On the Computational Power of Discrete Hopfield Nets

Pekka Orponen

Department of Computer Science, University of Helsinki
Teollisuuskatu 23, SF–00510 Helsinki, Finland
E-mail: orponen@cs.helsinki.fi

Abstract. We prove that polynomial size discrete synchronous Hopfield networks with hidden units compute exactly the class of Boolean functions PSPACE/poly, i.e., the same functions as are computed by polynomial space-bounded nonuniform Turing machines. As a corollary to the construction, we observe also that networks with polynomially bounded interconnection weights compute exactly the class of functions P/poly.

1 Background

Recurrent, or cyclic, neural networks are an intriguing model of massively parallel computation. In the recent surge of research in neural computation, such networks have been considered mostly from the point of view of two types of applications: pattern classification and associative memory (e.g. [15, 17, 19, 22]), and combinatorial optimization (e.g. [1, 6, 18]). Nevertheless, recurrent networks are capable also of more general types of computation, and issues of what exactly such networks can compute, and how they should be programmed, are becoming increasingly topical as work on hardware implementations progresses. (See, for instance, the several designs for analog, digital, hybrid, and optoelectronic implementations of recurrent networks contained in the compendium [30].) In this paper we address the task of comparing the computational power of the most basic recurrent network model, the *discrete Hopfield net* [17], to more traditional models of computation.

More precisely, our main interest is in the problem of Boolean function computation by undirected, or *symmetric* networks of weighted threshold logic units; but to get there, we first have to consider directed, or *asymmetric* nets.

In our model of network computation, the input to a net is initially loaded onto a set of designated input units; then the states of the units in the network are updated repeatedly, according to their local update rules until the network (possibly) converges to some stable global state, at which point the output is read from a set of designated output units. We only consider finite networks of units with binary states.

Since our networks in general contain cycles, their behavior may depend significantly on the update order of the units. (Note that we only deal with *discrete-time* networks here.) The updates may be *synchronous*, in which case all the units are updated simultaneously in parallel, or *asynchronous*, in which case the units are selected for updating one at a time in some order. We concentrate here on synchronous networks; not only because they are easier to deal with, but also because we do not want the notion of a function computed by a network depend on an arbitrary choice of an update order.

Following the early work of McCulloch and Pitts [24] and Kleene [21] (see also [25]), it has been customary to think of finite networks of threshold units as equivalent to finite automata. However, in Kleene's construction for this equivalence, the input to a net is given as a sequence of pulses, whereas from most of the current applications' point of view, it is more natural to think of all of the input as being loaded onto the network in the beginning of a computation. (Also, this input convention relates network computations more closely to standard circuit complexity theory [31].) Of course, this makes the network model nonuniform, as any single net operates on only fixed-length inputs, and to compute a function on an infinite domain one needs a *sequence* of networks.

Since a cyclic net of s units converging in time t may be "unwound" into an acyclic net of size $s \cdot t$, the class of Boolean functions computed by polynomial size, polynomial time cyclic asymmetric nets coincides with the class P/poly of functions computed by polynomial size circuits or, equivalently, polynomial time Turing machines with polynomially bounded, nonuniform "advice" [4, 20]. On the other hand, if the computation time of a cyclic network is not bounded, then it is fairly easy to see that the class of functions computed by polynomial size asymmetric nets equals the class PSPACE/poly of functions computed by polynomial space bounded Turing machines with polynomially bounded advice. (Parberry in [27] attributes this result to an early unpublished report [23], but for completeness we outline a proof in Section 3.) Unbounded computation time might not be a totally unreasonable modeling assumption when one considers systems with a potential for an extremely fast analog or optical implementations. Other characterizations of the class PSPACE/poly have been presented in [3].

Thus, the computational power of asymmetric nets is fairly easy to characterize. However, in the theory of neural computation, symmetric nets have attracted much more attention than general asymmetric ones, because of their very regular convergence behavior. The fundamental result here is Hopfield's [17] observation that a symmetric simple net (i.e, a net with no self-loops) using an asynchronous update rule always converges to a stable state. This was improved in [7] to show that in a symmetric simple net of p units with integer weights w_{ij}, the convergence requires at most a total of

$$3 \sum_{i,j} |w_{ij}| = O(p^2 \cdot \max_{i,j} |w_{ij}|)$$

unit state changes, under an asynchronous update rule. Under synchronous updates, a similar bound holds (and also for non-simple nets) [5, 8, 28]; but under synchronous updates the network may also converge to oscillate between two alternating states instead of a unique stable state.

Thus, in particular, symmetric networks with polynomially bounded weights converge in polynomial time. On the other hand, networks with exponentially large weights may indeed require an exponential time to converge, as was first shown in [11] for synchronous updates (a simplified construction appears in [9]), and in [13] for a particular asynchronous update rule. A network requiring exponential time to converge under any asynchronous update rule was demonstrated in [12]. Related work appears in [5, 10].

In this paper, we prove that despite their apparently very constrained behavior also symmetric polynomial size networks are capable of computing all functions in PSPACE/poly. The idea, presented in section 4, is to start with the simulation of space-bounded Turing machines by asymmetric nets, and then replace each of the asymmetric edges by a sequence of symmetric edges whose behavior is sequenced by *clock pulses*. The appropriate clock can be obtained from, e.g., the construction in [9] of a network that takes an exponential number of steps to converge. Of course, this clock network does not run forever (in this case it is not sufficient to have a network that simply oscillates between two states), but the exponentially long sequence of pulses it generates is sufficient to simulate a polynomially space-bounded computation. Applying the same technique to polynomial size nets with polynomially bounded weights shows that also in this case symmetric nets are computationally equivalent to asymmetric ones, i.e., capable of computing all of P/poly.

For a general introduction to neural computation, see the excellent textbook [16]; for aspects of recurrent networks, see [19]; and for computational complexity issues, see the survey papers [26, 27, 32].

2 Preliminaries

As in [27], we define a *(discrete) neural network* as a 6-tuple $N = (V, I, O, A, w, h)$, where V is a finite set of *units*, which we assume are indexed as $V = \{1, \ldots, p\}$; $I \subseteq V$ and $O \subseteq V$ are sets of *input* and *output units*, respectively; $A \subseteq V$ is a set of *initially active units*, of which we require that $A \cap I = \emptyset$; $w : V \times V \to Z$ is the *edge weight matrix*, and $h : V \to Z$ is the *threshold vector*. The *size* of a network is its number of units, $|V| = p$, and the *weight* of a network is defined as its sum total of edge weights, $\sum_{i,j \in V} |w_{ij}|$. A *Hopfield net (with hidden units)* is a neural network N whose weight matrix is symmetric, i.e. $w_{ij} = w_{ji}$ for all $i, j \in V$.

Given a neural network N, let us denote $|I| = n$, $|O| = m$; moreover, assume that the units are indexed so that the input units appear at indices 1 to n. The network then computes a partial mapping

$$f_N : \{0,1\}^n \to \{0,1\}^m$$

as follows. Given an input x, $|x| = n$, the "states" s_i of the input units are initialized as $s_i = x_i$. The states of the units in set A are initialized to 1, and the states of the remaining units are initialized to 0. Then new states s'_i, $i = 1, \ldots, p$, are computed simultaneously for all the units according to the rule

$$s'_i = \operatorname{sgn}\left(\sum_{j=1}^{p} w_{ij} s_j - h_i\right),$$

where $\operatorname{sgn}(t) = 1$ for $t \geq 0$, and $\operatorname{sgn}(t) = 0$ for $t < 0$. (Thus, we only consider synchronous updating.) This updating process is repeated until no more changes occur, at which point we say that the network has *converged*, and the output value $f_N(x)$ can be read off the output units (in order). If the network does not converge on input x, the value $f_N(x)$ is undefined.

For simplicity, we consider from now on only networks with a single output unit; the extensions to networks with multiple outputs are straightforward. The *language recognized* by a single-output network N, with n input units, is defined as

$$L(N) = \{x \in \{0,1\}^n | f_N(x) = 1\}.$$

Given a language $A \subseteq \{0,1\}^*$, denote $A^{(n)} = A \cap \{0,1\}^n$. We consider the following complexity classes of languages:

$$\text{PNETS} = \{A \subseteq \{0,1\}^* | \text{ for some polynomial } q \text{, there is for each } n \text{ a network of size at most } q(n) \text{ that recognizes } A^{(n)} \},$$

$$\text{PNETS(symm)} = \{A \subseteq \{0,1\}^* | \text{ for some polynomial } q \text{, there is for each } n \text{ a Hopfield net of size at most } q(n) \text{ that recognizes } A^{(n)} \},$$

$$\text{PNETS(symm,small)} = \{A \subseteq \{0,1\}^* | \text{ for some polynomial } q \text{, there is for each } n \text{ a Hopfield net of weight at most } q(n) \text{ that recognizes } A^{(n)} \}.$$

Let $\langle x, y\rangle$ be some standard pairing function mapping pairs of binary strings to binary strings (see, e.g. [4, p. 7]). A language $A \subseteq \{0,1\}^*$ belongs to the nonuniform complexity class PSPACE/poly ([3], [4, p. 100], [20]), if there are a polynomial space bounded Turing machine M, and an "advice" function $f : N \to \{0,1\}^*$, where for some polynomial q and all $n \in N$, $|f(n)| \leq q(n)$, and for all $x \in \{0,1\}^*$,

$$x \in A \Leftrightarrow M \text{ accepts } \langle x, f(|x|)\rangle.$$

The class P/poly is defined analogously, using polynomial time instead of space bounded Turing machines.

3 Simulating Turing machines with asymmetric nets

Simulating space-bounded Turing machines with asymmetric neural nets is fairly straightforward.

Theorem 1. PNETS = PSPACE/poly.

Proof. To prove the inclusion PNETS $\subseteq$ PSPACE/poly, observe that given (a description of) a neural net, it is possible to simulate its behavior *in situ*. Hence, there exists a universal neural net interpreter machine M that given a pair $\langle x, N\rangle$ simulates the behavior of net N on input x in linear space. Let then language A be recognized by a polynomial size bounded sequence of nets (N_n). Then $A \in$ PSPACE/poly via the machine M and advice function $f(n) = N_n$.

To prove the converse inclusion, let $A \in$ PSPACE/poly via a machine M and advice function f. Let the space complexity of M on input $\langle x, f(|x|)\rangle$ be bounded by a polynomial $q(|x|)$. Without loss of generality (see, e.g. [4]) we may assume that M has only one tape, halts on any input $\langle x, f(|x|)\rangle$ in time $c^{q(|x|)}$, for some constant c, and indicates its acceptance or rejection of the input by printing a 1 or a 0 on the first square of its tape.

Following the standard simulation of Turing machines by combinational circuits [4, pp. 106–112], it is straightforward to construct for each n an *acyclic* network that

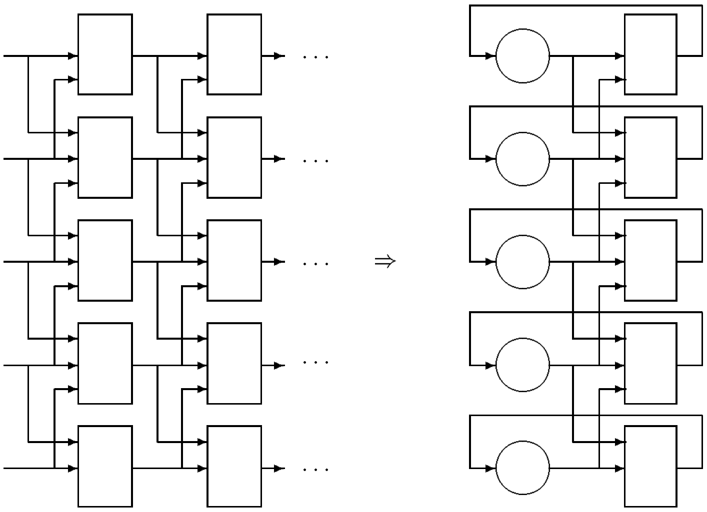

Fig. 1. Simulation of a space bounded Turing machine by an asymmetric cyclic net.

simulates the behavior of M on inputs of length n. (More precisely, the network simulates computations $M(\langle x, f(n)\rangle)$, where $|x| = n$.) This network consists of $c^{q(n)}$ "layers" of $O(q(n))$ parallel wires, where the tth layer represents the configuration of the machine M at time t (Fig. 1, left). Each two consecutive layers of wires are interconnected by an intermediate layer of $q(n)$ constant-size subcircuits, each implementing the local transition rule of machine M at a single position of the simulated configuration. The input x is entered to the network along input wires; the advice string $f(n)$ appears as a constant input on another set of wires; and the output is read from the particular wire at the end of the circuit that corresponds to the first square of the machine tape.

An important observation is now that the interconnection patterns between layers are very uniform: all the local transition subcircuits are similar, with a structure that depends only on the structure of M, and their number depends only on the length of x. Hence we may replace the exponentially many consecutive layers in the acyclic circuit by a single transformation layer that feeds back on itself (Fig. 1, right). The size of the cyclic network thus obtained is then only $O(q(n))$. When initialized with input x loaded onto the appropriate input units, and advice string $f(n)$ mapped to the appropriate initially active units, the network will converge in $O(c^{q(n)})$ update steps, at which point the output can be read off the unit corresponding to the first square of the machine tape.

$\square$

4 Simulating asymmetric nets with symmetric nets

Having now shown how to simulate polynomial space-bounded Turing machines by polynomial-size asymmetric nets, the main remaining problem is how to simulate the asymmetric edges in a network by symmetric ones. This is not possible in general, as is shown by the different convergence behaviors of asymmetric and symmetric nets. However, in the special case of *convergent* computations of asymmetric nets the simulation can be achieved.

Theorem 2. PNETS(symm) = PSPACE/poly.

Proof. Because PNETS(symm) $\subseteq$ PNETS, and by the previous theorem PNETS $\subseteq$ PSPACE/poly, it suffices to show the inclusion PSPACE/poly $\subseteq$ PNETS(symm).

Given any $A \in$ PSPACE/poly, there is by Theorem 1 a sequence of polynomial size asymmetric networks recognizing A. Rather than show how this specific sequence of networks can be simulated by symmetric networks, we shall show how to simulate the convergent computations of an *arbitrary* asymmetric network of n units and e edges of nonzero weight on a symmetric network of $O(n+e)$ units and $O(n^2)$ edges.

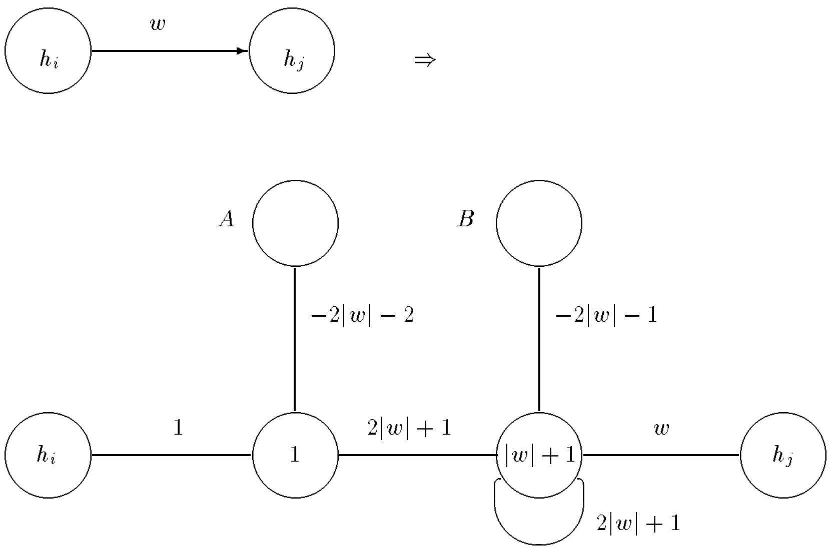

Fig. 2. A sequence of symmetric edges simulating an asymmetric edge of weight w.

The construction combines two network "gadgets": a simplified version of a mechanism due to Hartley and Szu [14] for simulating an asymmetric edge by a sequence of symmetric edges and their interconnecting units, whose behavior is coordinated by a system clock (Figs. 2, 3); and a binary counter network due to Goles and

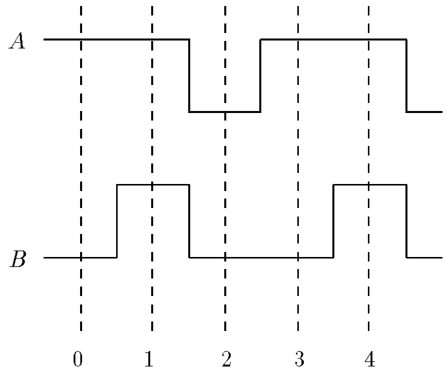

Fig. 3. The clock pulse sequence used in the edge simulation of Fig. 2.

Martínez ([9], [10, pp. 88–95]) that can count up to 2^n using about $3n$ units and $O(n^2)$ symmetric edges (Fig. 4). An important observation here is that any convergent computation by an asymmetric network of n units has to terminate in 2^n update steps, because otherwise the network repeats a configuration and goes into a loop; hence, the exponential counter network can be used to provide a sufficient number of clock pulses for the simulation to be performed.

Let us first consider the gadget for a symmetric simulation of an asymmetric edge of weight w from a unit i to a unit j (Fig. 2) [14]. Here the idea is that the two units inserted between the units i and j function as locks in a canal, permitting information to move only from left to right. The locks are sequenced by clock pulses emanating from the units labeled A and B, in cycles of length three as presented in Fig. 3.

At time 0 clock A is on, setting the first intermediate unit to zero, and clock B is off, permitting the state of the second intermediate unit to influence the state computation at unit j (at time 1). At time 1 clock B turns on, clearing the second unit (note that the connection from unit j is not strong enough to turn the intermediate unit back on). At time 2, clock A turns off, permitting a new state to be copied from unit i to the first intermediate unit. At time 3, clock A turns on again, clearing the first unit; but simultaneously the new state is copied to the second intermediate unit, from where it can then influence the computation of the new state of unit j (at time 4).

The next question is how to generate the clock pulses A and B. It is not possible to construct a symmetric clock network that runs forever. (At best such a network can end up oscillating between two states, but this is not sufficient to generate the period 3 pulse sequences required for the previous construction.) However, Fig. 4 presents the first two stages in the construction of a $(3n-4)$-unit symmetric network with a convergence time of 2^n synchronous update steps [9, 10]. The idea here is that the n units in the upper row implement a binary counter, counting from all 0's to all 1's (in the figure, the least significant bit is to the right). For each "counter" unit added to the upper row, after the two initial ones, two "control" units are added to

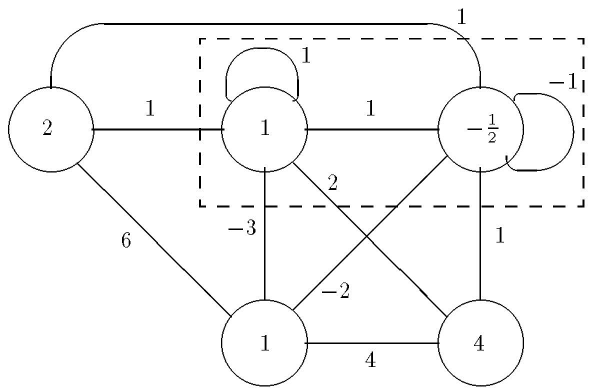

Fig. 4. The first two stages in the construction of a binary counter network.

the lower row. The purpose of the latter is to first turn off all the "old" units, when the new counter unit is activated, and from then on balance the input coming to the old units from the new units, so that the old units may restart counting from zero.

It is now possible to derive from this counter network a sufficient number of the A and B pulse sequences by means of the delay line network presented in Fig. 5. Here the unit at the upper left corner is some sufficiently slow oscillator; since we require pulse sequences of period 3, this could be the second counter unit, which has a period of ≥ 4 update steps, in the preceding construction. (Thus, to simulate a computation of length 2^n, we actually need a 2^{n+2}-counter.) The delay line operates as follows: when the oscillator unit turns on, it "primes" the first unit in the line; but nothing else happens until the oscillator turns off. At that point the "on" state begins to travel down the line, one unit per update step, and the pulses A and B are derived from the appropriate points in the line.

The value M used in the construction has to be chosen so large that the states of the units in the underlying network have no effect back on the delay line. It is sufficient to choose M larger than the total weight of that network. Similarly, the weights and thresholds in the counter network have to modified so that the connection to the delay line does not interfere with the counting. Assuming that $M \geq 3$, it is here sufficient to multiply all the weights and thresholds by $6M$, and then subtract one from each threshold.

□

Concerning the edge weights in the above constructions, one can see that in the network implementing the machine simulation (Figs. 1, 2), the weights actually are bounded by some constant that depends only on the simulated machine M; in the delay line, the weights are proportional to the total weight of the underlying network; and the weights in the counter network (Fig. 4) are proportional to the length of the required simulation (and, less significantly, to the weight of the delay line). Thus, we obtain as a corollary to the construction that if the simulated Turing machine (or, more generally, asymmetric network) is known to converge in polynomial time, then

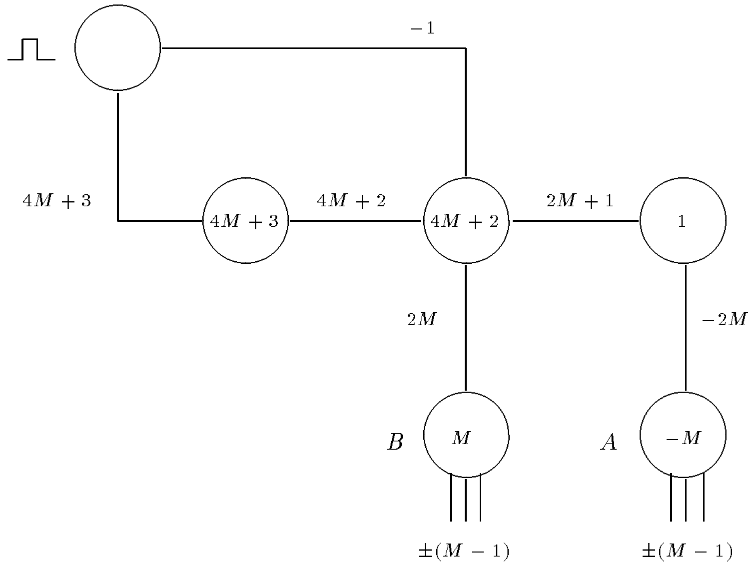

Fig. 5. A delay line for generating clock pulses from the binary counter network in Fig. 4.

it is sufficient to have polynomially bounded weights in the simulating symmetric network. Formulating this in terms of nonuniform complexity classes, we obtain:

Corollary 3. PNETS(symm,small) = P/poly.

□

5 Open problems

Two open problems remaining from this work are the following:

1. In Hopfield's original paper [17], and much of subsequent work, asynchronous update rules are used. Are also *asynchronous* Hopfield nets capable of computing all of PSPACE/poly? Or, with the small weights restriction, all of P/poly? The synchronous Goles/Martínez counting network could possibly be replaced by an asynchronous counter due to A. Haken [12], but it is not clear how to effect the rest of the construction.
2. Also in Hopfield's original model, all the units are used for both input and output, and no hidden units are allowed. Although this is a somewhat artificial restriction from the function computation point of view, it would nevertheless be of interest to neural network theory to characterize the class of mappings computable by such networks.

Acknowlegdment

I wish to thank Mr. Juha Kärkkäinen for improving my initial attempts at simplifying the Hartley/Szu network, and suggesting the elegant construction presented in Fig. 2.

References

1. Aarts, E., Korst, J. *Simulated Annealing and Boltzmann Machines.* John Wiley & Sons, Chichester, 1989.
2. Anderson, J. A., Rosenfeld, E. (eds.) *Neurocomputing: Foundations of Research.* The MIT Press, Cambridge, MA, 1988.
3. Balcázar, J. L., Díaz, J., Gabarró, J. On characterizations of the class PSPACE/poly. *Theoret. Comput. Sci. 52* (1987), 251–267.
4. Balcázar, J. L., Díaz, J., Gabarró, J. *Structural Complexity I.* Springer-Verlag, Berlin Heidelberg, 1988.
5. Bruck, J. On the convergence properties of the Hopfield model. *Proc. of the IEEE 78* (1990), 1579–1585.
6. Durbin, R., Willshaw, D. An analogue approach to the Travelling Salesman Problem using an elastic net method. *Nature 326* (1987), 689–691.
7. Fogelman, F., Goles, E., Weisbuch, G. Transient length in sequential iterations of threshold functions. *Discr. Appl. Math. 6* (1983), 95–98.
8. Goles, E., Fogelman, F., Pellegrin, D. Decreasing energy functions as a tool for studying threshold networks. *Discr. Appl. Math. 12* (1985), 261–277.
9. Goles, E., Martínez, S. Exponential transient classes of symmetric neural networks for synchronous and sequential updating. *Complex Systems 3* (1989), 589–597.
10. Goles, E., Martínez, S. *Neural and Automata Networks.* Kluwer Academic, Dordrecht, 1990.
11. Goles, E., Olivos, J. The convergence of symmetric threshold automata. *Info. and Control 51* (1981), 98–104.
12. Haken, A. Connectionist networks that need exponential time to converge. Manuscript, 10 pp., January 1989.
13. Haken, A., Luby, M. Steepest descent can take exponential time for symmetric connection networks. *Complex Systems 2* (1988), 191–196.
14. Hartley, R., Szu, H. A comparison of the computational power of neural networks. In: *Proc. of the 1987 Internat. Conf. on Neural Networks, Vol. 3.* IEEE, New York, 1987. Pp. 15–22.
15. Hinton, G. E., Sejnowski, T. E. Learning and relearning in Boltzmann machines. In [29], pp. 282–317.
16. Hertz, J., Krogh, A., Palmer, R. G. *Introduction to the Theory of Neural Computation.* Addison-Wesley, Redwood City, CA, 1991.
17. Hopfield, J. J. Neural networks and physical systems with emergent collective computational abilities. *Proc. Nat. Acad. Sci. USA 79* (1982), 2554–2558.
18. Hopfield, J. J., Tank, D. W. "Neural" computation of decisions in optimization problems. *Biol. Cybern. 52* (1985), 141–152.
19. Kamp, Y., Hasler, M. *Recursive Neural Networks for Associative Memory.* John Wiley & Sons, Chichester, 1990.
20. Karp, R. M., Lipton, R. J. Turing machines that take advice. *L'Enseignement Mathématique 28* (1982), 191–209.

21. Kleene, S. C. Representation of events in nerve nets and finite automata. In: *Automata Studies* (ed. C. E. Shannon, J. McCarthy). Annals of Mathematics Studies n:o 34. Princeton Univ. Press, Princeton, NJ, 1956. Pp. 3–41.
22. Kohonen, T. *Self-Organization and Associative Memory.* Springer-Verlag, Berlin Heidelberg, 1989 (3rd Edition).
23. Lepley, M., Miller, G. Computational power for networks of threshold devices in an asynchronous environment. Unpublished manuscript, Dept. of Mathematics, Massachusetts Inst. of Technology, 1983.
24. McCulloch, W. S., Pitts, W. A logical calculus of the ideas immanent in nervous activity. *Bull. Math. Biophys. 5* (1943), 115–133. Reprinted in [2], pp. 18–27.
25. Minsky, M. L. *Computation: Finite and Infinite Machines.* Prentice-Hall, Englewood Cliffs, NJ, 1972.
26. Orponen, P. Neural networks and complexity theory. In: *Proc. of the 17th Internat. Symp. on Mathematical Foundations of Computer Science* (ed. I. M. Havel, V. Koubek). Lecture Notes in Computer Science 629, Springer-Verlag, Berlin Heidelberg, 1992. Pp. 50–61.
27. Parberry, I. A primer on the complexity theory of neural networks. In: *Formal Techniques in Artificial Intelligence: A Sourcebook* (ed. R. B. Banerji). Elsevier – North-Holland, Amsterdam, 1990. Pp. 217–268.
28. Poljak, S., Sura, M. On periodical behaviour in societies with symmetric influences. *Combinatorica 3* (1983), 119–121.
29. Rumelhart, D. E., McClelland, J. L., et al. (eds.) *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. 1.* The MIT Press, Cambridge, MA, 1986.
30. Sánchez-Sinencio, E., Lau, C. *Artificial Neural Networks: Paradigms, Applications, and Hardware Implementations.* IEEE Press, New York, 1992.
31. Wegener, I. *The Complexity of Boolean Functions.* John Wiley & Sons, Chichester, and B. G. Teubner, Stuttgart, 1987.
32. Wiedermann, J. Complexity issues in discrete neurocomputing. In: *Aspects and Prospects of Theoretical Computer Science. Proc. of the 6th Meeting of Young Computer Scientists* (ed. J. Dassow, J. Kelemen). Lecture Notes in Computer Science 464, Springer-Verlag, Berlin Heidelberg, 1990. Pp. 93–108.