Complex Systems **2** (1988) 555–575

Learning by Choice of Internal Representations

Tal Grossman
Ronny Meir
Eytan Domany
Department of Electronics, Weizmann Institute of Science
Rehovot 76100 Israel

Abstract. We introduce a learning algorithm for two-layer neural networks composed of binary linear threshold elements. Whereas existing 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. We perform an efficient search in the space of internal representations. When a correct set of internal representations is arrived at, the weights can be found by the local and biologically plausible Perceptron Learning Rule. No minimization of any cost function is involved. We tested our method on three problems: contiguity, symmetry and parity. Our results compare favorably with those obtained using the backpropagation learning algorithm.

1. Introduction

The past few years have witnessed a surge of interest in neural networks. In particular, the most intriguing aspect, that of learning, has received wide attention [1–3].

Formal Logical Neurons were first introduced by McCulloch and Pitts [4], who have also argued that networks composed of such elements can serve to realize any desired input-output relationship. A particularly simple implementation of such a network is the single-layer perceptron introduced by Rosenblatt [5], who also proposed a learning algorithm, that was guaranteed to converge to a solution (if one exists). A related learning rule was introduced by Widrow and Hoff [1,6]. However, Minsky and Papert [7] have demonstrated that simple single-layer perceptrons have very limited applicability. On the other hand, it is known that multi-layered perceptrons can, given a sufficient number of units, implement any binary input/output relation (predicate). Back-propagation (BP) is a learning algorithm, for multilayered perceptrons, proposed [8,9] and recently investigated by a number of groups [2,10]. It uses continuous variables and a smooth sigmoid single unit input-output response function. An error function, which measures the deviation of actual from desired output, is then minimized by the algorithm. There is no *convergence theorem* for back-propagation: 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* [11] for example, is extremely time consuming.

In this communication we introduce [12] a learning algorithm for two-layer perceptrons, that use binary McCulloch-Pitts neurons in their input, internal and output layers, and a sharp single unit threshold function. Our algorithm learns by *choosing internal representations* (CHIR). It differs from back-propagation technically as well as *conceptually*. Whereas BP views the *weights* as the independent variables of the learning process, CHIR views the *internal representations* as the fundamental entities to be determined. Internal representations are the states taken by the hidden layers of the network, generated in response to presentation of inputs from the training set. Once a correct set of internal representations is arrived at, the weights can be found by existing standard methods, such as the *perceptron learning rule* (PLR) [5]. The CHIR algorithm is completely local, and perhaps slightly more plausible biologically than back-propagation, due to the simpler computational requirements of each unit. We present the algorithm in detail below, and also describe its performance on three prototype problems: symmetry, parity, and contiguity. There is no proof of convergence for our procedure. Numerical experiments on symmetry and contiguity show that given enough time, a solution is always found. For parity, the situation is not as clear.

To state the problem of learning in concrete terms, consider a network of binary linear threshold elements (figure 1). Element i can be in one of two possible states, $S_i = \pm 1$. A particularly simple network architecture is presented in Fig. 1. This is a feed-forward network with N inputs, a single output, and H hidden units. There are 2^N possible input patterns, $(S_1^{in}, S_2^{in}, \ldots, S_N^{in})$. The states of hidden layer units S_i^h and the output unit, S^{out} are determined by the input according to the rules

$$S_i^h = \mathbf{sign}\left(\textstyle\sum_{j=1}^N W_{ij} S_j^{in} + \theta_i\right) \tag{1.1}$$

$$S^{out} = \mathbf{sign}\left(\textstyle\sum_{i=1}^H w_i S_i^h + \theta\right) \tag{1.2}$$

Here W_{ij} are the weights assigned to the connections from input unit j to hidden unit i; w_i to the connection from the latter to the output unit. The parameters θ_i (θ) are biases associated with the hidden layer (output) cells. These may be considered as weights which connect the units to a constant input ($S_0 = 1$); from here on reference to the "weights" implies biases as well.

A typical task of such a network is *classification*; divide input space into two categories A and B; whenever a pattern that belongs to A is presented as input, the network is required to respond by setting $S^{out} = +1$, whereas an input from category B gives rise to $S^{out} = -1$. There are 2^{2^N} possible distinct input-output mappings (predicates). Obviously, an arbitrary set of

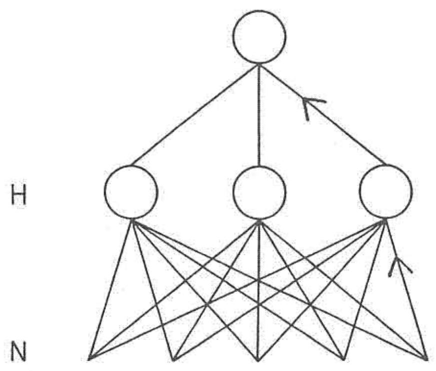

Figure 1: Feedforward network with layered architecture. State of cell i in layer $l+1$ is determined by the states of all cells of layer l. Input patterns are presented to the bottom layer, output is read out from the top. Each cell contains a binary variable $S_i^l = \pm 1$ that defines its state.

weights and biases, when used in the dynamic rule (1), will not produce the mapping required for our A versus B classification task. The basic problem of learning is to find an algorithm, i.e a synaptic modification rule, that produces a set of connections and thresholds which enables the network to perform a preassigned task. If there are no hidden units, and input is directly connected to output, a proper set of weights can be found in a simple, elegant and local fashion by the PLR, described in detail below. Whenever an error occurs, in the course of a PLR training session, weights are modified, in a Hebbian manner [13], toward values that correct the error.

The most impressive aspect of this learning rule is the existence of an associated *convergence theorem*, which states that if a solution to the problem exists, the PLR will find a solution in a finite number of steps [5,7]. However, single-layer perceptrons can solve only a very limited class of problems. The reason for this is that of the 2^{2^N} possible partitions of input space only a small subset (less than $2^{N^2}/N!$) is linearly separable [15]. One of the most widely known examples of a problem that cannot be solved by a single-layer perceptron is *parity* [7]: assign output $+1$ to inputs that have an odd number of $+1$ bits, and -1 otherwise. Parity (and any other) classification task can be solved once a single hidden layer is inserted between input and output. This, however, makes learning difficult; in particular, the PLR cannot be implemented. The reason for this is that in order to determine the corrective increment of a weight W_{ij}, one has to know the correct state of both pre and post synaptic cells, i and j. For multilayer perceptrons only the states of the input "cells" and output are known; no information regarding the correct state of the hidden layer is a priori available. Therefore, when in the course of the training session, the network errs, it is not clear which connection is to blame for the error, and what corrective action is to be taken.

Back-propagation circumvents this "credit-assignment" problem by replacing binary linear threshold elements by units with continuous valued outputs; the discontinuous threshold function (1) is also replaced by a continuous sigmoid function. Thus one can define various cost functions that measure the deviation of actual outputs from those required by the classification task. The cost function depends continuously on the weights, and is chosen so that its lowest value is obtained when each input gives rise to the correct output. Hence the problem of learning is reduced to one of minimizing the cost function over the space of weights. Most such algorithms view the weights as the basic independent variables, whose values determine the internal representations as well as the output obtained in response to each input pattern. Obviously, there is no guarantee that such a procedure will find a global (versus local) minimum, that corresponds to an error-free network. Nevertheless, back-propagation was demonstrated to yield solutions to a variety of problems [2,8]. We note, in passing, that a new version of BP, "back propagation of desired states", which bears some similarity to our algorithm, has recently been introduced [10]. See reference [14] for a related method.

The central point of the CHIR algorithm is the notion that the *internal representations* associated with various inputs should be viewed as the basic independent variable of the learning process. This is a conceptually plausible assumption; in the course of learning, a biological or artificial system should form various maps and representations of the external world. Once such representations are chosen, simple and local Hebbian learning rules, such as the PLR, can be trivially implemented, and the problem of learning becomes one of *choosing proper internal representations.* Failure of the PLR to converge to a solution is used as an indication that the current guess of internal representations needs to be modified.

The remainder of the paper is organized as follows. In section 2 we give a detailed description of the CHIR algorithm. Section 3 contains a presentation of results obtained with the algorithm for a variety of problems, while section 4 summarizes our findings, and discusses possible extensions.

2. The algorithm

Since one of the basic ingredients of our method is the Perceptron Learning Rule, we briefly review the version actually implemented in the CHIR algorithm. The learning process takes place in the course of a training session. Consider $j = 1, \dots N$ source units, which are directly connected to a single target unit i. When the source units are set in any one of $\mu = 1, \dots M$ patterns, i.e. $S_j = \xi_j^\mu$, we require that the target unit (determined using (1)) takes preassigned values ξ_i^μ. A set of weights, W_{ij}^*, θ_i^*, for which this input-output relationship is satisfied, constitutes a solution of the problem. Starting from any arbitrary initial guess for the weights, an input ν is presented, resulting in the output taking some value S_i^ν. Now modify every weight according to the rule

$$W_{ij} \quad \rightarrow W_{ij} + \Delta W_{ij} \tag{2.1}$$
$$\Delta W_{ij} \quad = \eta(1 - S_i^\nu \xi_i^\nu)\xi_i^\nu \xi_j^\nu \tag{2.2}$$

where $\eta > 0$ is the step size parameter. The bias θ gets modified by the same rule, with $\xi_j^\nu = 1$. Now another input pattern is presented, and so on, until all inputs draw the correct output. Note that the PLR modifies weights only when presentation of input ν produces an erroneous output. When that happens, each weight is changed, in a Hebbian fashion, towards values that correct the error.

As explained above, without knowing the internal representations, e.g the states taken by the hidden layer when patterns from the training set are presented, the PLR is not applicable. On the other hand, if the internal representations *are* known, the weights can be found by the PLR. This way the problem of learning becomes one of choosing proper internal representations, rather than of minimizing a cost function by varying the values of weights. To demonstrate the difference between these approaches consider again the classification problem mentioned above; the system is required to produce preassigned output values, $S^{out,\mu} = \xi^{out,\mu}$, in response to $\mu = 1, \ldots, M$ input patterns. If a solution is found, it first maps each input onto an internal representation generated on the hidden layer, which, in turn, produces the correct output. Now imagine that we are not supplied with the weights that solve the problem; however the correct internal representations are revealed. That is, we are given a *table* with M rows, one for each input. Every row has H bits $\xi_i^{h,\mu}$, for $i = 1, \ldots H$, specifying the state of the hidden layer obtained in response to input pattern μ. One can now view each hidden-layer cell i as the target cell of the PLR, with the N inputs viewed as source. Given sufficient time, the PLR will converge to a set of weights $W_{i,j}$, connecting input unit j to hidden unit i, so that indeed the input-output association that appears in column i of our table will be realized. In a similar fashion, the PLR will yield a set of weights w_i, in a learning process that uses the hidden layer as source and the output unit as target. Thus, in order to solve the problem of learning, all one needs is a search procedure in the space of possible internal representations, for a table that can be used to generate a solution. This sounds, *a priori*, a rather hopeless undertaking; if the training set contains all possible inputs (i.e. $M = 2^N$), there are 2^{H2^N} possible distinct internal representations (i.e. tables). Our algorithm actually searches a much smaller space of tables; the reason for this is explained in section 4. Needless to say, the updating of weights can be done in parallel for the two layers, using the current table of internal representations. In the present algorithm, however, the process is broken up into four distinct stages:

1. **setinrep**: Use existing couplings W_{ij} and θ_i to generate a table of internal representations $\{\xi_i^{h,\nu}\}$ on the hidden layer. This is done simply

by presenting each input pattern from the training set and calculating the state on the hidden layer, using equation (2.1).

2. **learn23**: The hidden layer cells are used as source, and the output as the target unit of the PLR. The current table of internal representations is used as the training set; the PLR tries to find appropriate weights w_i and θ to obtain the desired outputs. If a solution is found, we stop; the problem has been solved. Otherwise we stop after I_{23} learning sweeps, and keep the current weights (for more details on the choice of this parameter, see remark below).

3. **inrep**: Use the current values of w_i and θ to generate a new table of internal representations, which, when used in (2.2), yields the correct outputs. This is done by presenting the table sequentially, row by row, to the hidden layer. If for row ν the wrong output is obtained, the internal representation $\xi^{h,\nu}$ is changed. Having the wrong output means that the "field" produced by the hidden layer on the output unit, $h^{out,\nu} = \sum_j w_j \xi_j^{h,\nu}$ is either too large or too small. We then randomly pick a site j of the hidden layer, and check the effect of flipping the sign (changing the "activity") of $\xi_j^{h,\nu}$ on $h^{out,\nu}$; if it changes in the right direction, we replace the entry $\xi_j^{h,\nu}$ of our table by $-\xi_j^{h,\nu}$. Namely,

$$\text{if } w_j \xi_j^{h,\nu} \xi^{out,\nu} < 0 \text{ then } \xi_j^{h,\nu} \rightarrow -\xi_j^{h,\nu} \tag{2.3}$$

We keep picking sites and changing the internal representation of pattern ν until the correct output is generated. Note that we always generate the correct output this way, provided $\sum_j |w_j| > |\theta^{out}|$. It is easy to design the perceptron learning process, in the *learn23* stage, so that this condition is satisfied. This procedure ends with a modified table which is our next guess of internal representations. During this stage only the table $\xi_j^{h,\nu}$ is changed; all weights and thresholds remain fixed. If weights W_{ij} (connecting input to hidden layer) can now be found, so that this table is indeed the outcome of presenting all inputs in our training set, we have solved the problem.

4. **learn12**: Using the new table obtained from *inrep*, apply the PLR with the first layer serving as source, treating every hidden layer site separately as target. Actually we use a slightly modified version of the PLR: when an input from the training set is presented to the first layer, we first check whether the correct result is produced on the *output* unit of the network. If we get wrong overall output, we use the PLR for every hidden unit i, modifying weights incident on i according to (2), using column i of the table as the desired states of this unit. If, however, presentation of an input ν to the first layer does yield the correct final output, we insert the current state of the hidden layer as the internal representation associated with pattern ν, and no learning steps are taken. We sweep in this manner the training set, modifying weights W_{ij}, (between input and hidden layer), hidden-layer biases

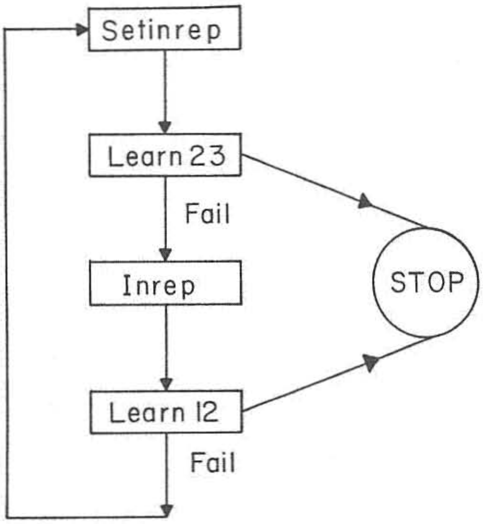

Figure 2: Flow chart for the algorithm described in this paper. Details about the different stages are given in section 2 of the main text.

> θ_i, and, as explained above, internal representations. 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 PLR stage, and keep the present values of weights and thresholds.

The algorithm starts out by setting W_{ij} and θ_i randomly. Then the initial guess for our table is obtained by using *setinrep*. Next, an attempt is made, using *learn23*, to learn w_i and θ; failure to do so in less than I_{23} sweeps of the table sends us to *inrep*. A new table is generated, and the couplings W_{ij}, θ_i are learned using *learn12*. Failure to achieve perfect performance in I_{12} sweeps of the training set results in freezing the weights and restarting the cycle with *setinrep*, and so on. The flow chart is given in figure 2; the program performs a preset number of cycles before "giving up" its attempt to find a solution.

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

(a) *The "impatience" parameters*, I_{12} and I_{23}, which are rather arbitrary, are introduced to guarantee that the PLR 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 current table of internal representations. Thus, if the PLR stage does not converge within the time limit specified, a new table of internal representations is formed. The parameters have to be large enough to allow the PLR to find a solution (if one exists) with sufficiently high probability. On the other hand, too large 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 indicates, however, that once a "reasonable" range of values is found, performance is fairly insensitive to the precise choice.

(b) *Stochasticity*: While using the PLR, we randomly choose which pattern is presented next from the training set. Sequential presentation of patterns may cause the "deterministic" algorithm described above to enter a cycle [16]. It is possible to introduce stochasticity in a different manner, so that sequential presentation of the training set does not generate cycles. This can be done by flipping an entry in our table in a stochastic fashion, e.g. with probability

$$P(\xi_i^{h,\nu} \rightarrow -\xi_i^{h,\nu}) = \exp(-w_i \xi_i^{h,\nu} \xi^{out,\nu}/T)/2\cosh(w_i/T)$$

When the "temperature"-like parameter, T, is set to zero, we recover the version described above. When T is non-zero there is a finite probability to flip an internal unit even if doing so pulls the output in the wrong direction. We do not report here details of the effect of $T > 0$.

(c) *Treating Multiple Outputs*: A simple modification of the *inrep* procedure provides a different way to avoid cycles, as well as a method applicable to deal with multiple output architectures. In the version of *inrep* described above, we keep flipping the internal representations until we find one that yields the correct output, i.e. zero error for the given pattern. Instead, we can allow only for a pre-specified number of attempted flips, and go to the next pattern even if zero error was not achieved. In the modified version we also introduce a slightly different criterion for accepting or rejecting a flip. Having chosen (at random) a hidden unit i, we check the effect of flipping the sign of $\xi_i^{h,\nu}$ on the output error (and *not* on the output field, as described above). If the output error is not increased, the flip is accepted and the table of internal representations is changed accordingly (other variants of this acceptance criterion are also possible).

This version of the algorithm is applicable for multiple-output architectures, and will be described elsewhere. Here we report only investigations that use the previously described "restrictive" version of *inrep*.

(d) *Integer weights*: When using binary units, setting $\eta = \frac{1}{2}$ ensures that $\Delta W_{i,j} = 0, \pm 1$, and one can use integer $W_{i,j}$'s without any loss of generality. Thus, the algorithm can be fully implemented using only integers, which is an important advantage in many applications.

(e) *Optimization*: The algorithm described uses several parameters, which should be optimized to get the best performance. These parameters:

I_{12} and I_{23} - see section (a) above.

Time limit — upper bound to the total number of training sweeps.

T (temperature, defined for the stochastic version of *inrep*)

PLR training parameters — i.e the increment of the weights and thresholds during the PLR stage. In the PLR we used values of $\eta \simeq 0.1$ (see equation (2.3, 2.4)) for the weights, and $\eta \simeq 0.05$ for thresholds, whereas the initial (random) values of all weights were taken from the interval $(-0.5, 0.5)$, and thresholds from $(-0.05, 0.05)$. In the integer weights program, described above, these parameters are not used.

3. Performance of the algorithm

Many workers, in presenting their results concerning learning in neural networks, do not report the success rates of their algorithms. It is thus very difficult to judge the quality of different learning algorithms, since it is usually not clear whether the results given are typical, representative or rare. Thus, it is important to discuss criteria for assessing the success of a learning algorithm.

The task of learning in the type of network we have been discussing is to produce couplings and thresholds which yield the desired input/output relations. In our algorithm, as in many others, there are several parameters which affect the performance. For a specific learning task and a set of parameters (denoted by A), complete characterization of the learning algorithms' 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 3). This can be obtained by performing the learning many times, each with different initial couplings and thresholds. For any practical application of a learning algorithm a time limit must be externally specified. Thus, the 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 the average is sufficient.

The question now arises as to how to measure time in our (and similar) algorithms. Since learning usually takes place by continually presenting the

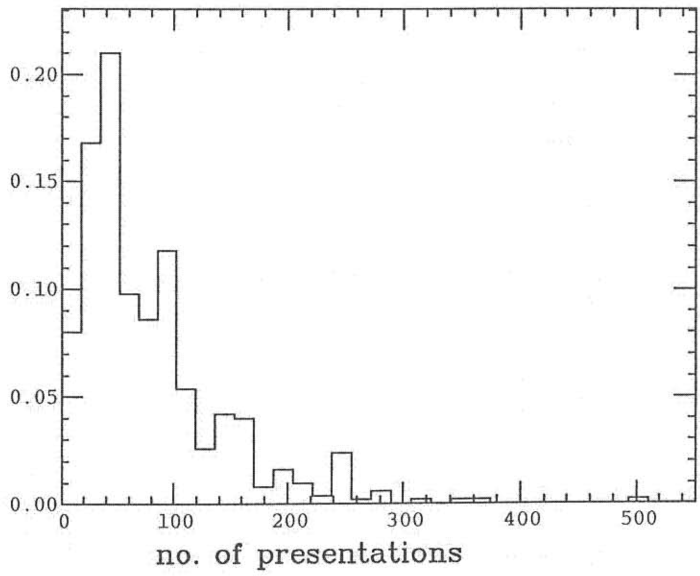

Figure 3: *Contiguity*: Histograms of the number of training sweeps (pattern presentations) needed to solve the "4 or more" predicate, with $N = H = 8$ cells. The horizontal axis is the number of presentations t, while the vertical axis indicates the fraction of cases solved within each time interval. We present results for $I_{12} = 20$ and $I_{23} = 5$.

network with patterns to be learned, a possible definition of "time" is just the number of times the training set has been presented. In the CHIR algorithm, there are $I_{12} + I_{23}$ such pattern presentation sweeps in each cycle. One should keep in mind however, that various algorithms perform different computations during each presentation, and therefore this characterization is not the best possible. Nevertheless, it does eliminate the need to introduce ad-hoc measures that may bias the result in various ways.

Since, however, various algorithms often do not converge to a solution within the time limit specified, the question arises as to 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 as [19]:

$$\tau = \left(\frac{1}{n}\sum_{i=1}^{n} r_i\right)^{-1} \tag{3.1}$$

where

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

In equation (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 short, "lucky" runs, even when they are rare; the penalty for long unfruitful search is small.

It seems to us that a better characterization of the success of the algorithm is t_m, the *median* "time" taken to solve the problem. The median measures the time needed, for a success rate of at least 50%, and can be obtained from simulations, run over many different initial conditions. However, a success rate of at least 50% is required. In all (but two) cases reported below we indeed had success rates of more than 50%; this allowed us to calculate the median running time in each such case. In addition, we have also plotted distribution histograms, as described above. For two cases (see Parity) when it was impractical to achieve a success rate of more than 50%, we reported the time needed for a 10% success rate.

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

3.1 Contiguity

This problem has been studied quite extensively in the recent literature [17]. It 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 give a yes/no answer regarding the number of clumps (i.e. contiguous blocks) of +1's. For example, determine whether the input does or does not have more than n such blocks; this is called the "n-or-more" predicate. These problems have a simple "human" or geometric solution, based on edge detection. Some of the questions raised have to do with the ability of the network to find such a solution, and the constraints that one may have to impose on the learning process to guide it towards it. These questions were raised mainly to address the issue of generalization in a controlled, quantitative fashion. Here we studied the efficiency of the learning process, using an exhaustive set of inputs as the training set. We wanted to compare performance of our algorithm, measured, as explained above, with backpropagation. Also, we were interested in the manner that learning time scales with problem size.

As the first test, we taught a network with $N = H = 8$ cells to solve the "4 or more" problem, using the entire space of 256 possible inputs as our training set. In figure 3 we present a histogram based on 500 trials (i.e. initial choices of weights). The horizontal axis indicates t, the number of training sweeps (presentations of the training set) needed to achieve perfect performance. We have included in this count presentations to both the input and hidden layers. The vertical axis indicates the percentage of cases for which learning was completed in the corresponding number of training sweeps. In the cases summarized in the histogram of figure 3, each learning cycle contained $I_{12} = 20$ presentations of all inputs to the first, and $I_{23} = 5$ presentations to the second (hidden) layer. For this set of parameters a *solution was found in all 500 cases* in less than 100 learning cycles ($t < 2500$ sweeps). The average number of presentations of the training set was $t = 79 \pm 3$. This set of parameters was found to be close to optimal, and was used always (unless otherwise stated). The effect of choosing the impatience parameters was also studied. We found that if I_{12} is increased for fixed I_{23}, t decreases initially, but then levels off. If we keep I_{12} fixed, performance becomes worse as I_{23} increases.

We also trained the network to solve a slightly different problem; 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 plot, in figure 4, the median number of training passes needed to learn, t_m, as a function of H. Again, 500 cases were used for each data point. For comparison we also present results reported by Denker et al. [17], who studied the same problem using an efficient cost function for backpropagation. The results of this comparison are rather interesting. First, we note that CHIR learns much faster than backpropagation. More important is the dependence of t_m on H. Our algorithm exhibits a 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 of our algorithm, we also studied the 2 versus 3 predicate for networks with $N = H$ units, in the range $5 \leq N \leq 10$. Results for the median number of passes needed to solve are given in Fig. 5. No backpropagation data are available for comparison regarding this scaling. We found that t_m increases exponentially with N initially, but levels off and *decreases* for $N = 10$. This decrease is statistically significant; it may be due to the fact that we kept the "problem size" (e.g. 2 versus 3) fixed, while the input size (N) was increased. Typical CPU-times for these networks, for $N = H = 8, 9, 10$ were 1.2, 3, and 10 sec/cycle, respectively, on an IBM 3081. We imposed a cutoff of 200 on the number of cycles allowed; for all sizes studied solutions were found in the majority of cases for less then 200 cycles. Therefore t_m does not depend on the cutoff.

3.2 Symmetry

The second problem we investigated was that of *symmetry* [18]. In this case 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 $H = 2$.

To study the dependence of the learning time on the problem size, we present in figure 6 the median number of pattern presentations needed to solve the problem as a function of the size of the input layer. The number of hidden layer units was fixed at $H = 2$, independent of the input field size N. In the cases presented the system was trained over the complete set of 2^N inputs, and the results were averaged over 500 cases. The parameters used were $I_{12} = 10$ and $I_{23} = 5$. The maximum allowed number of cycles was 200.

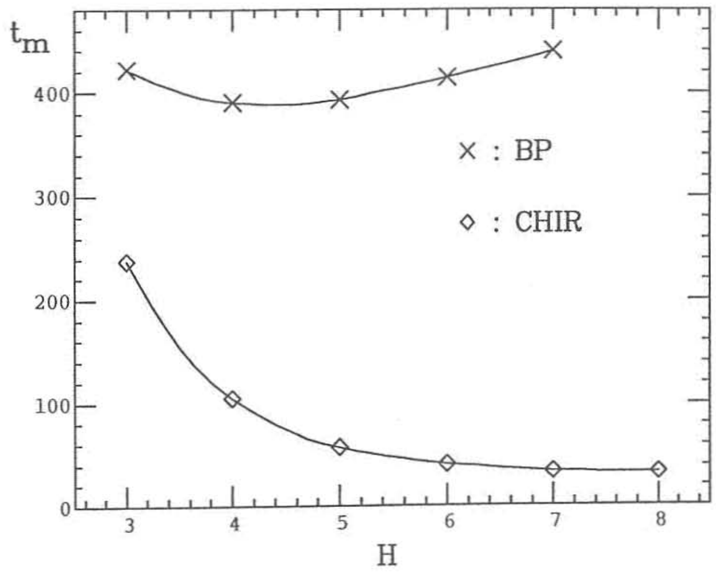

Figure 4: *"2 vs 3" problem*: Median number of sweeps t_m, needed to train a network of $N = 6$ input units, over an exhaustive training set, to solve the " 2 vs 3" clumps predicate, plotted against the number of hidden units H. Results for back-propagation [17] ($\times$) and this work ($\diamond$) are shown. We always find a solution in less than 200 cycles; each point uses 500 cases.

For $N \leq 6$ the network always found a solution, while it had over 90% success for $N = 8, 10$. We note that using the back-propagation algorithm we have *not* been able to find a solution in hundreds of attempts for this architecture, although a solution has been reported in reference [18]. However, we did not make an exhaustive check of parameter space for the BP algorithm.

Next, we investigated the dependence of the learning time on the number of hidden units for the case $N = 8$. Figure 7 depicts the dependence of the number of pattern presentations on H, for the case where the system was taught 240 out of the 256 possible patterns. As we see the learning time increases from $H = 2$ to $H = 3$ and then decreases rapidly for $H = 4$, from where it also decreases, but more slowly. Decrease of learning time for increasing H was also seen in our study of contiguity. We note that addition of many unnecessary hidden units does not hinder the learning ability of our algorithm, but does not seem to help either (for the present task).

Finally, we checked the "rule extraction" ability of the network. By this we mean the following: a fraction f of all possible input patterns constitutes the training set, used to teach the network. Subsequently, the network is

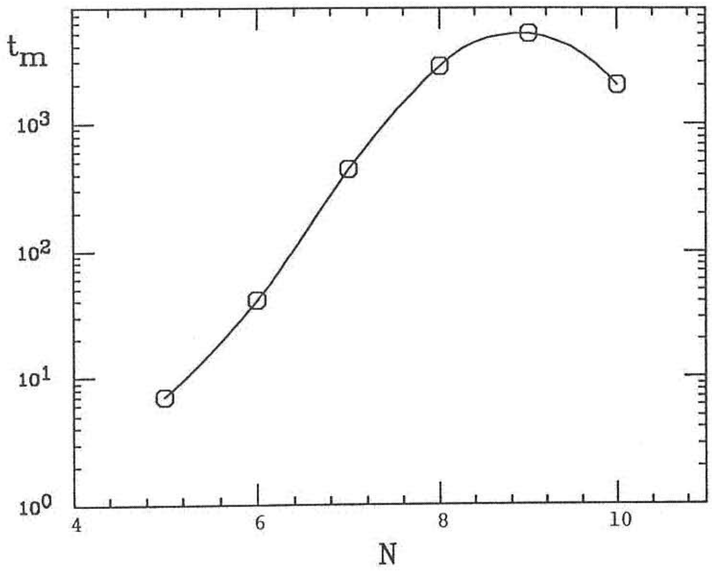

Figure 5: *"2 vs 3" problem*: Median number of sweeps t_m, needed to train the network using an exhaustive training set, for the "2 versus 3" predicate, plotted versus network size ($N = H$).

tested on the patterns it has not been taught. In Fig. 8 we present the rule extraction score S (fraction of correct scores on patterns not in the training set) as a function of f, for a fixed input size $N = 8$, with varying $H = 2,3,4,5,6,8$. As can be seen in the figure, the number of errors in the output is, to a good approximation, independent of H.

4. Parity

In the *Parity* problem one requires $S^{out} = 1$ for an even number of $+1$ bits in the input, and -1 otherwise. This problem is considered computationally harder than the other two, since the output is sensitive to a change in the state of any single input unit. Tesauro and Janssen [19] used BP to teach networks to perform this task, and studied how training time scales with the input size N. In order to compare performance of the CHIR algorithm to that of BP, we studied the Parity problem, using networks with an architecture of $N : 2N : 1$, as chosen by Tesauro and Janssen [19].

We used the integer version of our algorithm, briefly described above. In this version of the CHIR algorithm the weights and thresholds are integers, and the increment size, for both thresholds and weights, is unity. As an initial condition, we chose them to be $+1$ or -1 randomly. In the simulation of this version, all possible input patterns were presented sequentially in a

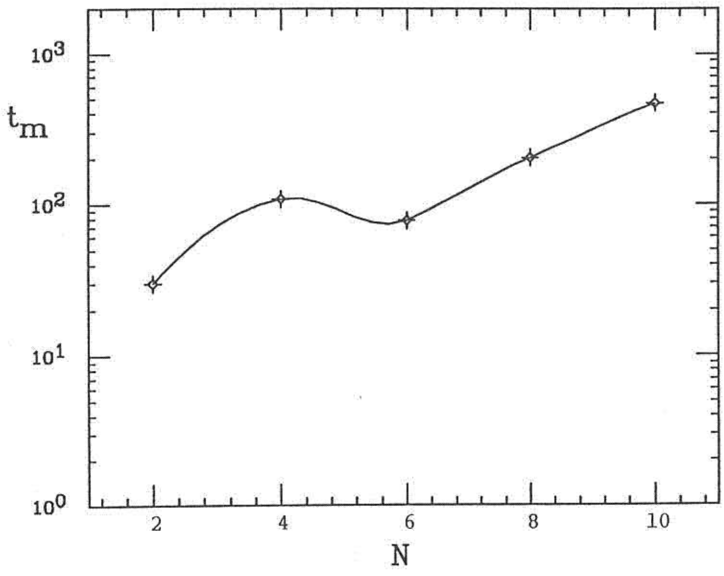

Figure 6: *Symmetry*: Median number of sweeps t_m, needed to train the network using an exhaustive training set, versus number of inputs N, with a fixed number $H = 2$ of hidden units.

fixed order (within the perceptron learning sweeps).

The results are presented in figure 9 and in table 1. For the sake of comparison with BP, we also present the performance criterion used by Tesauro and Janssen [19] — the inverse average rate τ, defined in equation (2.2). For all choices of the 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 cycles t_{max}, specified in the table. Note that BP does get caught in local minima, but the percentage of such occurrences was not reported. In addition to the inverse rate, we give also the average and the median number of presentations needed for learning. It is interesting to note that these numbers decrease when we increase N from 8 to 9.

Using integer weights makes it easy to study their distribution and the number of bits needed for their specification. This is an interesting question, and quite relevant for several applications, e.g. when designing a "hardware implemented" network, like a VLSI chip. For the trained $N : 2N : 1$ systems, we calculated the standard deviation of the weights distribution, as well as the maximal absolute value of weight that occurred in a trained system. These results are presented in table 2. According to these results 8 bits per weight suffice to solve the parity problem for $N \leq 9$. Even though scaling

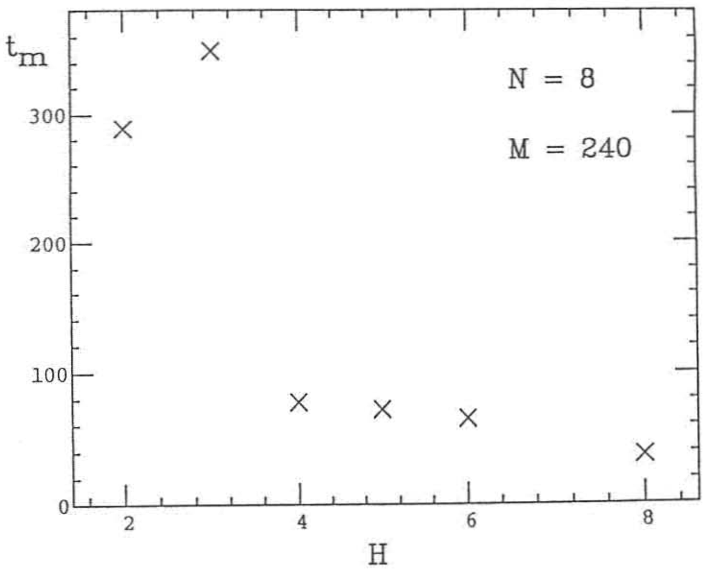

Figure 7: *Symmetry*: Average number of sweeps t_m, needed to train the network, vs the size of the hidden layer H for $N = 8$. We used 240 out of the 256 possible inputs.

of the weight distribution's width with N cannot be determined accurately from this data, it appears to increase as a small power of size.

With $H = N$ (instead of $2N$), the parity problem becomes much harder. In fact, N is the minimal number of hidden units needed to solve this problem. Performance of the integer algorithm for this architecture is given in table 3. The success rate here is smaller, as shown in the table. For $N = 6,7$, the success rate is less than 50% and therefore it was not possible to calculate the median. Instead, we give the number of presentations needed for a success rate of 10%.

5. Discussion

We have presented a learning algorithm for two-layer perceptrons, that searches for internal representations of the training set, and determines the weights by the local, Hebbian perceptron learning rule. Learning by choice of internal representation may turn out to be most useful in situations where the "teacher" has some information about the desired internal representations. For example, when one is aware of a partial set of features that are likely to be relevant for solving the desired classification task. We demonstrated that our algorithm works well on three typical problems. Comparisons with back-

N	n	(I_{12}, I_{23})	t_{max}	Median	Average	τ(Inv. Rate)
3	200	(8,4)	10	3	6	3
4	200	(9,3)(6,6)	20	4	9	4
5	300	(12,4)(9,6)	40	8	22	6
6	200	(12,4)(10,5)	120	19	84	9
7	210	(12,4)(15,5)	240	290	380	30
8	100	(20,10)	900	2900	4000	150
9	80	(20,10)	600	2400	2900	1300

Table 1: Parity with N:2N:1 Architecture: median, average, and inverse rate τ. n is the number of cases run for each N. When two choices of the parameters (I_{12}, I_{23}) are given, they were both simulated with similar results for both. The success rate was 100% in all cases, and the statistical errors were around 10%.

N	σ	max	σ	max
3	1.1	4	1.9	6
4	1.1	5	2.5	9
5	1.4	6	3.8	15
6	2.3	11	5.8	23
7	3.8	17	1.7	39
8	5.1	20	16.8	52
9	6.1	26	21.9	77

Table 2: Weight Distribution in Parity N:2N:1 architecture. σ is the standard deviation of the weight distribution, while "max" denotes the maximal weight obtained. The first pair of columns refer to W_{12}, the second to W_{23}.

N	n	(I_{12}, I_{23})	t_{max}	Success rate	Median*	τ
3	200	(6,6)	30	0.62	60	10
4	400	(9,3)(6,6)	120	0.65	240	34
5	130	(9,6)	240	0.78	750	120
6	200	(18,6)(15,10)	900	0.23	2800*	3900
7	120	(20,10)(15,10)	1200	0.13	4300*	7700

Table 3: Parity with N:N:1 Architecture: success rate, median time, and inverse rate τ.
*For $N = 6, 7$ we give the number of presentations required for a 10% success rate (instead of the median).

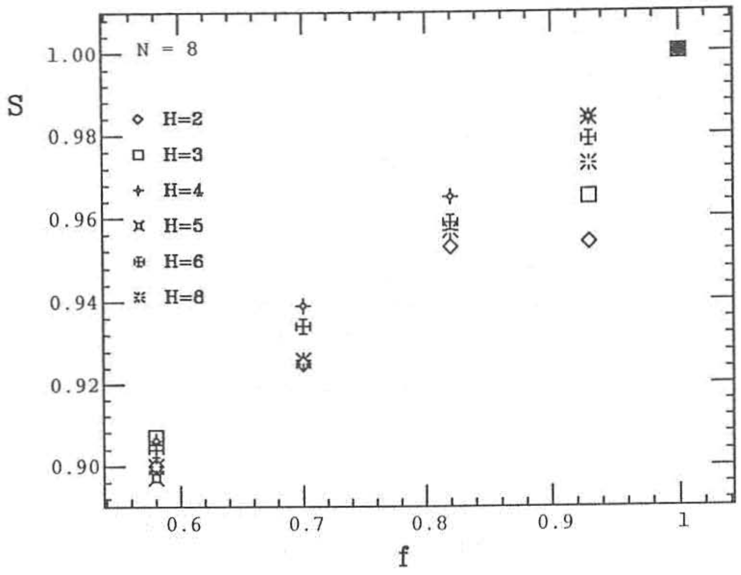

Figure 8: *Symmetry*: Rule extraction score S (fraction of correct scores on patterns not taught) for the symmetry problem, versus f, the fraction of input states (out of 2^N) used in the training set, for fixed $N = 8$ input units and varying H. We used 500 cases for each point.

propagation were also made. It should be noted that one training sweep involves much less computations than that of back-propagation. We presented also results concerning the manner in which training time varies with network size; non-monotonic size dependence was found, with significant variation from problem to problem. For the symmetry problem we also investigated the capacity of the algorithm for "rule extraction".

As mentioned above, the CHIR algorithm searches for a correct internal representation in a space much smaller than that of all 2^{HM} possible tables. The reason for this is the following. We alternate between two sets of tables that are obtained in two distinct ways. All tables generated by the *setinrep* procedure belong to the first set, denoted by T, the set of all possible *Targets*. These are the tables that can be obtained by presenting the training patterns as input, and using the dynamic rule (1) with all possible assignments of weights $W_{i,j}$. The number of tables in T is less than $(2^{N^2}/N!)^H$, which for large M is much smaller than the total number of possible tables. Tables obtained using *inrep* belong to a second set, denoted by S (for *Source*). This set contains all tables that can give rise, using all possible w_i, to the correct output. The tables that solve the learning problem constitute the set $T \bigcap S$

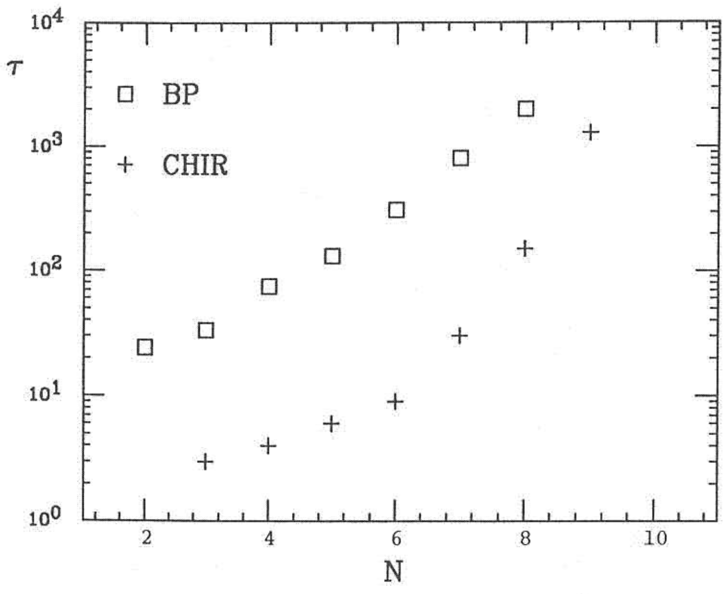

Figure 9: *Parity*: Inverse rate, τ (defined in equation (2.2)), versus problem size, for N : $2N$: 1 architecture. Open squares are the results reported using the BP algorithm [19], while the + signs give our results.

(see figure 10). Our algorithm takes a table $t \in T$, and checks (using *learn23*) whether it does or does not belong also to S. If $t \notin S$, a new table $s \in S$ is generated and we check (using *learn12*) whether it belongs also to T or not. If not, a new table t' is generated, and the procedure is repeated. This way our search is limited to subsets that are much smaller than the set of all possible tables. Note that in the course of learning the weights change, and this change will, in general, give rise to a new guess for our table that differs from the previous one (e.g. $t' \neq t$).

An appealing feature of our algorithm is that it can be implemented in a manner that uses only integer-valued weights and thresholds. Thus, the state of the network at each given time is given by a point in a discrete weight-space. This discreteness makes the analysis of the behavior of the network much easier, since we know the exact number of bits used by the system in constructing its solution, and do not have to worry about round-off errors. From a technological point of view, for hardware implementation it may also be more feasible to work with integer weights. Moreover, biological evidence suggests that nervous systems are not capable of fine-tuning their connections in a smooth way, as is required by the BP and other algorithms. It is thus more realistic to allow the weights to take on only a discrete set of

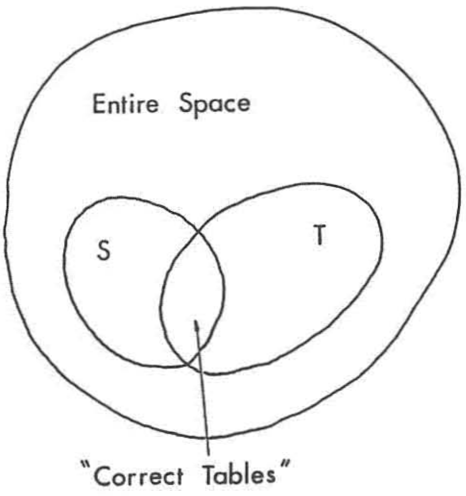

Figure 10: *Search in the space of tables*: Tables that solve the learning problem constitute the set $S \bigcap T$. Both S and T are much smaller than the set of all possible 2^{MH} tables of internal representations. Our algorithm searches for a correct table by alternatively generating tables in S and T; see text for details.

possible values. Of course, one would like this set to be as small as possible, thus rendering the most compact and efficient solution; however, this is a rather stringent condition which is difficult to realize in general.

We are extending this work in various directions. The present method needs, in the learning stage, MH bits of memory: internal representations of all M training patterns are stored. This feature is biologically implausible and may be technologically limiting; we are developing a method that does not require such memory.

The algorithm presented in this paper is tailored for a system comprising a single layer of hidden units and a single output. Generalizing it to networks with multiple outputs is not entirely straightforward, due to the fact that in the *inrep* stage the hidden units may get conflicting instructions from the different output sites (this fact has been realized by Plaut et al. [10] in their related learning algorithm). We did, however, succeed to solve this problem (see point (c) in section 2, and found that a network with multiple outputs did function well on various problems of the same kind as discussed above. This extension will be described elsewhere. It appears that the modification needed to deal with multiple outputs also enables us to solve the learning problem for network architectures with more than one hidden layer.

Other directions of current study include extensions to networks with continuous variables, and to networks with feed-back.

References

[1] T. Kohonen, *Self Organization and associative Memory*, (Springer Verlag, 1984).

[2] D. E. Rumelhart and J. L. McClelland, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, 2 vols. (The MIT Press, Cambridge, MA, 1986).

[3] R. P. Lippmann, *IEEE ASSP Magazine*, **4** (1987) 4.

[4] W. S. McCulloch and W. Pitts, *Bull. Math. Biophys.*, **5** (1943) 115.

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

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

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

[8] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, *Nature*, **323** (1986) 533–536.

[9] D. B. Parker, *MIT technical report*, **TR-47** (1985).

[10] D. C. Plaut, S. J. Nowlan, and G. E. Hinton, Technical Report CMU-CS-86-126.

[11] S. Kirkpatrick, C. D. Gelatt and M. P. Vechi, *Science*, **229** (1983) 4598.

[12] A preliminary version of this work was presented at the Neural Network for Computing meeting, Snowbird 1988.

[13] D. O. Hebb, *The organization of Behavior*, (John Wiley, New York, 1949).

[14] Y. Le Cun, *Proc. Cognitiva*, **85** (1985) 593.

[15] P. M. Lewis and C. L. Coates, *Threshold Logic*, (Wiley, New York, 1967).

[16] Note that in the "inrep" stage we choose randomly the hidden layer site, whose internal representation we attempt to change. However, there may arise situations in which this stochasticity is insufficient (e.g. when there is only one "wrong" site).

[17] J. Denker, D. Schwartz, B. Wittner, S. Solla, J. J. Hopfield, R. Howard, and L. Jackel, *Complex Systems*, **1** (1987) 877–922.

[18] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, in volume 1 of reference [2], page 318.

[19] G. Tesauro and H. Janssen, preprint 1988.