Complex Systems **4** (1990) 519–541

Learning by CHIR without Storing Internal Representations

Dimitry Nabutovsky
Tal Grossman
Eytan Domany
Department of Electronics, Weizmann Institute of Science, Rehovot 76100 Israel

Abstract. A new learning algorithm for feedforward networks, learning by choice of internal representations (CHIR), was recently introduced [1,2]. Whereas many algorithms reduce the learning process to minimizing a cost function over the *weights*, our method treats the *internal representations* as the fundamental entities to be determined. The algorithm applied a search procedure in the space of internal representations, together with cooperative adaptation of the weights (e.g., by using perceptron learning). Tentative guesses of the internal representations, however, had to be stored in memory. Here we present a new version, CHIR2, which eliminates the need to store internal representations and at the same time is faster than the original algorithm. We first describe a basic version of CHIR2, tailored for networks with a single output and one hidden layer. We tested it on three problems — contiguity, symmetry, and parity — and compared its performance with backpropagation. For all these problems our algorithm is 30–100 times faster than backpropagation, and, most significantly, learning time increases more slowly with system size. Next, we show how to modify the algorithm for networks with many output units and more than one hidden layer. This version is tested on the combined parity+symmetry problem and on the random associations task. A third modification of the new algorithm, suitable for networks with binary weights (all weights and thresholds are equal to ± 1), is also described, and tests of its performance on the parity and the random teacher problems are reported.

1. Introduction

Studies of different learning algorithms for feedforward neural networks constitute a very active and interesting field of research on connectionist models. The simplest implementation of such a network is the single-layer perceptron

© 1990 Complex Systems Publications, Inc.

introduced by Rosenblatt [3], who also proposed a learning algorithm that was guaranteed to converge to a solution (if one existed). A related learning rule was introduced by Widrow and Hoff [4]. However, Minsky and Papert [5] have demonstrated that simple single-layer perceptrons have very limited applicability. On the other hand, it is known that multilayer perceptrons can, given a sufficient number of units, realize any binary input–output relation (predicate). Backpropagation is a learning algorithm for multilayer perceptrons, proposed [6] and more recently investigated by several groups [7,8]. It uses basic units with continuous variables and a smooth sigmoid input–output response function. An error function, which measures the deviation of actual from desired output, is minimized by the algorithm. There is no convergence theorem for backpropagation: like any minimization procedure, it may find a local minimum that does not correspond to a solution. Trying to avoid this problem, by using *simulated annealing* [9], for example, is extremely time-consuming.

Grossman, Meir, and Domany introduced an algorithm for networks with binary valued neurons and a single hidden layer, which learns by choosing internal representations (CHIR) [1]. CHIR shifts the emphasis of the learning procedure from adjusting weights to finding good internal representations (i.e., the states taken by the hidden layer in response to a presented input). This basic idea was subsequently adopted and extended by various groups. It was incorporated in processes that minimize cost functions (over internal representations) for networks with continuous [10] as well as discrete [11] variables. CHIR was also modified to treat multiple outputs and more than one hidden layer [2] and networks with binary weights [12].

The idea of manipulating internal representations, to be followed by adjusting weights, was adopted also by groups [13] that study the problem of learning in networks with flexible architectures. It should be noted that these algorithms use a "global control," that is, they need a unit that "knows" the state of all neurons, all weights, and the learning process requires adding an uncontrolled number of new neurons.

CHIR, on the other hand, works for perceptrons with a fixed architecture, does not use "global control," and does not change the number of neurons during learning. CHIR (and its variants) outperforms backpropagation, but needs an extra memory of PH bits, where P is the number of patterns in the training set and H is the number of hidden units. This need arises since CHIR learns the weights by the perceptron learning rule (PLR). To adjust a weight w_{ij} by this method, one must know the state of the source neuron j and the desired state of the target neuron i. The correct states of the hidden layer (i.e., the internal representations) are not known. When CHIR learns the weights that connect the hidden layer to the output, the current guess for the internal representation can be generated simply by presenting an input to the network. This is not so in the part of the CHIR process that learns the weights connecting the input to the hidden layer. In this part, the internal representations cannot be recovered simply, using only stored weights. Therefore CHIR stores a table of the "current guess" for internal

representations. This table is modified during the learning process. The need to store the entire table of internal representations makes CHIR biologically implausible and may be practically limiting.

In this paper we introduce a method for learning by choosing internal representations without storing them. The central idea is to present the training set pattern by pattern and, if the answer is wrong, to change the internal representation *of the currently presented pattern* by changing weights. This ensures that even though only current weights are stored, the internal representation can be retrieved.

We may, however, introduce errors into this retrieval process, since the current change of weights may affect the internal representations of some previously treated pattern. We try not to disturb the internal representation for other patterns, by using a minimal disturbance principle. It is related to the MRII algorithm of Widrow and Winter [14] and to the one used by Mitchison and Durbin [15]. The main conceptual difference is that all these algorithms change weights in a favorable direction, without ensuring that the internal representation actually changes, whereas we change weights so that the internal representation for the currently presented pattern is certainly modified. Another important difference is a definition of minimal disturbance. In references [14,15], minimal disturbance means a minimal *changing of weights*, whereas in our procedure it defined as a minimal *probability to change an internal representation* for a random pattern.

The outline of this paper is as follows. In section 2 we describe the algorithm; a brief outline of the strategy is followed by a detailed presentation of the single-output version. In section 3 we report tests of its performance on the standard benchmark problems of contiguity, symmetry, and parity. Section 4 describes a generalized version of an algorithm, suitable for multiple-output (and multilayered) networks. In section 5 we further modify the algorithm so it can train networks with binary valued weights. These versions are also tested and the results are presented. Finally, our work is summarized and discussed in section 6.

2. Description of the algorithm

Consider a two-layer perceptron, with N input, H hidden, and 1 output unit (see figure 1). The elements of network are binary linear threshold units, whose states are determined according to

$$S_i = \text{sign} X_i, \quad X_i = \sum_{j=0}^{N} w_{ij} s_j \tag{2.1}$$

$$S = \text{sign} X, \quad X = \sum_{i=0}^{H} W_i S_i \tag{2.2}$$

Here w_{ij} are weights assigned to connections from input to hidden layer, the weight W_i connects hidden unit i to the output. w_{i0} and W_0 are, respectively,

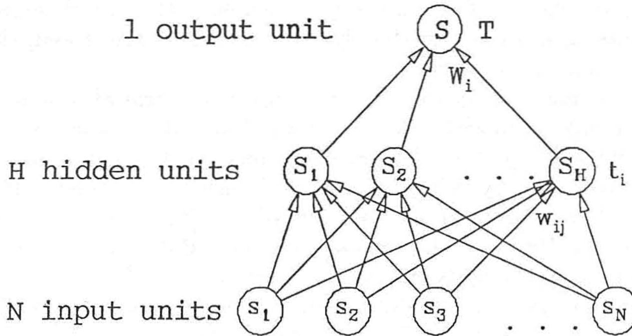

Figure 1: Feedforward network with layered architecture. State of cell i at layer $l+1$ is determined by the states of layer l. Input patterns are presented to a bottom layer, output is read out from the top.

biases of the hidden and output units, with $s_0 = S_0 = 1$. For $i > 0$ the variables $s_i/S_i/S$ denote the states taken by the input/hidden/output units. During a training session, the input units are set in any one of $\mu = 1 \ldots P$ patterns, i.e., $s_j = \xi_j^\mu$. In a typical task for such a network, P specified answers, $S^\mu = \xi^\mu$ are required in response to the P input patterns.

2.1 A short description of the algorithm

The learning process (described in detail in section 2.2) starts out by setting w_i and W_{ij} randomly. We adopt the basic strategy of CHIR, in that our procedure alternates between two learning stages:

Learn23 The hidden layer serves as source, and the output as the target unit of the perceptron learning rule (PLR), used to learn the W_i. For this stage, storage of the internal representations is *not* required. For fixed w_{ij} we present a pattern μ as input (i.e., set $s_j = \xi_j^\mu$). The resulting state of the hidden layer, as obtained from (2.1), is the internal representation of pattern μ. This specifies the state of the *source* units, used by the PLR to search for appropriate weights W_i, to obtain the desired outputs $S = \xi^\mu$. If the PLR finds such W_i, we stop; the complete learning problem has been solved. Otherwise we stop after I_{23}

learning sweeps, keep the current weights, and turn to the next stage, *learn12*.

Learn12 While the current values of W_i remain fixed, apply a learning process to w_{ij}. Using the PLR at this state is problematic, since only the states of the *source* units, i.e., the input of network, is known. The corresponding target state (of the hidden layer) is not known. CHIR overcomes this problem by *storing the current guess* for the internal representations of all patterns in the training set. This is the storage requirement that we succeed in eliminating in the present paper. Here we learn the w_{ij} by presenting the training set sequentially, pattern by pattern. If we get a wrong answer, we choose a hidden neuron that gives a wrong contribution to the field acting on the output unit. This neuron is chosen according to a minimal disturbance principle, described in the next section. The state (± 1) of this unit (obtained in response to the current input) is flipped by modifying the weights w_{ij} incident on it, using an Abbott–Kepler linear rule [16], which ensures that the internal representation is actually changed. Note that the main difference between the present algorithm and CHIR is that here the internal representation is modified *during* the process of learning the weights w_{ij}.

We periodically check whether the network has achieved error-free performance for the entire training set; if it has, learning is completed and a solution of the problem has been found. If no solution has been found after I_{12} sweeps of the training set, we abort the learn12 stage, restart the cycle with learn23, and so on.

2.2 Detailed description of the algorithm

Initialize $-1 < W_i, w_{ij} < 1$ randomly, and normalize:

$$W_i := \frac{W_i}{\sqrt{\sum_i W_i^2}} \quad ; \quad w_{ij} := \frac{w_{ij}}{\sqrt{\sum_j w_{ij}^2}} \tag{2.3}$$

Learn23: We attempt to find a solution for an existing internal representation. Training patterns are presented sequentially. For every training pattern $s_j^\mu = \xi_j^\mu$ we determine the states of the hidden and output units, S_i and S, according to (2.1,2.2), and change the weights W_i (see figure 1) using a perceptron learning rule:

$$\Delta W_i = \frac{\eta}{H+1} \cdot \frac{1 - S^\mu \xi^\mu}{2} \cdot \xi^\mu S_i^\mu \tag{2.4}$$

where ξ^μ is the desired output for pattern μ, H is the number of hidden units, $\eta > 0$ is a step size parameter. Note that the PLR modifies weights only when presentation of input μ produces an erroneous output $S^\mu \neq \xi^\mu$. When that happens, each W_i is changed, in a Hebbian fashion, toward values that correct the error. After every learning step we normalize according to equation (2.3).

After every complete sweep of the training set we ensure that the output S actually depends on the internal representation, that is, that $|W_0| < \sum_{i>0} |W_i|$. If this is not satisfied, we set $W_0 := \frac{W_0}{|T|} \sum_{i>0} |W_i|$.

If a solution is found, we stop. Otherwise we stop after I_{23} learning sweeps, keep the current weights, and go to the next stage.

Learn12: In this stage we try to learn the w_{ij} for fixed W_i. Again, we present input patterns μ one by one and for every pattern determine S_i and S according to (2.1,2.2). If the resulting output S equals the desired output ξ^μ, present the next pattern. If, however, $S = -\xi^\mu$, we choose *one* of the hidden neurons i, and change its incident weights and threshold. To decide which neuron to select, we first check how flipping its state (i.e., $S_i^\mu \to -S_i^\mu$) affects the field that acts on the output unit, $X^\mu = \sum W_j S_j^\mu$. Since the output for pattern μ is wrong, X^μ has the wrong sign. Flipping the state of some hidden units will pull X^μ in the right direction. These hidden units are the candidates from which we select the one to be flipped. Once we decide on this unit i, we change its incident weights W_{ij} to an extent that *ensures* that when input μ is presented, the sign of unit i will be the opposite of what it was before. In order to achieve this, the size of Δw_{ij} must be large enough to flip the sign of the field incident on i, $X_i^\mu = \sum_j w_{ij}\xi_j^\mu$. Hence the size of our learning step must increase with $|X_i^\mu|$ (see equation 2.6 below). One should bear in mind, however, that changing w_{ij} may induce an error to the response to a pattern that was presented previously, by affecting *its* internal representation. To minimize the number of such occurrences, we select the neuron i according to a *minimal disturbance principle.*

References [14,15] also use such a principle, but they aim at generating minimal change of *weights*, $(\Delta\vec{w}_i)^2$. We, on the other hand, define as minimal disturbance a learning step that minimizes the probability to change the *internal representation* for a *random* input pattern, $s_i^r = \pm 1$. For $N \gg 1$ such a pattern generates on hidden unit i a normally distributed field, with mean $\langle X_i^r \rangle = 0$ and variance $\sigma^2 = \sum_j w_{ij}^2$.

Similarly, the change of input field, ΔX_i^r, induced by a PLR step (which was taken in response to training pattern μ), is normally distributed with mean 0 and standard deviation

$$\sqrt{\sum_j (\Delta w_{ij})^2}$$

Since the random pattern r and the training pattern μ are not correlated, clearly X_i^r and ΔX_i^r are also uncorrelated, and it is easy to see that minimal disturbance means minimal $[(\Delta\vec{w}_i)^2]/(\vec{w}_i^2)$.

Suppose now that all $\vec{w}$'s are normalized. Then the minimal disturbance principle means just a minimal $(\Delta\vec{w}_i)^2$. As explained above, the size of our learning step $|\Delta\vec{w}_i|$ increases with X_i, minimal $|\Delta\vec{w}_i|$ means also minimal X_i, i.e., units with minimal input field to be chosen to learn.

In this sense our minimal disturbance principle is similar to that applied in [14,15]. However, equivalence of the latter to our procedure holds only if the weights are indeed normalized, which, apparently, is not done by the

above-mentioned methods. Note that choosing the neuron to be modified on the basis of small field, without normalizing the weights, may cause an instability that can give rise to exponential decrease of the weights incident on some neurons. The reason is that application of the perceptron learning rule to neuron i may decrease its incident weights w_{ij}. If, however, w_{ij}^2 becomes smaller, we are more likely to pick neuron i again as the one that causes minimal disturbance, and so on. Decisions based on the size of X_i make sense only if normalization is included in the procedure.

In practice we choose stochastically the neuron whose weights are to be modified. That is, the probability of neuron i to be chosen is given by

$$p_i = \frac{\theta(-W_i S_i S) e^{-\beta |X_i|}}{\sum_j \theta(-W_j S_j S) e^{-\beta |X_j|}} \tag{2.5}$$

where the θ-function in the numerator ensures that we flip only units that pull the output in the wrong direction. β is an inverse-temperature-like parameter that determines how often (2.5) chooses a neuron whose input field $|X|$ is not minimal. The probability of such an event becomes smaller as β is increased. When $\beta \to \infty$, (2.5) always chooses the best neuron according to the minimal disturbance principle, but sequential presentation of patterns may cause such a deterministic algorithm to enter a cycle. Such a cycle may occur when there are only a few errors to be corrected, e.g., when two patterns alternate in changing the same hidden unit back and forth.

We choose β to equal $N_{\rm err}$, the number of errors in the last sweep. This means that when there are many errors we usually follow the minimal disturbance principle. But, on the other hand, when $N_{\rm err}$ is small, there is a better chance to choose a neuron whose $|X_i|$ is not minimal. This is the minimal stochasticity needed to avoid cycles. Note that initially $N_{\rm err}$ is large, and it decreases during learning. Hence our process realizes a specific *heating schedule*, as the effective temperature $T = 1/\beta$ *increases* in the course of learning.

Once neuron i has been chosen according to (2.5), the weights w_{ij} are changed. As stated above, the change has to be large enough to flip the internal representation for the currently presented pattern. This can be done using an Abbott–Kepler linear rule [16]:

$$\Delta w_{ij} = \frac{-1}{N+1} S_i \, s_j \cdot \begin{cases} 2|X_i|, & |X_i| > k \\ |X_i| + k, & |X_i| \le k \end{cases} \tag{2.6}$$

where k is another arbitrary (small) parameter. Finally, we normalize $(\vec{w}_i)$ according to (2.3), for the reasons described above.

This stage is completed when a solution is found. If no solution has been found after I_{12} sweeps of the training set, we abort this stage and return to learn23.

This is a fairly complete account of our procedure. There are few details that need to be added.

2.3 Choice of parameters

This algorithm has five arbitrary parameters:

The *impatience parameters* I_{12} (I_{23}) are introduced to guarantee that the learn12 (learn23) stage is aborted if no solution is found. This is necessary since it is not clear that a solution exists for the weights, given the internal representation used. The parameters have to be large enough to allow learn12 (learn23) to find a solution (if one exists) with sufficiently high probability. On the other hand, too large of values are wasteful, since they force the algorithm to execute a long search even when no solution exists. Therefore the best values of the impatience parameters can be determined by optimizing the performance of the network; our experience shows, however, that once a reasonable range of values is found, performance is fairly insensitive to the precise choice.

The step size parameter η from (2.4), $0 < \eta < 2$ was always chosen to be 0.234. The algorithm is quite insensitive to the precise choice of η, as long as it is not very large. Choosing very small η means simply that we must increase I_{23}.

The stochasticity parameter β in (2.5), which was used in all the experiments reported here, was chosen to be N_{err} as explained in 2.2. Although this "heating schedule" proved to be the most efficient in most of these experiments, other choices for β (e.g., constant β) were also tested and functioned quite well.

The Abbott–Kepler parameter k from (2.6) was usually chosen as $k = 0.2$, and in some experiments was set to $k = 0$. The updating scheme (2.6) itself is not an essential ingredient of the algorithm. It can be replaced by other updating rules without a significant change in the performance. The important thing is to ensure that the state of the chosen hidden unit will be changed. For this purpose one can use for example rule (2.4), with $\eta = (1 + \epsilon)|X|$.

3. Performance of the CHIR2 algorithm

The task of learning in the type of networks we have been discussing is to produce couplings and thresholds that yield the desired input–output relations. In our algorithm, as in many others, there are several parameters that affect the performance. For a specific learning task and a set of parameters (denoted by A), complete characterization of the learning algorithm's performance is given by $P(t, A)$, the probability that the algorithm finds a solution in less than t "time" steps. One can estimate this function by plotting the distribution histogram of the "time" needed to reach a solution (see figure 2). This can be obtained by performing the learning process many times, each run starting with different initial weights. For any practical application of a learning algorithm, a time limit must be externally specified. Thus, a quantity of interest is the probability of an algorithm to converge within the given time limit. Of course when the success rate is 100%, the calculation of average learning time is sufficient.

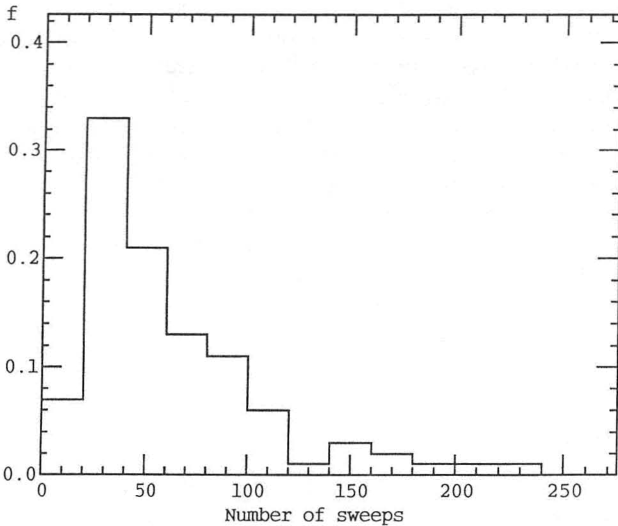

Figure 2: *Parity*: Histogram of the number of training sweeps for 7-14-1 network. The fraction of cases solved within each time interval f versus the number of sweeps of the training set. We present results for $I_{23} = 7$, $I_{12} = 60$, $\eta = 0.234$, $k = 0.2$.

The question now arises as how to measure time in our (and similar) algorithms. Since learning usually takes place by presenting the network with patterns to be learned, a possible definition of "time" is just the number of times the training set has been presented. In our algorithm, there are $I_{12} + I_{23}$ such pattern presentation sweeps in each training cycle. One should remember, however, that various algorithms perform different computations during each presentation, and therefore this characterization is not the best possible. However, it does eliminate the need to introduce ad-hoc measures that may bias the result in various ways.

Since, however, learning algorithms sometimes do not converge to a solution within the specified time limit, the question arises how to take this fact into account in evaluating performance. Clearly, calculating averages (and in general higher moments) that take into account only the cases where the algorithm succeeded in finding a solution is not satisfactory. Another possibility is to calculate the inverse average rate τ, defined in [17] as

$$\tau = \frac{1}{\langle r \rangle} \tag{3.1}$$

where

$$r_i = \begin{cases} 1/t_i & \text{if run } i \text{ is successful} \\ 0 & \text{otherwise} \end{cases} \tag{3.2}$$

In (3.2), t_i is the time needed to solve the problem, i.e., the total number of pattern presentations, as discussed above. Such a measure will be dominated by the small, lucky runs, even when they are rare; the penalty for long unfruitful search is small. In what follows we call "successful runs" only cases in which the learning algorithm found a weight vector that gives a 100% performance on the training set and within a given time limit, $t_{\max}$ (the maximal number of training sweeps). It seems to us that a better characterization of the success of the algorithm is the *median* time taken to solve the problem. The median measures the time needed for a success rate of 50%. However, a success rate of at least 50% is required. In all cases below we really had success rates of more than 50%. The success rate is simply defined as the fraction of successful runs.

Turning now to describe our results, we present our findings for three problems.

3.1 Contiguity

This extensively studied problem [1,8,18] is suitable for a network that makes binary decisions; a string of N digits is presented to the input layer, and the system has to distinguish between inputs according to the number of clumps (i.e., contiguous blocks) of +1's. This problem has a simple "human" or geometric solution, based on edge detection. When the network receives "hints" in the form of spatially limited receptive fields of the hidden units, learning time decreases significantly [8,18]. Here we report results obtained for the harder problem of fully connected networks using an exhaustive set of inputs as the training set. We wanted to compare performance of CHIR2, measured as explained above, with backpropagation and with CHIR. In particular, we were interested in the manner that learning time scales with the problem size.

We trained the network to solve the "2 versus 3" clumps predicate. For this problem we used all possible inputs that have 2 or 3 clumps as our training set. Keeping N fixed, we varied H and plotted t_m, the median number of training passes needed to learn, as a function of H in figure 3. We used 100 cases for each data point at $5 \leq N \leq 7$ and 50 cases for $N \geq 8$. For comparison we present results obtained by CHIR along with results reported by Denker et al. [18], who studied the same problem using an efficient cost function for backpropagation. First we note that our algorithm learns 20–40 times faster than backpropagation and about 10 times faster than CHIR. More important is the dependence of t_m on H. Our algorithm exhibits decrease of t_m with increasing H; adding (possibly unnecessary) hidden units *does not hinder learning*. Backpropagation exhibits increasing t_m with hidden layer size.

To further investigate the size dependence we also studied the 2-versus-3 predicate for networks with $N = H$ units, in the range $5 \leq N \leq 10$. We always found a solution, that is, the success rate was always 100%. Results for the median number of passes needed to solve are given in figure 4.

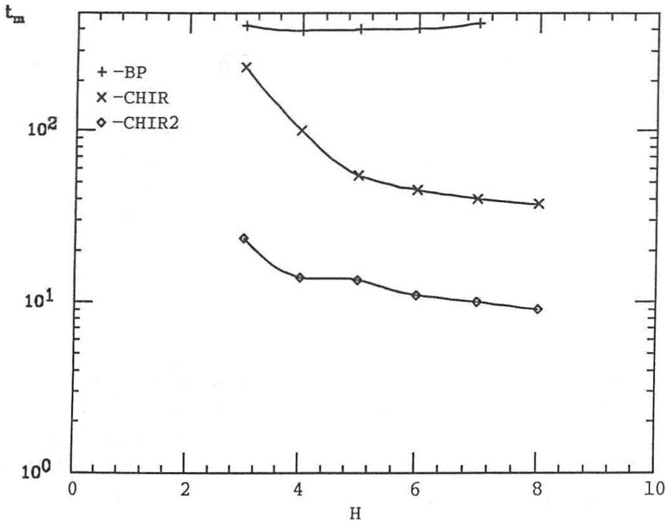

Figure 3: *2 versus 3 clumps problem*: Median number of sweeps t_m, needed to train a network with $N = 6$ input units, over an exhaustive training set, plotted against the number of hidden units H. Results for backpropagation (+) [18], old CHIR algorithm (×) [1], and this work (◇) are shown. The parameters used were $I_{23} = 1$, $I_{12} = 15$, $\eta = 0.234$, $k = 0$ (see section 2.3).

3.2 Symmetry

The second problem we investigated was *symmetry* [19]. Here the output should be 1 if the input pattern is symmetric around its center and -1 otherwise. The minimal number of hidden layer units needed to solve this problem is $H = 2$.

We present in figure 5 the median number of pattern presentations needed to solve the problem as a function of N, the number of input units. The number of hidden units was fixed at $H = 2$. The system was trained over the complete set of 2^N inputs patterns, and the results were averaged over 100 cases. It is interesting that for CHIR2, at large values of N, the median *learning time becomes independent of* N (and equals 11 ± 1), whereas both for an old version of the CHIR algorithm and for backpropagation *learning time increases with* N *very fast.* In [19], Rumelhart et al. report that backpropagation found a solution after 1208 learning steps, and in [6], after 1425 steps. Even if these are typical results (which is unlikely), our algorithm is 100 times faster than backpropagation for $N = 6$. Of course, this ratio increases with N. It is also interesting that the success rate, shown in table 1, *increases* with N for $N > 2$. The reason is that at large N most $(1 - 2^{-N/2})$

N	2	4	6	8	10
Success rate	1	0.85	0.98	1	1

Table 1: Success rate for the symmetry problem with $N:2:1$ architecture ($t_{\max}=50$).

desired answers are -1. In such a case the internal representations that map onto -1 as output contain 3 (of the possible 4) points, and hence the output must represent an AND-like function, which is found with a high success rate.

3.3 Parity

In the *parity* problem one requires $S=1$ for even number bits in the input, and $S=-1$ otherwise. This problem is computationally harder than the previous two, since the output is sensitive to a change in the state of any single input unit. In order to compare performance of our algorithm to that of backpropagation (BP), we studied the parity problem, using networks with an architecture of $N:2N:1$, as chosen by Tesauro and Janssen [17].

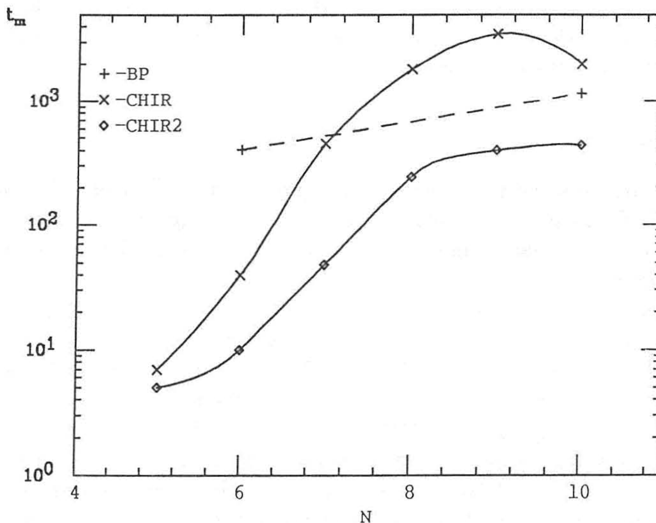

Figure 4: *2 versus 3 clumps problem*: Median number of sweeps t_m, needed to train a network with N input and $H=N$ hidden units, over an exhaustive training set, plotted against the number of input units N. Results for old CHIR algorithm ($\times$) [1] and this work ($\diamond$) are shown. We present results for $\eta=0.234$, $k=0.2$.

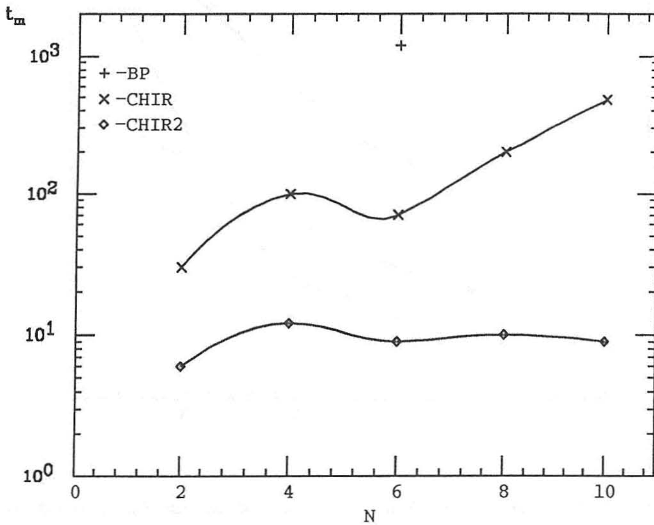

Figure 5: *Symmetry problem*: Median number of sweeps t_m, needed to train a network with N input and $H = 2$ hidden units, over an exhaustive training set, plotted against the number of input units N. Results for backpropagation (+) [19], old CHIR algorithm (×) [1], and this work (◇) are shown.

N	Median	Average	Inv. Rate τ	$t_{\max}$	(I_{23}, I_{12})	# of experiments
2	6	6	6	15	(2,5)	100
3	7	8	8	20	(3,10)	100
4	10	13	11	40	(4,20)	100
5	16	20	15	120	(5,40)	100
6	31	42	27	180	(6,50)	100
7	53	63	42	250	(7,60)	100
8	130	130	80	400	(8,100)	50
9	210	240	150	500	(9,120)	35

Table 2: Parity with $N : 2N : 1$ architecture. The success rate was 1 for all experiments. Parameters: $k = 0.2$, $\eta = 0.234$.

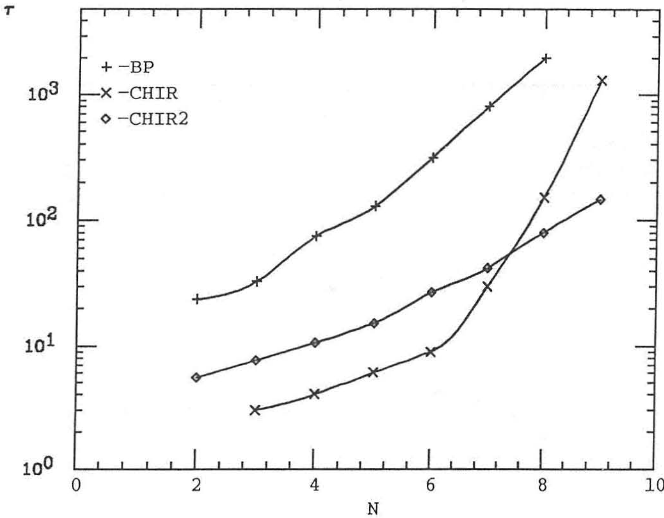

Figure 6: *Parity problem*: Median number of sweeps t_m, needed to train a network with N input and $H = 2N$ hidden units, over an exhaustive training set, plotted against the number of input units N. Results for backpropagation (+) [17], old CHIR algorithm (×) [1], and this work (◇) are shown. The point $N = 8$ uses 50 cases; others use 100 cases. $\eta = .234$, $k = 0.2$. See table 2 for more information.

For this problem, at large N our algorithm is 40 times faster than BP. The results are presented in figure 6 and in table 2. For the sake of comparison with BP, we also present the performance criterion used by Tesauro and Janssen [17]: the inverse average rate τ, defined in equations (3.1,3.2). For all choices of parameters (I_{12}, I_{23}) that are mentioned in the table, our success rate was 100%. Namely, the algorithm did not fail even once to find a solution in less than the maximal number of training sweeps $t_{\max}$, as specified in the table. Note that BP does get caught in local minima, but the percentage of such occurrences is not reported. In addition to inverse rate, we give also the average and the median number of presentations needed for learning. When compared with a new and improved version of BP [20], which is tested on the same problem with $N = 2, 3, 4$, CHIR2 is more than 4 times faster.

When $H = N$ (instead of $2N$), the problem becomes much harder. In fact, N is the minimal number of hidden units needed to solve the problem. Performance of the algorithm for this architecture is given in table 3. The success rate here is lower, as shown in the table.

N	Median	Inv. Rate τ	Success rate	t_{max}	(I_{23}, I_{12})	# of experiments
3	25	18	0.95	500	(2,10)	100
4	67	45	0.99	1000	(3,20)	100
5	374	90	1.00	3000	(4,40)	100
6	1500	330	0.88	6000	(5,60)	50

Table 3: Parity with $N : N : 1$ architecture. Parameters: $k = 0.2$, $\eta = 0.234$.

4. Multiple-output CHIR2

In the previous sections we described the basic version of CHIR2. That version is tailored for a single-output network. In this section we present a way to generalize the algorithm, so that it can train networks with many output units and more than one layer.

4.1 Description of the multiple-output algorithm

It is clear that when generalizing the single-output CHIR into a multiple-output algorithm, we must change the way we choose the hidden unit to be flipped, in the learn12 procedure. The learn23 procedure works as before, by applying the PLR (or any single layer learning rule) for each of the output units independently.

Here, as in the single-output version, the learn12 procedure presents each pattern to the input, and the output of the network is evaluated. If the output is correct, no changes are made. If, however, there is an error, i.e., at least one bit is wrong, one of the hidden units is selected, and its incoming weights are updated in such a way that its state flips its sign.

With one output unit, it is always possible to identify a hidden unit whose contribution to the output field is "wrong." When we have more than one output unit, however, it might happen that an error in one output unit cannot be corrected without introducing an error in another unit. Therefore we take a simple "flip and check" approach (which is similar to the way by which the old CHIR was generalized for multiple-output architectures [2]).

For each hidden unit i, we check the effect of flipping its state ($S_i^\nu \to -S_i^\nu$) on the total output error, i.e., the number of wrong bits, for this pattern, ν. The change in the error is denoted by $\Delta\text{error}(i, \nu)$ (which is negative when the error is reduced). This information about the influence of each hidden unit on the output is now used by the selection procedure in the following manner.

For each hidden unit i evaluate an "energy change," ΔE_i^ν, that includes two terms

$$\Delta E_i^\nu = |X_i^\nu| + \mu\Delta\text{error}(i, \nu) \tag{4.1}$$

The first term is the absolute input field induced on unit i, when pattern ν is presented, and the second is the change in the output error (for the current

pattern ν) if this unit is flipped. We then choose *one* of the hidden units with a probability proportional to $\exp(-\beta\Delta E_i^\nu)$ and change its incoming weights according to equation (2.6). In principle, the parameter μ, which determines the relative strength of the error term in this decision, is a new parameter to be optimized. In practice, however, it was just taken to be unity, and β is the same as before (see section 2.3). All experiments reported here were done with this combination, which was found to be successful (see, however, a few changes in the binary weights version that is described in the next section).

After one of the hidden units is chosen and the weights incident on it from the input are updated and normalized, the learn12 procedure goes on to the next pattern. Since the single-output version also checks all the hidden units for candidates to be flipped, this modification of the algorithm does not increase significantly the amount of computations. Nevertheless, it allows it to handle multiple-output networks.

The single- and multiple-output versions of CHIR2 decide on the hidden unit to be selected for a learning step in slightly different ways, which we would like to emphasize and explain. The value taken by X_i, the field incoming to hidden unit i, affects the selection decision in a similar fashion for the two versions. The difference lies in the manner in which the effect of flipping the state ($S_i \to -S_i$) on the output affects the selection decision. In the single-output version unit i was accepted as a candidate for flipping if the condition $W_i S_i S < 0$ was satisfied; that is, if the flip *pulled* the output field in the right direction. The multiple-output version, on the other hand, takes into account only the actual effect of the flip on the *state* of the output units. For example, if flipping does pull output unit j in the right direction, but W_{ij} is too small and hence the flip does not actually change the state S_j^{out} of output unit j, hidden unit i is treated the same way as if flipping S_i pulled the output in the wrong direction. Therefore the multiple-output version does not probe the *weights* of the output units, but rather only their output state, and the only information that is fed back to the network is the single scalar ΔE. This feature makes the multiple-output version more plausible biologically (like the simpler reinforcement algorithms, see e.g. [21]).

CHIR2 can be further generalized for multilayered feedforward networks by applying the last learn12 procedure to each of the hidden layers, one by one, proceeding from the hidden layer that is closest to the output toward the first hidden layer (which is connected to the input).

4.2 Testing the algorithm

The multiple-output version of CHIR2 was tested on two learning tasks that were studied before with the old CHIR algorithm [2]: (1) the combined parity and symmetry problem and (2) the "random associations" task.

In the combined parity and symmetry problem the network has two output units, both connected to all hidden units. The first output unit performs the parity predicate on the input, and the second performs the symmetry predicate. The network architecture was $N : 2N : 2$ with an exhaustive

N	Median	Average	Inv. Rate τ	(I_{23}, I_{12})	t_{max}	# of experiments
4	20	29	18	(8,16)	240	100
5	105	120	83	(12,24)	360	100
6	150	210	93	(20,40)	1200	100
7	335	470	290	(30,60)	2700	100
8	5800	8600	970	(40,80)	48000	33

Table 4: Combined parity and symmetry with $N : 2N : 2$ architecture.

N	Median	Average	Inv. Rate τ	(I_{23}, I_{12})	t_{max}	# of experiments
4	60	110	41	(6,12)	360	200
8	120	140	110	(12,24)	720	100
16	220	240	210	(12,24)	720	100
32	650	710	620	(30,50)	4000	37

Table 5: Random problem with $N : N : N$ architecture and $P = 2N$ patterns.

training set, and the results for $4 \leq N \leq 8$ are given in table 4. Our choices of the impatience parameters I_{12}, I_{23}, and n, the number of independent runs for each N, are also given there. The parameters k and η were 0.1 and 0.2, respectively, in all the experiments reported here. With these parameters, success rate was 1.00 for every N in the table.

We consider these as good results, much faster (about a factor 20) compared to the results obtained earlier by CHIR [2].

As a second test for the new version we used the "random problem" or "random associations," in which one chooses P random patterns as input and the network is required to learn P random patterns as the desired output. In our test we used an architecture of $N : N : N$, and the number of patterns was $P = 2N$. For each run, the components of the input and output patterns were chosen randomly and independently to be $+1$ or -1 with equal probability. The results, with the typical parameters, for $N = 4$, 8, 16, 32, are given in table 5. Again, it is evident that in addition to the large memory saving, CHIR2 also yields a significant improvement in learning time. Comparison between the old CHIR and CHIR2 on this test problem is plotted in figure 7.

5. CHIR2 for binary weights

In this section we describe how CHIR2 can be used in order to train feed-forward networks with binary weights. According to this strong constraint, all the weights in the system (including the thresholds) can be either $+1$ or -1. The way to do this within the CHIR framework is simple [12]: instead of applying the PLR (or any other single layer, real weights algorithm) for the updating of the weights, we can use a binary perceptron learning rule.

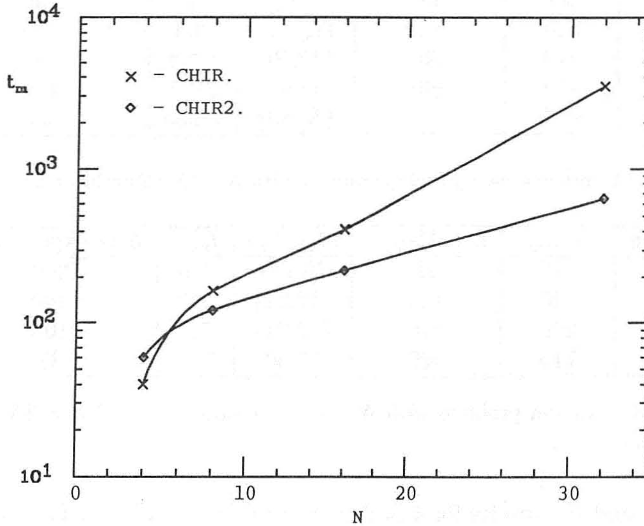

Figure 7: *Random problem*: Median number of sweeps t_m needed to train a network with N input and N hidden units, plotted against the number of input units N. Results for the old CHIR algorithm ($\times$) [2] and this work ($\diamond$) are shown.

Several ways to solve the learning problem in the binary weight perceptron were suggested recently [22,23]. The one that we used is a modified version of the directed drift algorithm introduced by Venkatesh [22,12]. Like the standard PLR, the directed drift algorithm works on-line, namely, the patterns are presented one by one, the state of a unit i is calculated according to (2.1,2.2), and whenever an error occurs the incoming weights are updated. When there is an error it means that

$$\xi_i^\nu h_i^\nu < 0$$

Namely, the field $h_i^\nu = \sum_j W_{ij}\xi_j^\nu$ (induced by the current pattern ξ_j^ν) is "wrong." If so, there must be some weights that pull it to the wrong direction. These are the weights for which

$$\xi_i^\nu W_{ij}\xi_j^\nu < 0$$

Here ξ_i^ν is the desired output of unit i for pattern ν. The learning process consists of simply flipping (i.e., $W_{ij} \rightarrow -W_{ij}$) at random f of these weights.

The number of weights to be changed in each learning step, f, can be a pre-fixed parameter of the algorithm, or, as suggested by Venkatesh, can be decreased gradually during the learning process in a way similar to a cooling schedule (as in simulated annealing). We take $f = |X|/2 + 1$, making sure,

like in relaxation algorithms, that just enough weights are flipped in order to obtain the desired target for the current pattern. Several tests have shown that this modification makes the directed drift algorithm significantly faster. As was done with the modification of CHIR for binary weights, this simple and local rule is "plugged" into the learn12 and learn23 procedures instead of (2.4) and (2.6), and the initial weights are chosen at random to be +1 or −1. We use the multiple-output version of the algorithm, as described in the previous section, with the following changes:

(a) We do not normalize the binary weights; instead, we divide the input fields X_i in equation (4.1) by $N+1$. Note that this scaling with N differs from the one used in the continuous weights version (where we normalize the weight vector). Nevertheless we use the same schedule for β as before.

(b) We do change the μ parameter in equation (4.1). It was found that larger μ values are needed. The performance of the algorithm is not sensitive to the exact value, as long as it is large enough. In the experiments that are reported here we used $\mu = 3.0$. Note, however, that the two learning parameters η and k, are not needed in this version.

We tested the binary version of CHIR2 on two problems. The first is the parity problem with the $N : N : 1$ architecture. This architecture is known to be the smallest layered network that can solve the parity problem with *continuous weights.* Moreover, in the well known continuous weights "human" solution, the hidden units perform the task of counters and their weights scale linearly with N. Therefore it was interesting to realize that this problem can be solved by a network with the same architecture but only ± 1 weights. A typical solution for $N = 6$ is given in table 6. It can be easily generalized to any (even) N. Because of the network and task symmetries, many other solutions can be generated from this one by various transformations (e.g., permutations of the hidden units or inverting all the weights that are incident on and leaving any hidden unit). Yet more solutions, apart from this class of equivalent ones, exist. Note also that different binary solutions may represent disconnected solution regions in the space of continuous weights. One other comment about binary weights perceptrons is that when the number of inputs (including the bias) of such a unit is even, the input field can be zero. In such cases we set the output of this unit to −1 (which is equivalent to a small offset in the bias).

Results for learning times, obtained for $4 \leq N \leq 7$ with typical parameters, are given in table 7. Success rate was 1.00 for all cases. It is also interesting to find that (at least for these small networks) solution times are shorter than those of the continuous weights algorithm. A different algorithm for feedforward networks with binary weights, which is also based on the old CHIR, together with an error minimization approach, was recently presented by Saad and Marom [25]. It was tested on this problem with $N = 5$, and it seems that it is much slower than CHIR2.

i	w_{i0}	w_{i1}	w_{i2}	w_{i3}	w_{i4}	w_{i5}	w_{i6}
1	+	+	+	+	−	−	−
2	+	−	+	+	+	−	−
3	+	−	−	+	+	+	−
4	+	−	−	−	+	+	+
5	+	+	−	−	−	+	+
6	+	+	+	−	−	−	+
W	−	+	+	+	+	+	+

Table 6: A solution for the parity problem with binary weights and 6 : 6 : 1 architecture.

N	Median	Average	Inv. Rate τ	(I_{23}, I_{12})	$t_{\max}$
4	19	26	19	(8,16)	480
5	23	60	29	(25,50)	750
6	215	370	130	(25,50)	3750
7	225	330	130	(25,50)	7500

Table 7: The parity problem with binary weights and $N : N : 1$ architecture.

The second test problem is the "random teacher" task [24,12]. In this problem a "teacher network" is created by choosing a random set of $+1/-1$ weights for the given architecture. The training set is then created by presenting P input patterns to the network and recording the resulting output as the desired output patterns. In what follows we took $P = 2^N$ (exhaustive learning) and an $N : N : 1$ architecture. For each network size N we generated an ensemble of 50 independent runs, with different random teachers and starting with a different random choice of initial weights.

The results, with the typical parameters, for N = 3, 4, 5, 6, are given in table 8. The binary weights version of the old CHIR was also tested on this task [12]. The improvement in the learning times of CHIR2 on this problem is again about a factor of 10.

6. Discussion

The recently introduced CHIR learning algorithm works by combining perceptron learning with a search in the space of internal representations. The

N	Median	Inv. Rate τ	Success rate	(I_{23}, I_{12})	$t_{\max}$
3	17	4	1	(15,30)	450
4	25	13	1	(20,42)	620
5	38	21	1	(25,50)	15000
6	2000	220	0.88	(30,60)	81000

Table 8: The random teacher problem with binary weights and $N : N : 1$ architecture.

algorithm was demonstrated to work as well or better than backpropagation for a variety of simple "toy" problems, defined for networks with a single hidden layer and one output unit. Subsequently, CHIR was extended to handle multilayer and multiple-output networks, as well as networks with binary weights.

An immediately obvious shortcoming of CHIR was the necessity to store at all stages a complete *table of internal representations*, i.e., the state of every hidden unit obtained in response to all training patterns. While this requirement poses no real difficulty, neither for software nor hardware applications, it is aesthetically unappealing and biologically implausible.

In this manuscript we presented CHIR2, a modified version of the CHIR algorithm, that works without storing internal representations. The essence of our modification is a successful incorporation of the ideas on which another learning algorithm, MRII, is based into the general CHIR scheme and philosophy. That is, instead of storing internal representations directly, we follow presentation of a training pattern that (drew wrong response) by a learning step which *ensures* that a modified internal representation is embedded in the network. The danger with this process is that when a new pattern is learned, the new learning step may cause unlearning of a previously presented training pattern. The probability of this occurring is minimized by choosing the hidden unit to be subjected to learning according to a *minimal disturbance principle*. Our version of this principle minimizes the likelihood of flipping the internal representation associated with a *random* input pattern.

We tested CHIR2 on the same problems as was done for CHIR. To our surprise we found that CHIR2 works significantly better that CHIR. Its learning times, measured in the number of sweeps of the training set needed to achieve perfect learning of the training set, are smaller than backpropagation by factors that range from 10 to 100. More important is the fact that for some problems CHIR2 exhibits learning times *that do not increase with the number of input units*, whereas backpropagation (and CHIR) exhibit exponential learning times.

Next we demonstrated an extension of the basic CHIR2 algorithm to networks with more than one output and more than one hidden layer. We also found a way to modify the basic algorithm so that it can be applied to networks with binary weights, and we tested it on a number of learning tasks. Networks with binary weights are relatively easy for hardware implementation, which makes this modified version particularly interesting.

An appealing feature of the CHIR algorithm is the fact that it does not use any kind of "global control" that manipulates the internal representations (as is used for example in [13]). The mechanism by which the internal representations are changed is local, in the sense that changes are made for each unit and each pattern without conveying information from other patterns (representations). Information from other units, for the same pattern, is used only indirectly, via the probabilistic selection procedure. The only feedback from the "teacher" to the system is a single scalar quantity, namely,

what is the total output error (in contrast to BP, for example, where one informs each and every output unit about its individual error).

Other advantages of our algorithm are the simplicity of the calculations, the need for only integer, or even binary weights and binary units, and high success rate in finding solutions. In addition one should bear in mind the fact that a CHIR training sweep involves many fewer computations than that of backpropagation. It seems that further research will be needed in order to study the practical differences and the relative advantages of the CHIR2 and the MRII algorithms.

References

[1] T. Grossman, R. Meir, and E. Domany, *Complex Systems*, **2** (1988) 555.

[2] T. Grossman, *Complex Systems*, **3** (1989) 407.

[3] F. Rosenblatt, *Psych.Rev.*, **62** (1958) 386; *Principles of Neurodynamics* (Spartan, New York, 1962).

[4] B. Widrow and M.E. Hoff, *WESCON Conv. Record IV* (1960) 96.

[5] M. Minsky and S. Papert, *Perceptrons* (MIT Press, Cambridge, MA, 1969).

[6] D. Rumelhart, G. Hinton, and R. Williams, *Nature*, **323** (1986) 533; D.B. Parker, MIT Technical Report TR-47 (1985); Y. LeCun, *Proc. Cognitiva*, **85** (1985) 599.

[7] D.C. Plaut, S.J. Nowlan, and G. E. Hinton, Technical Report CMU-CS-86-126, Carnegie Mellon University (1986).

[8] S.A. Solla, E. Levin, and M. Fleisher, *Complex Systems*, **2** (1988) 625.

[9] S. Kirkpatrick, C.D. Gelatt, and M.P. Vechi, *Science*, **229** (1984) 4598.

[10] R. Rohwer, in *Advances in Neural Information Processing Systems 2*, D. Touretzky, ed. (Morgan Kaufmann, San Mateo, 1990) p. 558; A. Krogh, G.I. Thorbergsson, and J.A. Hertz, ibid., p. 733.

[11] D. Saad and E. Marom, *Complex Systems* (to be published).

[12] T. Grossman, in *Advances in Neural Information Processing Systems 2*, D. Touretzky ed. (Morgan Kaufmann, San Mateo, 1990) p. 516.

[13] M. Mezard and J.P. Nadal, *J. Phys. A*, **22** (1989) 2191; J.P. Nadal, *Intl. J. Neural Systems*, **1** (1989) 55; P. Rujan and M. Marchand, *Complex Systems*, **3** (1989) 229.

[14] B. Widrow and R. Winter, *Computer*, **21**(3) (1988) 25.

[15] G.J. Mitchison and R.M. Durbin, *Biological Cybernetics*, **60** (1989) 345; see also N. Nillson, *Learning Machines* (McGraw Hill, New York, 1965) p. 97.

[16] L.F. Abbott and T.B. Kepler, *J. Phys. A*, **22** (1989) L711.

[17] G. Tesauro and H. Janssen, *Complex Systems*, **2** (1988) 39.

[18] J. Denker, D. Schwartz, B. Wittner, S. Solla, J.J. Hopfield, R. Howard, and L. Jackel, *Complex Systems*, **1** (1987) 877.

[19] D.E. Rumelhart and J.L. McClelland, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, Vol. 1 (MIT Press, Cambridge, MA, 1986) p. 318.

[20] R. Battiti, *Complex Systems*, **3** (1989) 331.

[21] A.G. Barto, R.S. Sutton, and P.S. Brouwer, *Biol. Cybern.*, **40** (1981) 201.

[22] S. Venkatesh, preprint (1989).

[23] E. Amaldi and S. Nicolis, *J. Phys. France*, **50** (1989) 2333; H. Kohler, S. Deiderich, W. Kinzel, and M. Opper, *Z. Phys. B*, **78** (1990) 333; G.A. Kohring, KFA Julich preprint (1990).

[24] E. Gardner and B. Derrida, *J. Phys. A*, **22** (1989) 1983.

[25] D. Saad and E. Marom, preprint (1990).