Complex Systems **6** (1992) 213–236

Training Recurrent Neural Networks via Trajectory Modification

D. Saad*
Faculty of Engineering, Tel Aviv University, 69978, Israel

Abstract. Trajectory modification of recurrent neural networks is a training algorithm that modifies both the network representations in each time step and the common weight matrix. The present algorithm is a generalization of the energy minimization formalism for training feed-forward networks via modifications of the internal representations. In a previous paper we showed that the same formalism leads to the back-propagation algorithm for continuous neurons and to a generalization of the CHIR training procedure for binary neurons. The TRAM algorithm adopts a similar approach for training recurrent neural networks with stable endpoints, whereby the network representations in each time step may be modified in parallel to the weight matrix. In carrying out the analysis, consistency with other training algorithms is demonstrated when a continuous-valued system is considered, while the TRAM learning procedure, representing an entirely different concept, is obtained for the discrete case. Computer simulations carried out for the restricted cases of parity and teacher-net problems show rapid convergence of the algorithm.

1. Introduction

Several methods have been applied to the training of recurrent neural networks (RNN) with stable endpoints [4, 5, 6] and to RNN that produce trajectories in time [2, 7, 8, 9]. The two kinds of networks differ in their nature and in the set of tasks to which they are applicable; their training procedures are therefore tackled using different training algorithms. Most of these methods are based upon direct modification of the weight matrix in accordance with the decrement of a cost function related to the problem, which is usually similar to the one used in the back-propagation (BP) algorithm [2]. All of the training methods applied to RNN suffer from a very long training procedure because one should obtain a weight matrix that fits all of the network

*Current address: Department of Physics, University of Edinburgh, J. C. Maxwell Building, Mayfield Road, Edinburgh EH9 3JZ UK. Electronic mail address: `saad@castle.ed.ac.uk` representations in the various time steps, which enforces tough restrictions on the desired weight matrix. Moreover, the training methods applied to RNN with stable endpoints have two additional major restrictions:

- the endpoints should be stable; and
- there is no information concerning the various trajectories prior to the endpoints (for both hidden and output neurons).

The methods currently used to train RNN with stable endpoints [4, 5] concentrate on modifying the stable states—disregarding the trajectories—by performing direct weight matrix modifications. On the other hand, the rest of the RNN training algorithms, aimed at tackling trajectory tasks, focus on a direct iterative weight matrix modification due to the required output vector along the entire trajectory.[1]

Because each weight modification affects the entire trajectory, we use in the trajectory modification (TRAM) algorithm a different approach designed to modify *all* of the system representations in the various time steps, including the stable point, according to a modification rule derived from the proper cost function. The modification of the weight matrix, *common to the entire trajectory*, is performed only after *all* of the system representations along the trajectories have been defined, thus avoiding conflicts in the weight matrix modifications. The modification rules for the trajectories, including the stable states, are derived according to the formalism described in a previous paper [1], that is, a variation of the gradient descent procedure designed to minimize a cost function and limit the required weight modifications. Once the trajectories have been defined the weight matrix is modified according to the perceptron learning rule (PLR) [10], derived from the same cost function via the gradient descent procedure.

The performance of the TRAM algorithm was examined by applying the procedure to train a net to produce the same input-output relations as those produced by a teacher net with similar morphology and an arbitrarily chosen weight matrix. To demonstrate the capabilities of RNN and examine the ability of the TRAM algorithm to tackle difficult problems, we applied the training procedure to solve the parity problem using as few hidden neurons as possible (for N input neurons, we used in these simulations $N-1$, $N-2$, and even $N-3$ hidden neurons). The computer simulations show a rapid convergence of the training procedure for all cases, and a high percentage of converging cases starting with an arbitrarily chosen weight matrix.

The computer simulations performed within the framework of this paper were limited to problems with binary input and output vectors as well as to RNN that produce stable endpoints. However, one should note that the algorithm is useful for a much wider variety of problems:

[1]One exception to the concept of direct weight modification is the "moving targets" algorithm presented by Rohwer [9] for producing trajectories with respect to a certain continuous input. This algorithm is able to use the continuous system representations for defining the required weight matrix modifications by solving a large set of linear equations.

- Although the TRAM algorithm is designed to train RNN that produce stable endpoints, one can use the same procedures to train RNN that produce output trajectories, since the latter are actually special cases of the former. The effectiveness of the algorithm for this task with respect to the performance of the above-mentioned algorithms is not examined within the framework of this paper.

- The system configuration examined in this paper includes binary input and output vectors; however, the same procedures can be used to train RNN with continuous input and output vectors (with hidden binary units). The performance of the algorithm for problems with continuous input or output vectors is not examined in this paper.

The theoretical basis of the TRAM algorithm for continuous and binary neurons is explained in sections 2 and 3, respectively, while the implementation of the algorithm in a complete training procedure is presented in section 4. The examination of the algorithm via computer simulations is presented in section 5.

2. Modification rules for recurrent nets with continuous neurons

Most of the training methods that exist for RNN and feed-forward (FF) nets are based on direct modification of the weight matrix, as derived from a gradient descent procedure, designed to decrease a defined cost function. In conjunction, the basis of the TRAM algorithm is the use of the network representations in each time step as dynamical parameters together with the weight matrix elements, which are modified to minimize the proper cost function.

The modification rules for these representations, related to the current system configuration, are derived from an energy function similar to the one used originally in the recurrent back-propagation (RBP) algorithm:

$$E = \sum_{p=1}^{P} \sum_{k=1}^{N^{\text{out}}} (v_k^{p,o} - \tau_k^p)^2 \tag{1}$$

where τ^p is the desired output vector related to the p vector out of P training vectors used in the training procedure; $v^{p,o}$ is the continuous-value output vector in the stable state of the system, related to the p training input vector; and N^{out} is the number of output neurons in the system. We will use the index k whenever we want to emphasize that we regard only the output neurons; other indexes are related to all of the neurons. The dynamical equation of the system in a certain time step t is of the form

$$v_i^p(t+1) = f\left(\sum_{j=1}^{N} W_{ij} v_j^p(t)\right) = f\left(u_i^p(t+1)\right) \tag{2}$$

where

$$u_i^p(t+1) \equiv \sum_{j=1}^{N} W_{ij} v_j^p(t) \tag{3}$$

The function f, which represents the neural response, is considered a nonlinear operator acting on the product of the weight matrix W and the vector $v^p(t)$, thus connecting the representations in two consecutive time steps $v^p(t)$ and $v^p(t+1)$.

As with the BP algorithm, we shall search for a procedure that minimizes the energy E defined by equation (1). The system representations and the training rules for modifying the weight matrix are easily obtained for FF networks [1]; however, deriving similar modification rules directly for the RNN configuration is much more complicated. We therefore regard the RNN as an FF network with an infinite number of layers: each layer of length N, representing the state of the entire system at a certain time step, is connected to the next layer by the *same* weight matrix W. Assuming that for a certain pattern the system reaches a stable state in time step T, one can sketch the configuration of the FF net as in figure 1.

Although we are interested in minimizing the energy function defined by equation (1), it will be useful to define a set of energy functions in each of the time steps for deriving the various modification rules. *Assuming* that we know the proper representation $\tau^p(t)$ required for each time step t to produce the proper output vector, we define the following set of energy functions, for each time step t, similar to equation (1):

$$E(t) = \sum_{p=1}^{P} \sum_{i=1}^{N} \left(v_i^p(t) - \tau_i^p(t)\right)^2 \tag{4}$$

where N is the *total* number of neurons in the net. For simplicity we will derive the modification rules by considering only the mutual effect of two contiguous layers, *assuming* we know the desired output for the second layer, $\tau^p(t+1)$. We consider both the weight matrix and the system representation at time t as the free parameters for minimizing the energy function $E(t+1)$ related to $v^p(t+1)$, the actual representation of layer $t+1$.

The derivative of the energy function $E(t+1)$ is of the form

$$\frac{dE(t+1)}{d\mathcal{T}} = \frac{\partial E(t+1)}{\partial W}\frac{dW}{d\mathcal{T}} + \sum_{p=1}^{P} \frac{\partial E(t+1)}{\partial v^p(t)}\frac{dv^p(t)}{d\mathcal{T}} \tag{5}$$

where $\mathcal{T}$ is the training index and the derivatives are applied to each interconnection weight and each neuron of the representation vectors. The following changes in W and v^p shall assure a negative contribution to the energy function:

$$\begin{aligned}\Delta W_{ij} &= -\frac{\partial E(t+1)}{\partial W_{ij}} \\ &= -\eta_w \left(v_i^p(t+1) - \tau_i^p(t+1)\right) f'\left(u_i^p(t+1)\right) v_j^p(t)\end{aligned} \tag{6}$$

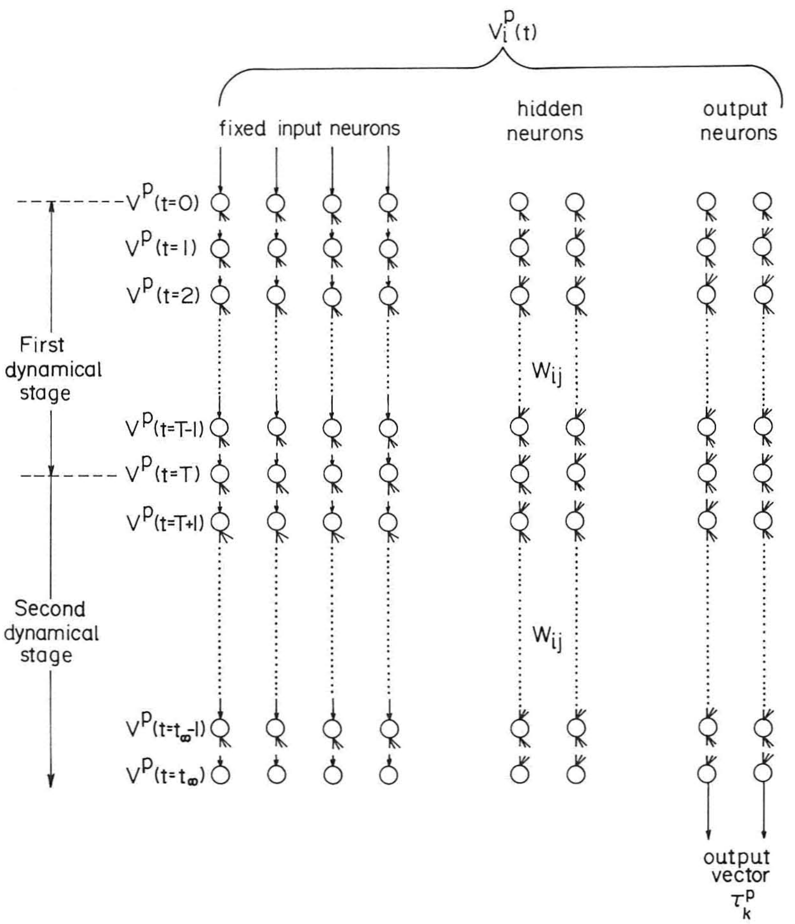

Figure 1: Development of a fully recurrent net with a stable end vector in time.

$$\begin{aligned}\Delta v_j^p(t) &= -\frac{\partial E(t+1)}{\partial v_j^p(t)} \\ &= -\eta_v \sum_{i=1}^{N} \left(v_i^p(t+1) - \tau_i^p(t+1)\right) f'\left(u_i^p(t+1)\right) W_{ij} \qquad (7)\end{aligned}$$

where f' stands for the derivative of f with respect to the argument in the parens, and η_w and η_v are convergence coefficients.

Allowing these changes to both the system representations and the weights, the energy function $E(t+1)$ will decrease with each iteration and converge to a minimum value. The order of performing the various modifications is not determined according to the gradient descent procedure, so they can therefore be applied in principle to each "training vector" $v^p(t)$ taken one at a time or to any number of them taken in parallel. However, this set of modification rules is related to the connection between *two contiguous time steps*, decreasing the cost function $E(t+1)$, whereas we are interested in minimizing the *global cost function* (equation (1)) via the weight matrix that is common to *all* time steps and patterns. Therefore we will first modify *all* of the system representations according to equation (7), and then apply the required changes in the trajectories to weight matrix modifications.

We will now examine explicitly the application of the set of modification rules (equations (6) and (7)) for minimizing the global cost function (equation (1)). The RNN presented in figure 1 includes two different dynamical stages:

1. the trajectory, between $t = 0$ and $t = T$; and
2. the stable state, presented as a constant representation between $t = T$ and $t \to \infty$.

We start by modifying the stable state of the system, applying the modification rule (equation (7)) to the system representations in the second stage. We assume that we obtained an erroneous output vector at the stable state $v^{p,o} \neq \tau^p$, where

$$\tau_k^p \equiv \tau_k^p(t_\infty) \tag{8}$$
$$v_k^{p,o} \equiv v_k^p(t_\infty) \tag{9}$$

for $t_\infty \to \infty$ and $k \in$ output neurons.

Since we are interested in a stable output vector of value τ_k^p, the first modification of the stable state representation that yields the desired result is straightforward: we simply replace the actual output neurons with those of the desired vector τ_k^p. The first modification will therefore be

$$\Delta v_i^p(t_\infty) = \tau_i^p(t_\infty) - v_i^p(t_\infty) \tag{10}$$

where the only actual modifications occur in the output neurons since the hidden neurons of τ^p are arbitrary, and are therefore chosen to be equal to those of v^p. The role of this modification is to correct the output neurons of the stable state to the desired values in all of the time steps related to the second dynamical stage. We therefore apply this modification to *all* of the time steps where $T \leq t \leq t_\infty$.

The next step is to modify the previous system representation in time step $t_\infty - 1$ in order to support the desired output vector. Applying equation (7) we get the modification for the $t_\infty - 1$ representation:

$$\Delta v_j^p(t_\infty - 1) = -\eta_v \sum_{i=1}^{N} \left(v_i^p(t_\infty) - \tau_i^p(t_\infty)\right) f'(u_i^p(t_\infty)) W_{ij} \tag{11}$$

Note that the modification is actually related only to the hidden neurons, since the input neurons remain fixed and the output neurons in each time step are corrected by the use of equation (10) in all time steps related to the stable phase. After applying the modification rule for the system representation in time step $t_\infty - 1$, we obtain a new desired vector:

$$\tau_j^p(t_\infty - 1) = v_j^p(t_\infty - 1) + \Delta v_j^p(t_\infty - 1) \tag{12}$$

This new target vector defines a new energy function similar to equation (4) for time step $t_\infty - 1$, from which one can derive the required modifications for previous time steps. Introducing the new target vector $\tau_j^p(t_\infty - 1)$ and the new energy function $E(t_\infty - 1)$, one can apply rule (7) to obtain the modification required for the system representation at time step $t_\infty - 2$:

$$\Delta v_j^p(t_\infty - 2) = -\eta_v \sum_{i=1}^{N} \left(v_i^p(t_\infty - 1) - \tau_i^p(t_\infty - 1)\right) f'\left(u_i^p(t_\infty - 1)\right) W_{ij} \tag{13}$$

Writing the modified vector explicitly—using the fact that $v_i^p(t_\infty - 1) = v_i^p(t_\infty)$ and $f'\left(u_i^p(t_\infty - 1)\right) = f'\left(u_i^p(t_\infty)\right)$ for all stable state representations—one obtains

$$\begin{aligned}&\Delta v_j^p(t_\infty - 2) = \\ &\quad - \textstyle\sum_{i=1}^{N} \left(v_i^p(t_\infty) - \tau_i^p(t_\infty)\right) \left[\eta_v f'\left(u_i^p(t_\infty)\right) W_{ij} + \eta_v^2 \left([f'(u^p(t_\infty))W]^2\right)_{ij}\right]\end{aligned} \tag{14}$$

We shall thereafter apply the same procedure to all previous time steps of the stable phase, thus obtaining an infinite series of required changes for the system representations. Accumulating all of the changes, we obtain the ultimate modification required for the vectors that represent the stable state. Since all of the vectors of the stable phase are equal, we will modify the *first vector* $v^p(T)$; all other vectors related to the stable phase will get identical representations:

$$\begin{aligned}\Delta v_j^p(T) = \tau_j^p(t_\infty) - v_j^p(t_\infty) - \sum_{i=1}^{N} \left(v_i^p(t_\infty) - \tau_i^p(t_\infty)\right) \cdot \Big[\eta_v f'\left(u_i^p(t_\infty)\right) W_{ij} \\ + \eta_v^2 \left([f'(u^p(t_\infty))W]^2\right)_{ij} + \eta_v^3 \left([f'(u^p(t_\infty))W]^3\right)_{ij} + \cdots\Big]\end{aligned} \tag{15}$$

Using the identity matrix δ_{ij}, one obtains a new form:

$$\begin{aligned}\Delta v_j^p(T) = \sum_{i=1}^{N} \left(\tau_i^p(t_\infty) - v_i^p(t_\infty)\right) \cdot \Big[\delta_{ij} + \eta_v f'\left(u_i^p(t_\infty)\right) W_{ij} \\ + \eta_v^2 \left([f'(u^p(t_\infty))W]^2\right)_{ij} + \eta_v^3 \left([f'(u^p(t_\infty))W]^3\right)_{ij} + \cdots\Big]\end{aligned} \tag{16}$$

It can easily be shown that this infinite series is identical to the inversion of a matrix of the form

$$L_{ij} \equiv \delta_{ij} - \eta_v f'\left(u_i^p(t_\infty)\right) W_{ij} \tag{17}$$

Using the new form, equation (16) becomes

$$\Delta v_j^p(T) = \eta_v \sum_{i=1}^{N} \left(\tau_i^p(t_\infty) - v_i^p(t_\infty)\right)[L^{-1}]_{ij}$$
$$\equiv \eta_v \sum_{k=1}^{N^{\text{out}}} \left(\tau_k^p(T) - v_k^p(T)\right)[L^{-1}]_{kj} \tag{18}$$

After implementing the modification of the stable state, expressed by equation (18), we can then apply the weight modification rule (equation (6)) to obtain the required weight matrix modifications that result from the modification of the stable state:

$$\Delta W_{ij} = \eta \sum_{k=1}^{N^{\text{out}}} \left(\tau_k^p(T) - v_k^p(T)\right) f'\left(u_i^p(T)\right) v_j^p(T)[L^{-1}]_{ki} \tag{19}$$

where $\eta \equiv \eta_v \eta_w$. Surprisingly we obtain the same rule as in references [4] and [5], even though we considered an RNN with discrete time steps whereas [4] and [5] considered an RNN with continuous time flow.[2]

Since the weight matrix is common to both the first and second dynamical stages, any weight modification prior to determining the representations required for the first phase of the trajectory might result in a *completely* different trajectory and stable point. We therefore should first continue the trajectory modifications until the full trajectories are defined, and only then modify the weight matrix common to all of them.

Once the new stable vector has been defined using equation (18), we can apply the modification rule (equation (7)) to modify the representation of the prior time step, $T-1$, then adopt the new stable vector $v^p(T) + \Delta v^p(T)$ as the desired vector:

$$\Delta v_j^p(T-1) = -\eta_v \sum_{i=1}^{N} \left(v_i^p(T) - \tau_i^p(T)\right) f'\left(u_i^p(T)\right) W_{ij} \tag{20}$$

We can therefore obtain a new representation for the $T-1$ time step via the modification rule mentioned above. Note that this time we modify *both the hidden and the output neurons*, and we do not add any enforced correction to the output neurons as we did for the stable state (equation (10)).

We can apply the same rules for all of the trajectories up to the first time step and to all trajectories related to the various input vectors, and thus obtain a new set of trajectories to be implemented by the weight modification rule (equation (6)).

This part of the procedure actually retrieves the BP-through-time algorithm [2] since the modification rules (6) and (7) have to retrieve the BP algorithm exactly [1]. The only difference between this procedure and the BP-through-time algorithm is that here the problem of defining the stable vector was tackled differently.

[2]A similar result, obtained via other considerations, was also published recently by Williams and Peng [11].

This part of the procedure also shows resemblance to other RNN training algorithms that produce trajectories in time [9, 7]. One of these algorithms [9] explicitly makes use of modifying system representations to obtain the weight matrix modifications, although these modifications are obtained by solving a large set of linear equations instead of the iterative solution used in this work.

The procedure presented thus far shows the relation between the TRAM procedure and currently used training algorithms. However, we do not expect a significant improvement due to the minor differences between the algorithm *in its continuous form* and other currently used algorithms. The improvement in the training performance will result from discretizing the network and the algorithm.

3. Modification rules for recurrent nets with binary neurons

After proving the consistency of the energy minimization approach for the continuous-valued net with that of RBP algorithms, we apply the same theoretical tools to the case of a discrete-valued net. We define an energy function similar to the one used for the continuous-valued net (equation (1)):

$$E = \sum_{p=1}^{P} \sum_{k=1}^{N^{\text{out}}} (v_k^{p,o} - \tau_k^p)^2 \tag{21}$$

All notations are similar to those used in equation (1) with one difference: the operator f, which represents the neural response in the dynamical equation of the system (equation (2)), is now defined as the *sgn* function, and therefore all neurons are of a binary representation $(+1, -1)$.

As in the previous case, we shall search for a procedure to minimize the energy E, described above, by allowing modifications to both the trajectories and the weight matrix. Applying similar methods to those mentioned above to obtain the trajectory and weight modifications for the discrete case will face one difficulty: the modifications for the trajectory and the weight matrix depend on the derivative of the neural response as shown earlier (equations (6) and (7)). Since the $sgn(x)$ function has zero derivative for most of its range, we approximate it by a function that has a small, almost constant, positive derivative along most of its dynamic range, as explained explicitly in [1]. We also neglect the region near $x = 0$, where the derivative is not constant, by defining the width of this area to be smaller than our resolution.

The trajectory and weight matrix modifications, as expressed in equations (6) and (7), are applicable also to the discrete case. Note, however, that the derivative function f', which is always positive, can be omitted in both equations: in the equation for discrete trajectory modifications we are interested only in sign changes, whereas in the equation for weight matrix modifications the contribution of the derivative f' can be included in the convergence coefficient η. By expressing the modification equations explicitly one obtains

$$\Delta W_{ij} = -\eta_w \left(v_i^p(t) - \tau_i^p(t)\right) v_j^p(t-1) \tag{22}$$

$$\Delta v_j^p(t) = -\sum_{i=1}^{N} \left(v_i^p(t+1) - \tau_i^p(t+1)\right) W_{ij} \tag{23}$$

Note that the required weight matrix modifications ΔW_{ij} have the form of the perceptron learning rule for each pair of system representations related to successive time steps. Applying the modification of the trajectories is more complicated due to the discrete nature of the representations. The modification of a system representation (equation (23)) requires flipping some of its bits. Such flips will be enforced whenever the modification term Δv_j^p is of opposite sign to the value of v_j^p.

Due to the coefficient η, the weight changes (equation (22)) contribute much less to the energy function than those due to the trajectory changes (equation (23)). This might lead to traps in local minima and to situations in which the slow weight modifications cannot implement the rapidly changing trajectories in the weight matrix. These obstacles can be avoided by updating the weights several times after defining the trajectory, thus allowing the system to stabilize the weight matrix in a global minimum. The number of times one should perform the weight change procedure after defining the trajectories is determined experimentally. In the simulations presented below this number was chosen to be roughly the number of input neurons in the system; choosing other "reasonable" values does not have a significant effect on the results.[3]

Since each trajectory modification results in a weight matrix modification, we should minimize the number of trajectory changes to avoid excessively restrictive demands that cannot be implemented using weight matrix modifications. Minimizing the number of modifications requires selecting the changes that contribute most to the energy functions, namely those whose implementations might prevent the use of additional modifications. Estimating the contributions of the various modifications can be achieved by comparing the energy difference before and after the modification:

$$\Delta E(t+1) = \sum_{p=1}^{P}\sum_{i=1}^{N}\left[f\left(\sum_{j=1}^{N} W_{ij}\left(v_j^p(t) + \Delta v_j^p(t)\right)\right) - \tau_i^p(t+1)\right]^2 - \sum_{p=1}^{P}\sum_{i=1}^{N}\left[f\left(\sum_{j=1}^{N} W_{ij} v_j^p(t)\right) - \tau_i^p(t+1)\right]^2 \tag{24}$$

[3]One can make a rough estimate for this number either in the form presented in our previous paper [1], comparing the energy contributions of the two changes, or by using PLR estimating techniques since this is a perceptron problem of $O(N^2)$ connections and $O(P \cdot L)$ patterns (P is the number of patterns and L is the number of layers). However, since these estimates are not accurate and are rather useless, we omitted them from this paper.

Using the discrete representation of the neurons and expanding $\Delta E(t+1)$ around the argument $u_i^p(t+1)$, one obtains the following expression for the energy difference [1]:

$$\Delta E(t+1) = -\sum_{p=1}^{P}\sum_{i=1}^{N}\sum_{j=1}^{N} f'\left(u_j^p(t+1)\right) W_{ij}\Delta v_j^p(t)\tau_i^p(t+1) \tag{25}$$

Since f' is a positive constant, as indicated earlier, equation (25) becomes

$$\Delta E(t+1) \propto -\sum_{p=1}^{P}\sum_{i=1}^{N}\sum_{j=1}^{N} W_{ij}\Delta v_j^p(t)\tau_i^p(t+1) \tag{26}$$

Therefore, the neural flips with maximal contribution will be those for which the expression in equation (26) is minimal, that is, when $\sum_{p=1}^{P}\sum_{i=1}^{N} W_{ij}\Delta v_j^p(t)\tau_i^p(t+1)$ is maximal. The need for a bit flip will be examined for these neurons selected with respect to all system representations related to the stable state of the various training patterns.

Relating this selection mechanism for the second dynamical stage is simple since we choose to modify only the most contributing neurons in each backward step until the procedure comes to a halt. The actual modification following the entire procedure affects only a single meaningful representation related to time step T, thus having a limited effect on the weight matrix. The role of this weight matrix modification is to keep the end vector stable, thus enforcing a two (identical) layer restriction.

However, using this rule for the first dynamical stage is more complicated since each modification of a representation influences the energy of the previous time step. We therefore should examine the cumulative energy contribution related to *all time steps*,

$$\sum_{p=1}^{P}\sum_{i=1}^{N}\sum_{t=1}^{T} W_{ij}\Delta v_j^p(t)\tau_i^p(t+1) \tag{27}$$

Calculating equation (27) is simple in theory, but in implementing this criterion we face a practical problem since each modification of a representation in a certain time step will affect all of the representations related to preceding time steps, and will thus affect the summation expressed in equation (27). The method used to avoid the problem is to select and modify the most contributing neurons in time step t' according to the cumulative energy contribution in the backward pass up to that stage, that is, the contribution included between time steps t' and T, $\sum_{p=1}^{P}\sum_{i=1}^{N}\sum_{t=t'}^{T} W_{ij}\Delta v_j^p(t)\tau_i^p(t+1)$. These modifications *will not actually be implemented* at first, but will be regarded *as if* they were implemented for further modifications related to prior time steps. Once the entire trajectory is "modified" in this manner, we conclude the summation of equation (27) by selecting the *most contributing neurons* over all patterns and time steps. Here also the need for a bit flip will be examined for the selected neurons with respect to all system representations related to the various training patterns.

The last significant point we should cover before turning to the actual implementation of the algorithm is the question of the temporal location of the stable point in the trajectory. Defining the new stable point by performing the above-mentioned modification and selection rules (equations (26) and (23)) yields a new desired stable vector. This vector might resemble a representation related to a time step earlier than T, and thus it might be better to advance the temporal location of the stable vector to an earlier stage. Two methods have been considered to define the location of the stable vector along the trajectory:

- the representation in which the activation most closely resembles the stable vector (t' where $\sum_{i=1}^{N} u_i^p(t')v_i^p(T)$ is maximal); and
- a random point between $t = 1$ and $t = T$.

Due to the extra computation required for the first possibility, and in order to insert some stochasticity into the algorithm, we used the second method in the simulations described below. However, we should note that the algorithm performance was similar when the first method was applied.

4. The complete algorithm

We will now combine the above-mentioned modification rules into a complete algorithm for an RNN with stable endpoints and binary representation. The various stages of the algorithm described below are also presented as a flowchart in figure 2.

Initialization

The weight matrix is chosen randomly while keeping the input vector fixed ($W_{ii} = 1$, $W_{ij} = 0$, $\forall i \in$ input neurons, $j \notin$ input neurons).

Introduction of training patterns

The training patterns are introduced to the system, producing the current trajectories that are related to the training ensemble. Trajectories related to input patterns that produce a proper output pattern are adopted unchanged and are not modified throughout the algorithm.

Definition of stable states

A backward pass is performed for the dynamical stage according to the modification equation (23) and the following paragraphs. This procedure is performed for each of the training patterns one after the other to obtain the set of desired stable vectors related to the training ensemble. The stable vectors obtained by this process will define the required representations of the hidden neurons for the stable vectors together with the output neurons defined previously by the training requirements. We start by modifying the

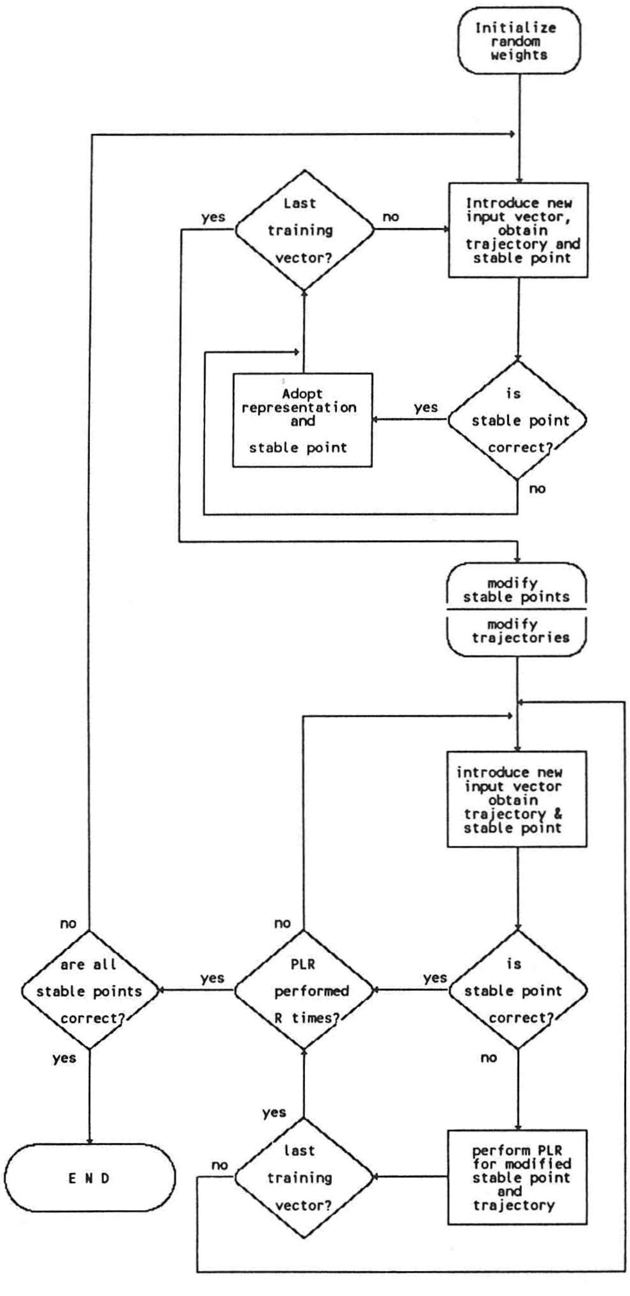
Initialize random weights
Introduce new input vector, obtain trajectory and stable point
Last training vector?
yes
no
is stable point correct?
yes
no
Adopt representation and stable point
modify stable points
modify trajectories
introduce new input vector obtain trajectory & stable point
is stable point correct?
PLR performed R times?
are all stable points correct?
perform PLR for modified stable point and trajectory
last training vector?
E N D

Figure 2: The TRAM algorithm: a flow chart.

most contributing neurons (according to equation (26)) in the system representations related to time step $t_\infty - 1$ (i.e., the time step that precedes the "last layer"), using the notations of figure 1. For carrying out the modification and for selecting the most contributing neurons, we define the current stable vector, with modified output neurons in agreement with the desired output vector, as the initial stable target vector. By updating the representation related to time step $t_\infty - 1$ (containing *proper output neurons*) we obtain a target representation for this step that differs from the actual vector related to it. Due to this difference we obtain in the same manner the target vector for time step $t_\infty - 2$; we assume that the desired stable vector of the system in time step $t_\infty - 2$ is the same as for time step $t_\infty - 1$—that is, $\tau^p(t_\infty - 2) \equiv \tau^p(t_\infty - 1)$—and we select the neurons to be modified in accordance with equation (23). The target vector for this modification rule is $\tau^p(t_\infty - 1)$, the actual "output" vector for this layer is $v^p(t_\infty - 1)$, and the target vector to be modified according to equation (23) is $\tau^p(t_\infty - 2)$. In the same manner we modify the system representations related to time steps $t_\infty - 3$, $t_\infty - 4$, and so forth until this procedure comes a halt, that is, until the desired target vector is no longer modified. This new vector is our desired stable vector for the rest of the procedure.

The modification procedure might cause a certain hidden neuron of a target "stable vector" to flip back and forth during the backward pass due to alternating conditions in the entire target vector. In order to prevent this possibility and to assure the termination of the stable state modification procedure, a limiting condition is enforced that prevents a bit flip if the energy contribution related to it is higher than the previous flip of the same neuron.

An additional modification of the algorithm for improving its performance is to weight the representation modifications required by output neurons differently from those required by hidden neurons, since the representations of the output neurons are strictly defined whereas the representations of the hidden neurons are rather fluid. This modification converts equation (23) to the form

$$\Delta v_j^p(t) = -\sum_{i=1}^{N} \left(v_i^p(t+1) - \tau_i^p(t+1)\right) \epsilon_i W_{ij} \qquad (28)$$

where $\epsilon_k > \epsilon_j$, $\forall k \in$ output neurons, $j \in$ hidden neurons. We should emphasize that the improvement due to this modification is minor. A ratio of 3 between these two coefficients was used in the simulations described below; however, choosing a ratio of 2 or 5 does not significantly influence the performance of the training algorithm.

It is also important to note that this iterative procedure can be performed *even if there is no stable state for the system.* Instead of adopting a real stable state, one can adopt any state of a certain time step as a "stable state," then perform the modification procedure in relation to that vector and obtain a new desired vector, which approximates a stable state that is in agreement with the current weight matrix. This procedure therefore offers a method that overcomes a basic problem of most RBP algorithms, namely how to tackle non-converging training patterns. The internal steps of this stage together with the following stage are presented in figure 3.

Definition of stable state position

Once the stable state is defined one must choose the proper time step in which stability occurs. In the simulations described below a random choice of this time step was used in the range $1 \leq t \leq T$. However, another criterion can be used with success similar to that described above (choosing the time step t' related to $\max \sum_{i=1}^{N} u_i^p(t')v_i^p(T)$).

Trajectory modification

The modification procedure for the rest of the trajectory is similar to the modification of the stable state. One difference in this procedure is that the desired output neurons are not defined prior to the stable state, so we therefore modify them in a manner similar to the modification of the hidden neurons, that is, according to equation (23). A second difference is the method for selecting the candidates for modification (i.e., the most contributing neurons). As indicated earlier, the source of the problem is the need to examine the contribution of a modification to the cumulative energy in all time steps, due to the influence of each modification on representations related to preceding time steps. We therefore choose during the backward pass the "most contributing neurons" according to the cumulative contribution up to the present stage (for modifying the representation related to time step t' we examine the expression $-\sum_{p=1}^{P}\sum_{i=1}^{N}\sum_{t=t'}^{T} W_{ij}\Delta v_j^p(t)\tau_i^p(t+1)$). By selecting the most contributing neurons we can update the desired representations for time step t', and thus obtain the required modifications for time step $t'-1$ and the energy contribution resulting from these modifications to be added to the cumulative energy contribution. Once the entire trajectory has been modified for each pattern, we select the most contributing neurons over all patterns and all time steps. Here also the need for a bit flip will be examined for the selected neurons according to equation (23) and with respect to all system representations related to the various training patterns.

It is advisable to select the optimal neurons to be modified *from the group of the most contributing neurons* by examining the selected neurons' activation $|u^p|$. Modification of neurons with minimal activation (in absolute value) has minimal effect on the weight matrix and is therefore preferred. One can therefore select from the "most contributing neurons" the actual neurons that will be modified using a criterion based on minimal accumulating activation $\sum_{t=1}^{T}|u_i^p(t)|$. The minor improvement obtained using this selection rule, as found in some of the simulations, does not justify the additional computation.

A flow chart summarizing the trajectory modification stage is presented in figure 4.

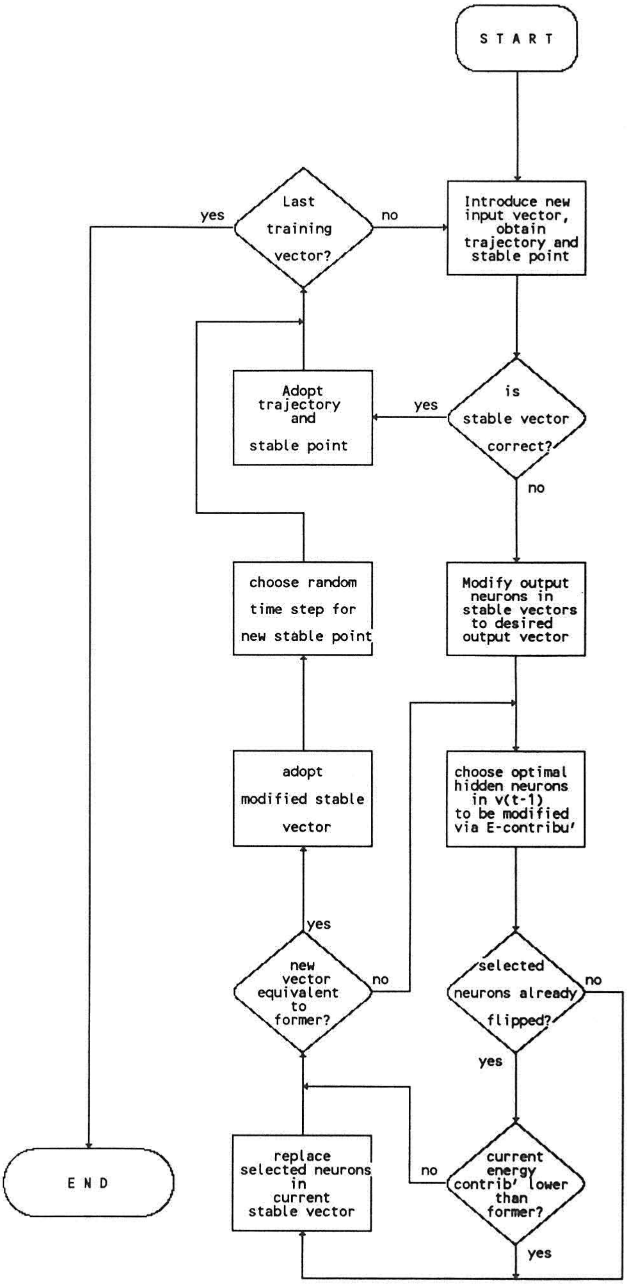
START
Introduce new input vector, obtain trajectory and stable point
Last training vector?
yes
no
is stable vector correct?
yes
no
Adopt trajectory and stable point
Modify output neurons in stable vectors to desired output vector
choose random time step for new stable point
choose optimal hidden neurons in v(t-1) to be modified via E-contribu'
adopt modified stable vector
selected neurons already flipped?
no
yes
new vector equivalent to former?
yes
no
current energy contrib' lower than former?
no
yes
replace selected neurons in current stable vector
END

Figure 3: A flow chart for defining the modified stable vector.

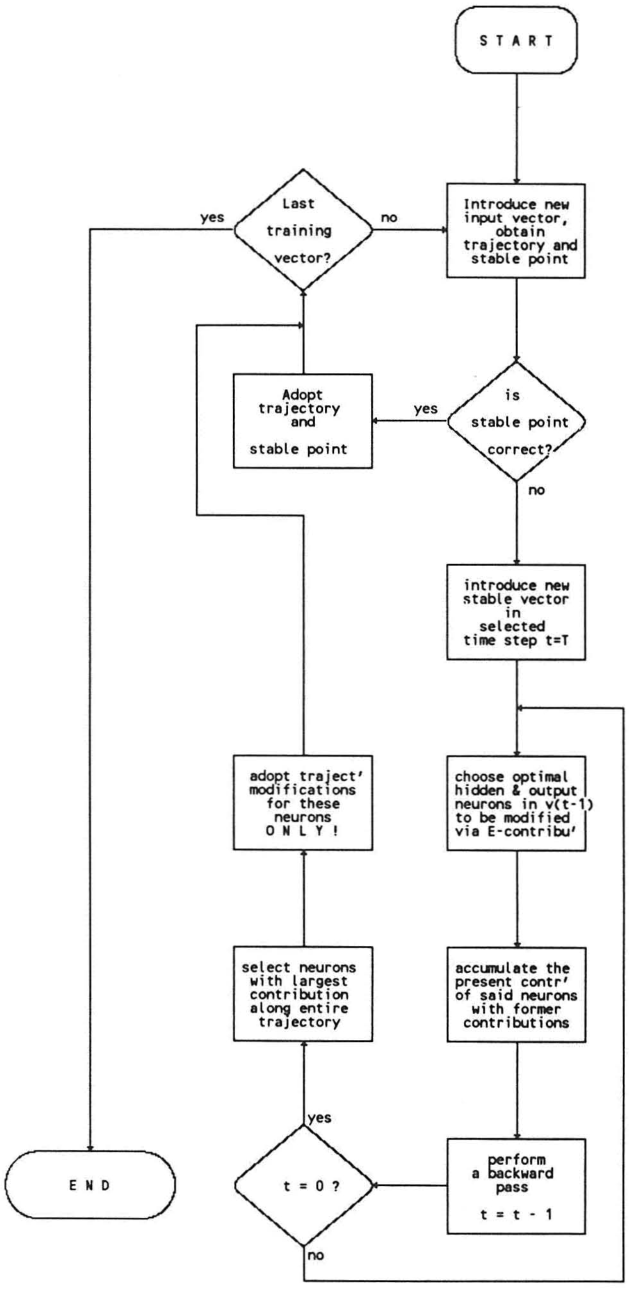
START
Introduce new input vector, obtain trajectory and stable point
is stable point correct?
yes
no
Adopt trajectory and stable point
Last training vector?
yes
no
introduce new stable vector in selected time step t=T
choose optimal hidden & output neurons in v(t-1) to be modified via E-contribu'
accumulate the present contr' of said neurons with former contributions
perform a backward pass t = t - 1
t = 0 ?
yes
no
select neurons with largest contribution along entire trajectory
adopt traject' modifications for these neurons ONLY!
END

Figure 4: A flow chart for the trajectory modification procedure.

Applying weight modification

Once the desired trajectories and stable vectors have been defined, the weight modification rule (equation (22)) is applied to adapt the weights to the new set of system representations. Since the weight modification rule is actually the PLR, we simply apply the perceptron procedure to a huge set of vector pairs that represent the full trajectories related to all input patterns. Whenever the introduction of a training vector leads to the desired stable output vector, no weight modification occurs. Moreover, the actual trajectory and stable point are adopted as the desired ones. Two additional minor modifications for the PLR that were used in our simulations are:

- If the total sum of the absolute values of all weights is less than the absolute value of the self-activating weight ($\sum_{j\neq i}|W_{ij}| < |W_{ii}|$), we set W_{ii} to a small random value.
- If the total sum of the absolute values of all weights is less than the absolute value of the threshold ($\sum_{j\neq 0}|W_{ij}| < |W_{i0}|$), we set W_{i0} to a small random value.

These modifications are applied for obvious reasons: a threshold or self-activating weight that are larger than all other weights prevent the neuron from representing any function other than a trivial one.

The weight modification procedure is applied several times in the manner described above until all of the training vectors produce the proper related output vectors or, if it fails to converge, until a certain number of iterations R is met. R is defined by the size of the net and the complexity of the problem. As a rule of thumb, in most of our simulations R was set equal to the number of input neurons.

If the PLR fails to adapt the weights for producing the proper output vectors with respect to *all* related training input vectors, we repeat the process starting from the second stage.

5. Computer simulations

One of the problems in examining the performance of the TRAM algorithm is the lack of tasks especially adequate for RNN with stable end vectors; moreover, there is no work that scales the performance of any training algorithm for such a task in relation to the size of the net nor with the size of the problem. We therefore choose two different tasks for examining the performance of the algorithm: the teacher problem, simulating a real world problem; and parity, for presenting especially difficult tasks and for demonstrating the superiority of RNN over FF networks. In our computer simulations we trained recurrent networks with various configurations and randomly chosen initial parameters to solve these two problems, for several hundred cases.[4]

[4]For small nets we used approximately 200 cases and for larger nets we examined 50 or 100 cases.

To make an adequate comparison between the TRAM algorithm and other training algorithms aimed at training RNN with stable endpoints—such as the RBP [4, 5] algorithm and the Bolzmann Machine training algorithm [6]—we examined the performance of these algorithms in training RNN on problems similar to those presented below. The performance of the TRAM algorithm for these tasks was significantly better than the other algorithms. For example, the RBP algorithm and the Bolzmann Machine training algorithm required approximately 100 and 500 iterations, respectively, to solve the simple XOR problem, compared to 20 iterations required by the TRAM algorithm. The difference in performance increases rapidly with the size and complexity of the problem, thus preventing an accumulation of statistical data required for an adequate comparison in the various cases.

The teacher problem

The teacher problem is defined by creating a net with a morphology similar to the learning net, having a randomly chosen set of weights. This net produces a set of output vectors related to the set of training input vectors. The training vectors that produce stable output vectors are used as the training set of the system since there exists a solution net for solving this problem, namely a net that produces for each input vector of the training net a corresponding stable output vector.

Since the functions created by a random choice of weights differ from one another, one must measure the training performance of the net using a statistical measure. The measure used in our simulations is the median number of learning steps required to train networks with random initial weights (the learning nets) so they perform similarly to randomly chosen networks (the teacher nets). To calculate properly the number of modification steps required to train a net, we consider *each weight modification related to a single time step and each modification of the representation related to a single step as one learning step.*

Before describing the results of the simulations it is important to describe how the teacher problem varies with respect to the number of neurons in the system. We observed that for smaller networks many trivial nets are randomly created with respect to the entire ensemble, for which the system converges to a stable vector. This is indicated by a large number of converging training vectors. As the system becomes larger the percentage of converging training vectors decreases, creating seldomly trivial nets. Figure 5 presents the median percentage of converging cases with respect to the number of hidden and output neurons in the system (the input neurons are fixed and are therefore excluded from the number of dynamic neurons in the system).

In the simulations we used various network configurations to examine how the TRAM algorithm performance is affected by the problem's complexity. The examined configurations were $N:1:1$, $N:2:1$, $N:3:1$, and $N:N:1$ (input neurons : hidden neurons : output neurons, respectively) for $N = 3$, 4, 5, 6, 7, and 8 input neurons (see figure 6), and $N:N:1$, $N:N:2$, $N:N:3$,

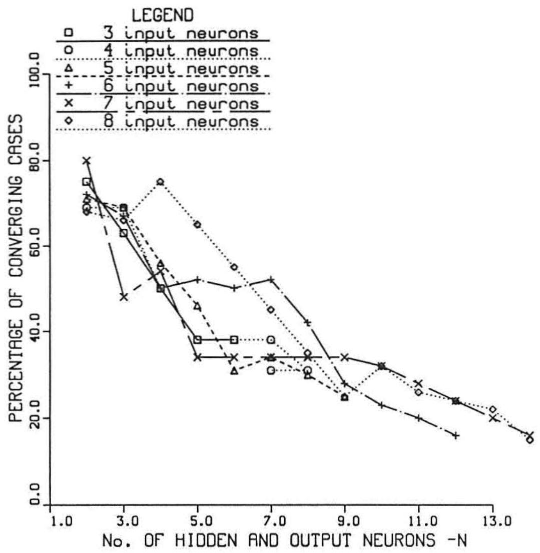

Figure 5: The percentage of training vectors that converge to a stable vector for the teacher net with N hidden + output neurons.

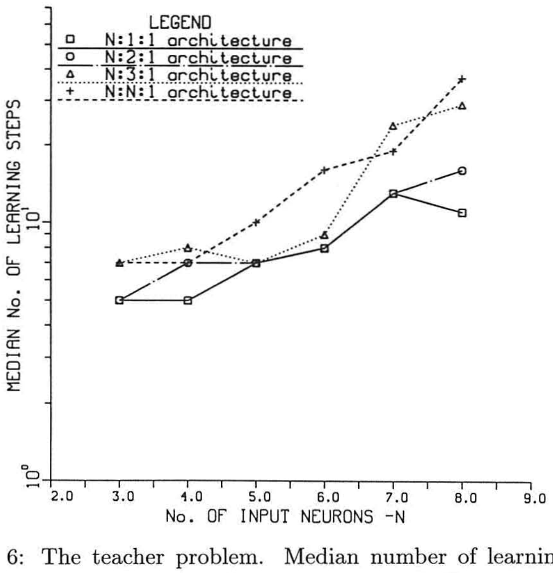

Figure 6: The teacher problem. Median number of learning steps required to train a net to perform in a manner similar to a randomly chosen teacher net of the following configurations: $N:1:1$, $N:2:1$, $N:3:1$, and $N:N:1$.

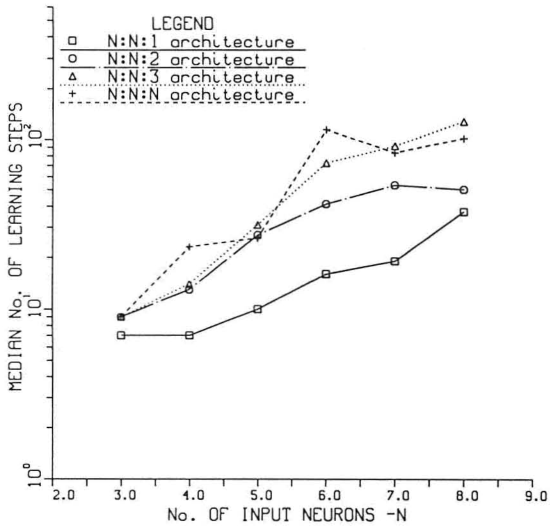

Figure 7: The teacher problem. Median number of learning steps required to train a net to perform in a manner similar to a randomly chosen teacher net of the following configurations: $N:N:1$, $N:N:2$, $N:N:3$, and $N:N:N$.

and $N:N:N$ for similar values of N (see figure 7). The algorithm parameters were: $\eta = 0.4$ for both the weight updates and the thresholds; the repetition parameter $R = N$; the number of neurons considered for modification was usually a third of the hidden neurons (obviously never less than 1); a single neuron was selected for modification throughout the entire trajectory.

The maximum number of iterations varied from 100 for the first set of configurations to 500 for the second set, each iteration being carried over the exhaustive training set. The percentage of training procedures that converged to a proper solution varied from 90% to 100%.

By examining the number of time steps required for the system to converge one can observe an interesting phenomenon, illustrated in figure 8 for two configurations: the solution networks converge to the same output vectors faster and with smaller variance than the teacher nets.

The parity problem

A parity criterion is one that yields an output of 1 when the number of $+1$ bits in the input vector is even and -1 otherwise. In the simulations we used $N = 2$, 3, 4, 5, and 6 input neurons; N, $N-1$, $N-2$, and $N-3$ neurons in the hidden layer (for those values where such a choice is applicable); and a single output neuron. The algorithm parameters were similar to those used for the teacher problem, except for the repetition parameter R since this problem is much harder. The values of R used for this problem were

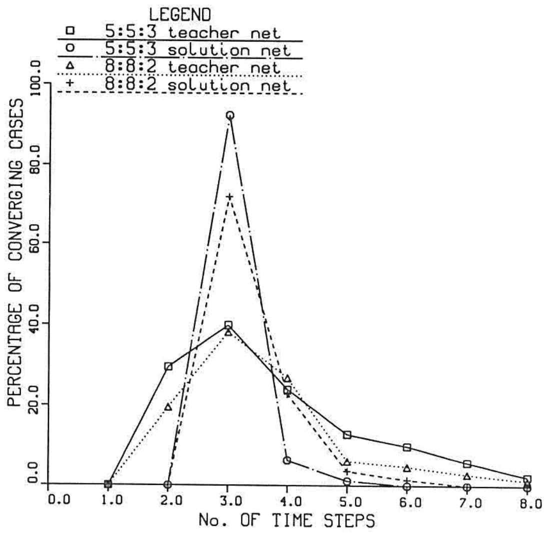

Figure 8: The distribution of the number of time steps required for RNN configurations 8 : 8 : 2 and 5 : 5 : 3 to reach a stable state: teacher net vs. solution net.

$R = 4$, 6, 7, 10, and 20, corresponding to the number of input neurons. Here also we obtained a high percentage of converging training procedures, even though it was somewhat lower than for the teacher problem (70% to 100%). Figure 9 shows the median number of steps required to train the net of this configuration to produce the proper output vectors.

It is important to note that these simulations indicate the superiority of RNN over FF nets for solving hard tasks since the parity problem for some of these configurations cannot be solved using FF nets with equal numbers of given neurons.

6. Conclusion

We have demonstrated that a continuously valued multilayer system, where both the system representations and the interconnection weights are simultaneously modified according to energy minimization principles, behaves similarly to existing RNN training algorithms such as the RBP, BP through time, and others in which only the weights are modified directly as a result of different considerations. By applying a similar approach to a discrete system we obtained the TRAM algorithm, which has some resemblance to training algorithms used for FF nets and that are based on modification of the internal representations, such as the CHIR algorithm. Computer simulations examining the performance of the TRAM algorithm on the "teacher

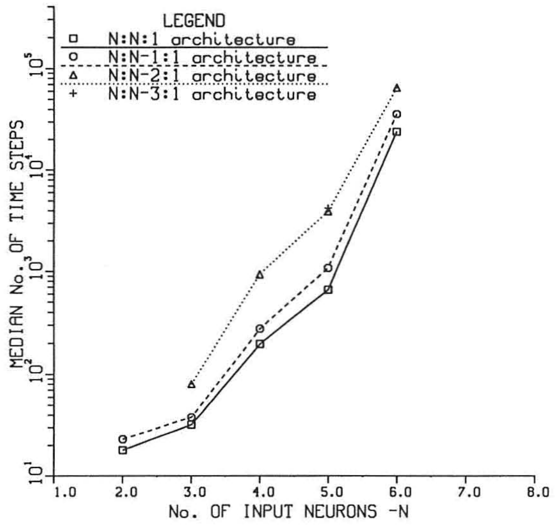

Figure 9: Parity. The median number of learning steps required to train networks of the following configurations to solve the parity problem using an exhaustive training set: $N:N:1$, $N:N-1:1$, $N:N-2:1$, and $N:N-3:1$.

problem" and the "parity problem" show a more rapid and reliable convergence than other existing methods of this type, raising the possibility of a new, rapidly converging learning algorithm for RNN.

Acknowledgments

The author would like to thank E. Marom for helpful comments. The author is supported by a stipend from the Buchmann fund, which is gratefully acknowledged.

References

[1] D. Saad and E. Marom, "Learning by Choice of Internal Representations: An Energy Minimization Approach," *Complex Systems*, **4** (1990) 107–118.

[2] D. E. Rumelhart and J. L. McClelland, *Parallel Distributed Processing* (Cambridge, MIT Press, 1986).

[3] T. Grossman, R. Meir, and E. Domany, "Learning by Choice of Internal Representations," *Complex Systems*, **2** (1989) 555–575.

[4] F. J. Pineda, "Generalization of Backpropagation to Recurrent Neural Network," *Physical Review Letters*, **18** (1987) 2229–2232.

[5] L. B. Almeida, "A Learning Rule for Asynchronous Perceptrons with Feed-Back in a Combinatorial Enviroment," *Proceedings of the First International Conference on Neural Networks*, volume II (1987) 609–618.

[6] G. E. Hinton, T. R. Sejnowski, and D. H. Ackley, "Boltzmann Machines: Constraint Satisfaction Networks That Learn," Carnegie-Mellon Technical Report CMU-CS-84-119 (Pittsburgh, 1984).

[7] R. J. Williams and D. Zipser, "A Learning Algorithm for Continually Running Fully Recurrent Neural Networks," *Neural Computation*, **1** (1989) 270–280.

[8] B. Perlmutter, "Learning Space State Trajectories in Recurrent Neural Networks," *Neural Computation*, **1** (1989) 263–269.

[9] R. Rohwer, "The Moving Targets Training Algorithm," pages 558–565 in *Advances in Neural Information Processing Systems II*, edited by D. S. Touretzky (San Mateo, CA, Morgan Kaufmann, 1989).

[10] M. L. Minsky and S. A. Papert, *Perceptrons*, 3rd edition (Cambridge, MIT Press, 1988).

[11] R. J. Williams and J. Peng, "Algorithm for Training Recurrent Networks," *Neural Computation*, **2** (1990) 490–501.