net. A neural net, simulating the automaton from the proof of theorem 4.1, would be of size $\Theta(c^N)$ for some constant $c > 0$.

To summarize the respective results, we see that when comparing finite neuroidal nets to standard, non-programmable neural nets, the programmability of the former does not increase their computational power; it merely contributes to their greater descriptive expressiveness.

6 Integer weight neuroidal nets and Turing machines

Now we will show that in the case of integer weights there exist neuroidal nets of the finite size that can simulate any Turing machine.

Since we will be interested in space–bounded machines w.l.o.g. we will first consider a single–tape Turing machine in place of a simulated machine. In order to extend our results also for sublinear space complexities we will later also consider single–tape machines with separate input tapes.

First we show that even a single neuroid is enough for simulation of a single tape Turing machine.

Theorem 6.1 *Any $S(n)$–space and $T(n)$–time–bounded single tape Turing machine[2] can be simulated in time $O(T(n)S^2(n))$ by a single neuroid making use of integer weights of size $O(S(n))$ and of a simple arithmetic weight update function.*

Sketch of the proof: Since we are dealing with space–bounded Turing machines (TMs), w.l.o.g. we can consider only single–tape machines. Thus in what follows we will describe simulation of one computational step of a single–tape Turing machine $\mathcal{M}$ of space complexity $S(n)$ with tape alphabet $\{0, 1\}$. It is known (cf. [2]) that the tape of such a machine can be replaced by two stacks, S_L and S_R, respectively. The first stack holds the contents of $\mathcal{M}$'s tape to the left from the current head position while the second stack represents the rest of the tape. The left or the right end of the tape, respectively, find themselves at the bottoms of the respective stacks. Thus we assume that $\mathcal{M}$'s head always scans the top of the right stack. For technical reasons we will add an extra symbol 1 to the bottom of each stack. During its computation $\mathcal{M}$ updates merely the top, or pushes the symbols to, or pops the symbols from the top of these stacks.

With the help of a neuroid n we will represent machine $\mathcal{M}$ in a configuration described by the contents of its two stacks and by the state of the machine's finite

[2]Note that in the case of single tape machines the input size is counted into the space complexity and therefore we have $S(n)) \geq n$.

state control in the following way. The contents of both stacks will be represented by two integers v_L and v_R, respectively. Note that both $v_L, v_R \geq 1$ thanks to 1's at the bottoms of the respective stacks. The instantaneous state of $\mathcal{M}$ is stored in the states of n.

To simulate $\mathcal{M}$ we merely have to manipulate the above mentioned two stacks in a way that corresponds to the actions of the simulated machine. Thus, the net has to be able to read the top element of a stack, to delete it (to pop the stack), and to add (to push) a new element onto the top of the stack. W.r.t. our representation of a stack by an integer v, say, reading the top of a stack asks for determining the parity of v. Popping an element from or pushing it to a stack means computing of $\lfloor v/2 \rfloor$ and $2v$, respectively. All this must be done with the help of additions and subtractions.

The idea of the respective algorithm that computes the parity of any $v > 0$ is as follows. From v we will successively subtract the largest possible power of 2, not greater than v, until the value of v drops to either 0 or 1. Clearly, in the former case, the original value of v was even while in the latter case, it was odd.

More formally, the resulting algorithm looks as follows:

while $v > 1$ do $p1 := 1; p2 := 2;$
  while $p2 \leq v$ do $p1 := p1 + p1; p2 := p2 + p2$ od;
  $v := v - p1$
od;
if $v = 1$ then return(odd) else return(even) fi;

By a similar algorithm we can also compute the value of $\lfloor v/2 \rfloor$ (i.e., the value of v shifted by one position to the right, losing thus its rightmost digit). This value is equal to the sum of the halves of the respective powers (as long as they are greater than 1) computed in the course of previous algorithm.

The time complexity of both algorithms is $O(S^2(n))$.

The "neuroidal" implementation of previous algorithms looks as follows. The algorithms will have to make use of the values representing both stacks, v_L and v_R, respectively. Furthermore, they will need access to the auxiliary values $p1$, $p2$, v, and to the constant 1. All the previously mentioned values will be "stored" as weights of n. For technical reasons imposed by functionality restrictions of neuroids, which will become clearer later, we will need to store also the inverse values of all previously mentioned variables. These will be also stored in the weights of n.

Hencefore, the neuroid n will have 12 inputs. These inputs are connected to n via 12 connections. Making use of the previously introduced notation, the first six will hold the weights $w_1 = v_L$, $w_2 = v_R$, $w_3 = v$, $w_4 = p1$, $w_5 = p2$, and $w_6 = 1$. The remaining six will carry the same but inverse values.

The output of n is connected to all 12 inputs.

The neuroid simulates each move of $\mathcal{M}$ in a series of steps. Each series perform one run of the previously mentioned (or of a similar) algorithm and therefore consists of $O(S^2(n)$ steps.

At the beginning of the series that will simulate the $(t+1)$-st move of $\mathcal{M}$, the following invariant is preserved by n for any $t > 0$. Weights w_1 and w_2 represent the

contents of the stacks after the t-th move and $w_7 = -w_1$ and $w_8 = -w_2$. The remaining weights are set to zero.

At the beginning of computation, the left stack is empty and the right stack contains the input word of $\mathcal{M}$. We will assume that n will accept its input by entering a designated state. Also, the threshold of n will be set to 0 all the time.

Assume that at time t the finite control of $\mathcal{M}$ is in state q. Until its change, this state is stored in all forthcoming neuroidal states that n will enter.

In order to read the symbol from the top of the right stack the neuroid has to determine the last binary digit of w_2 or, in other words, it has to determine the parity of w_2. To do so, we first perform all the necessary initialization assignments to auxiliary variables, and to their "counterparts" holding the negative values. In order to perform the necessary tests (comparisons), the neuroid must enter a firing state. Due to the neuroidal computational mechanism and thanks to the connection among the output of n and all its inputs, all its non–zero weights will participate in the subsequent comparison of the total excitation against n's threshold. It is here that we will make a proper use of weights with the opposite sign: the weights (i.e., variables) that should not be compared, and should not be forgotten, will participate in a comparison with opposite signs.

For instance, to perform the comparison $p2 \leq v$, we merely "switch off" the positive value of $p2$ and the negative value of v from the comparison by temporarily setting the respective weights w_5 and w_9 to zero. All the other weight values remain as they were. As a result, after the firing step, n will compare $v - p2$ against its threshold value (which is permanently set to 0) and will enter a state corresponding to the result of this comparison. After the comparison, the weights set temporarily to zero can be restored to their previous values (by assignments $w_5 := -w_{11}$ and $w_9 := -w_3$).

The transition of $\mathcal{M}$ into a state as dictated by its transition function is realized by $\mathcal{N}$ after updating the stacks appropriately, by storing the respective machine state into the state of n. The simulation ends by entering into the final state.

It is clear that the simulation runs in time as stated in the theorem. The size of any stack, and hence of any variable, never exceeds the value $S(n) + 1$. Hence the size of the weights of n will be bounded by the same value.

□

Note that a similar construction, still using only one neuroid, would also work in case a multiple tape Turing machine should be simulated. In order to simulate a k–tape machine, the resulting neuroid will represent each tape by 12 weights as it did before. This will lead to a neuroid with $12k$ incoming edges.

Next we will also show that a simulation of an off–line Turing machine by a finite neuroidal network with unbounded weights is possible. This will enable us to prove a similar theorem as before which holds for arbitrary space complexities:

Theorem 6.2 *Let $\mathcal{M}$ be an off–line multiple tape Turing machine of space complexity $S(n) \geq 0$. Then $\mathcal{M}$ can be simulated in a cubic time by a finite neuroidal net that makes use of integer weights of size $O(S(n))$ and of a simple arithmetic weight update function.*

Sketch of the proof: In order to read the respective inputs the neuroidal net will be equipped with the same input tape as the simulated Turing machine. Except the neuroid n that takes care of a proper update of stacks that represent the respective machine tapes, the simulating net will contain also two extra neuroids that implement the control mechanism of the input head movement. For each move direction (left or right) there will be a special — so–called *move neuroid* — which will fire if and only if the input head has to move in a respective direction. The symbol read by the input head will represent an additional input to neuroid n simulating the moves of $\mathcal{M}$.

The information about the move direction will be inferred by neuroid n. As can be seen from the description of the simulation in the previous theorem, n keeps track on that particular transition of $\mathcal{M}$ that should be realized during each series of its steps simulating one move of $\mathcal{M}$.

Since s can transmit this information to the respective move neuroids only via firing and cannot distinguish between the two target neuroids, we will have to implement a (finite) counter in each move neuroid. The counter will count the number of firings of s occurring in an uninterrupted sequence. Thus at the end of the last step of $\mathcal{M}$'s move simulation (see the proof of the previous theorem) s will send two successive firings to denote the left move and three firings for the right move. The respective signals will reach both move neuroids, but with the help of counting they will find which of them is in charge for moving the head. Some care over synchronization of all three neuroids must be taken.

□

It is clear that the computations of finite neuroidal nets with integer weights can be simulated by Turing machines. Therefore the computational power of both devices is the same.

In 1995, Siegelmann and Sonntag [4] proved that the computational power of certain analog neural nets is equivalent to that of Turing machines. They considered finite neural nets with fixed rational weights. At time t, the output of their analog neuron i is a value between 0 and 1 which is determined by applying a so-called *piecewise linear activation* function ϕ to the excitation w_i of i at that time (see definition 2.1): $\phi : w_i \rightarrow \langle 0, 1 \rangle$. For negative excitation, ϕ takes the value 0, for excitation greater than 1 value 1, and for excitations between 0 and 1, $\phi(w_i) = w_i$. The respective net computes synchronously, in discrete time steps.

We will call the respective nets as *synchronous analog neural nets.*

Siegelmann and Sonntag's analog neural networks simulating a universal Turing machine consisted of 883 neurons. This can be compared with the simple construction from Theorem 5.1 requiring but a single neuroid. Nevertheless, the equivalency of both types of networks with Turing machines proves the following corollary:

Corollary 6.1 *Finite synchronous analog neural nets are computationally equivalent to finite neuroidal nets with integer weights.*

7 Real weight neuroidal nets and the additive BSS model

Now we will characterize the computational power of neuroidal nets with real parameters. We will compare their efficiency towards a restricted variant of the BSS model. The BSS model (cf. [1]) is a model that is similar to RAM which computes with real numbers under the unit cost model. In doing so, all four basic arithmetic operations of additions, subtractions, multiplication and division are allowed. The additive BSS model allows only for the former two arithmetic operations.

Theorem 7.1 *The additive real model of neuroidal nets is computationally equivalent to the additive BSS model working over binary inputs.*

Sketch of the proof: The simulation of a finite additive real model of neuroidal net $\mathcal{N}$ on the additive BSS model $\mathcal{B}$ is a straightforward matter.

For the reverse simulation, assume that the binary input to $\mathcal{N}$ is provided to $\mathcal{B}$ by a mechanism similar to that from Theorem 6.2. One must first refer to the theorem (Theorem 1 in Chapter 21 in [1]) that shows that by a suitable encoding a computation of any additive machine can be done using a fixed finite amount of memory (in a finite number of "registers", each holding a real number) without exponential increase in the running time. The resulting machine $\mathcal{F}$ is then simulated by $\mathcal{N}$ in the following way: The contents of finitely many registers of $\mathcal{F}$ are represented as (real) weights of a single neuroid r. Addition or subtraction of weights, as necessary, is done directly by weight update function. A comparison of weights is done with the help of r's threshold mechanism in a similar way to that in the proof of Theorem 6.1. To single out from the comparison the weights that should not be compared one can use a similar trick as in Theorem 6.1: to each such a weight a weight with the opposite sign is maintained. □

The power of finite additive neuroidal nets with real weights comes from their ability to simulate oracular or nonuniform computations. For instance, in [1] it is shown that the additive real BSS machines decide all binary sets in exponential time. Their polynomial time coincides with the nonuniform complexity class P/*poly*.

8 Conclusions

The paper brings a relatively surprising result showing computational equivalence between certain kinds of discrete programmable and analog finite neural nets. This result offers new insights into the nature of computations of neural nets.

First, it points to the fact that the ability of changing weights is not a condition *sine qua non* for learning. A similar effect can be achieved by making use of reasonably restricted kinds of analog neural nets.

Second, the result showing computational equivalency of the respective nets supports the idea that all reasonable computational models of the brain are equivalent (cf. [9]).

Third, for modeling of cognitive or learning phenomena, the neuroidal nets seem to be preferred over the analog ones, due to the transparency of their computational or learning mechanism. As far as their appropriateness for the task at hand is concerned, neuroidal nets with a finite set of weights seem to present the maximal functionality that can be achieved by living organisms. As mathematical models also more powerful variants are of interest.

Bibliography

[1] Blum, M. — Cucker, F. — Shub, M. — Smale, M.: Complexity and Real Computation. Springer, New York, 1997, 453 p.

[2] Hopcroft, J.E. — Ullman, J. D.: Formal Languages and their Relation to Automata. Addison–Wesley, Reading, Mass., 1969

[3] Indyk, P.: Optimal Simulation of Automata by Neural Nets. Proc. of the 12th Annual Symp. on Theoretical Aspects of Computer Science STACS'95, LNCS Vol. 900, pp. 337–348, 1995

[4] Siegelmann, H. T. — Sonntag, E.D.: On Computational Power of Neural Networks. *J. Comput. Syst. Sci.*, Vol. 50, No. 1, 1995, pp. 132–150

[5] Šíma, J. — Wiedermann, J.: Theory of Neuromata. Journal of the ACM, Vol. 45, No. 1, 1998, pp. 155–178

[6] Valiant, L.: Functionality in Neural Nets. Proc. of the 7th Nat. Conf. on Art. Intelligence, AAAI, Morgan Kaufmann, San Mateo, CA, 1988, pp. 629–634

[7] Valiant, L.G.: Circuits of the Mind. Oxford University Press, New York, Oxford, 1994, 237 p., ISBN 0–19–508936–X

[8] Valiant, L.G.: Cognitive Computation (Extended Abstract). In: Proc. of the 38th IEEE Symp. on Fond. of Comp. Sci., IEEE Press, 1995, p. 2–3

[9] Wiedermann, J.: Simulated Cognition: A Gauntlet Thrown to Computer Science. To appear in ACM Computing Surveys, 1999