Experimental Analysis of the Real-time Recurrent Learning Algorithm

RONALD J. WILLIAMS & DAVID ZIPSER

The real-time recurrent learning algorithm is a gradient-following learning algorithm for completely recurrent networks running in continually sampled time. Here we use a series of simulation experiments to investigate the power and properties of this algorithm. In the recurrent networks studied here, any unit can be connected to any other, and any unit can receive external input. These networks run continually in the sense that they sample their inputs on every update cycle, and any unit can have a training target on any cycle. The storage required and computation time on each step are independent of time and are completely determined by the size of the network, so no prior knowledge of the temporal structure of the task being learned is required. The algorithm is nonlocal in the sense that each unit must have knowledge of the complete recurrent weight matrix and error vector. The algorithm is computationally intensive in sequential computers, requiring a storage capacity of the order of the third power of the number of units and a computation time on each cycle of the order of the fourth power of the number of units. The simulations include examples in which networks are taught tasks not possible with tapped delay lines—that is, tasks that require the preservation of state over potentially unbounded periods of time. The most complex example of this kind is learning to emulate a Turing machine that does a parenthesis balancing problem. Examples are also given of networks that do feedforward computations with unknown delays, requiring them to organize into networks with the correct number of layers. Finally, examples are given in which networks are trained to oscillate in various ways, including sinusoidal oscillation.

Introduction

Recurrent neural networks can implement dynamical systems of arbitrary complexity. To make use of this ability in cases where the dynamical system is defined only in terms of its input and output we need learning procedures capable of programming recurrent networks. A general framework for the problem was laid out by Rumelhart *et al.* (1986), who unfolded the recurrent network into a multilayer feedforward network that grows by one layer on each time step. Algorithms based on this concept, which

Ronald J. Williams, College of Computer Science, Northeastern University, Boston, MA 02115, USA; David Zipser, Institute for Cognitive Science, University of California, San Diego, La Jolla, CA 92093, USA. This research was supported by Grant IRI-8703566 from the National Science Foundation to the first author and Contract N00014-87-K-0671 from the Office of Naval Research, Grant 86-0062 from the Air Force Office of Scientific Research, and grants from the System Development Foundation to the second author.

might be called *backpropagation through time*, work well in cases where enough is known about the time structure of the problem to limit the number of layers to some reasonable, fixed value. Recently, Pearlmutter (1988) has extended backpropagation through time to continuous time approximations. Backpropagation through time deals well with dynamical problems with fixed, or at least previously known, temporal periods. Generating periodic, dynamic behavior from static inputs is an example of the kind of problem that can be learned by this technique.

Unfortunately, many interesting problems are not of this type. For example, strings of arbitrary length generated by a probabilistic finite state grammar do not have a fixed, previously known length so backpropagation through time must have some *a priori* knowledge of their maximum length to set up a system with sufficient size to learn to recognize them. Also, many problems in signal processing and speech recognition involve learning about temporal sequences with *a priori* unknown temporal properties. To get around these limitations Jordan and others (Jordan, 1986; Stornetta *et al.*, 1987; Elman, 1988) have used networks with a limited set of carefully chosen recurrent connections. These networks are able to learn some of the interesting tasks possible with continuously running networks having recurrent connections. These networks do not try to do credit assignment back through time but rather use the previous state of the network as part of the current input. This provides rich, but not total, information about the past so long as appropriate teaching signals are applied to the network. However, the ability of these networks to deal with long, confusing temporal sequences is clearly limited (Servan-Schreiber *et al.*, 1988). Mozer (1988) has attempted to overcome some of these difficulties by adding a layer of units that each have a single self-recurrent connection that is trained by a true gradient-following learning rule. This network has shown considerable promise in a variety of sequence-processing tasks.

One situation in which the use of backpropagation through time leads to a simple algorithm is when the network's actual and desired dynamics consist of settling to a fixed equilibrium state on each teaching cycle. Almeida (1987), Pineda (1988) and Rohwer & Forrest (1987) have all derived various versions of this algorithm, and Almeida has made the useful observation that the corresponding backpropagation computation, itself involving a settling operation, necessarily converges stably whenever the forward computation does. While this algorithm has the attractive computational features of the more familiar feedforward backpropagation algorithm, the requirement that both the actual and desired network dynamics have only point attractors and that any external input to the network be constant during the settling poses a strong practical limitation on the use of this approach.

To overcome the limitations of the previous approaches, an algorithm is needed that can train fully recurrent, continually running networks to implement dynamical systems described only by the temporal stream of their inputs and outputs. In this paper, we describe studies using a powerful learning procedure for networks of this kind, called real-time recurrent learning, or RTRL (Robinson & Fallside, 1987; Bachrach, 1988; Mozer, 1988; Williams & Zipser, 1989). The derivation of this algorithm has been given elsewhere (Williams & Zipser, 1989) but is repeated here for convenience. The main body of work described here consists of simulation experiments with the RTRL algorithm that demonstrate its power in a variety of tasks. RTRL can be used with networks in which any unit can be connected to any other, and any unit can receive external input. These networks run continually in the sense that they sample their inputs on every update cycle, and any unit can receive training signals on any cycle. The procedure differs from backpropagation through time in that its ability do to continuous processing frees it from any requirement for a fixed, or even bounded, epoch length. RTRL differs from the procedures that use past states for current input in that it correctly does credit assignment to past events.

While the RTRL algorithm is very powerful, it has two significant drawbacks: It requires a great deal of computation on each update cycle, and it is nonlocal. The amount of storage and computation required are independent of time but increase with the number of units in the network. In sequential computers, the algorithm requires, for n units, a storage capacity of roughly n^3, and a computation time on each cycle of order n^4. The algorithm is nonlocal in the sense that each unit must have knowledge of the complete recurrent weight matrix and error vector. However, the algorithm is very parallel and in a parallel computer its computation time can be reduced to order $n^2\log(n)$ by using a processor for each weight.

The computational requirements are not so great as to preclude the study of the algorithm on problems of considerable interest using networks of moderate size, that is, 20 to 30 units. Later we present a series of examples designed to demonstrate the power of the RTRL algorithm. The examples include demonstrations that recurrent networks can be taught tasks not possible with tapped delay lines. These are tasks that require the network to preserve state, the simplest requiring only a flip-flop. The most complex example of this kind of task is a network that learns to emulate a Turing machine that does a parenthesis balancing problem. Examples are also given of recurrent networks that learn to configure themselves into feedforward networks with the correct number of layers required by the task. Finally, examples are given of training networks to oscillate in various ways.

Derivation of the Learning Algorithm

An Exact Gradient-following Algorithm

First we derive the true error gradient-following procedure,[1] and then introduce the assumptions needed for practical algorithms. The derivation is for fully connected networks in which all units can receive input and can be taught to produce targeted output on any cycle. This form of the algorithm is completely general since it encompasses all simpler network architectures, including feedforward networks, as special cases where some of the connection weights are fixed and not trainable.

Let the network have n units, with m external input lines. Let $\mathbf{y}(t)$ denote the n-tuple of outputs of the units in the network at time t, and let $\mathbf{x}(t)$ denote the m-tuple of external input signals to the network at time t. It will be convenient in what follows to also define $\mathbf{z}(t)$ to be the $(m+n)$-tuple obtained by concatenating $\mathbf{y}(t)$ and $\mathbf{x}(t)$ in some convenient fashion. To distinguish the components of $\mathbf{z}$ representing unit outputs from those representing external input values where necessary, let U denote the set of indices k such that z_k, the kth component of $\mathbf{z}$, is the output of a unit in the network, and let I denote the set of indices k for which z_k is an external input. Furthermore, we assume that the indices on $\mathbf{y}$ and $\mathbf{x}$ are chosen to correspond to those of $\mathbf{z}$, to that

$$z_k(t) = \begin{cases} x_k(t) & \text{if } k \in I \\ y_k(t) & \text{if } k \in U. \end{cases} \tag{1}$$

.For example, in a computer implementation using zero-based array indexing, it is convenient to index units and input lines by integers in the range $[0,m+n)$, with indices in $[0,m)$ corresponding to input lines and indices in $[m,m+n)$ corresponding to units in the network.

Let **W** denote the weight matrix for the network, with a unique weight between every pair of units and also from each input line to each unit. By adopting the indexing convention just described, we can incorporate all the weights into this single $n \times (m+n)$ matrix. The element w_{ij} represents the weight on the connection of the ith unit from either the jth unit, if $j \in U$, or the jth input line, if $j \in I$. Furthermore, note that to accommodate a bias for each unit, we simply include among the m input lines one input whose value is always 1; the corresponding column of the weight matric contains as its ith element the bias for unit i.

As an example, consider the network depicted in Figure 1. There $n=2$ and $m=5$, and the 14 weights indicated form a 2×7 matrix, with each row representing the weights for a single unit.

For this analysis, we assume that the network consists entirely of semilinear units, although the technique used to derive the learning algorithm is clearly applicable to any form of differentiable unit computation whatsoever. For semilinear units it is convenient to let

$$s_k(t) = \sum_{l \in U} w_{kl} y_l + \sum_{l \in I} w_{kl} x_l = \sum_{l \in U \cup I} w_{kl} z_l(t) \tag{2}$$

denote the net input to the $k$tj unit at time t, for $k \in U$. (We have written this here in two equivalent forms; the longer one clarifies how the unit outputs and the external inputs are both used in the computation, while the more compact expression illustrates why we introduced **z** and the corresponding indexing convention above. Hereafter, we only use the latter form.)

Another assumption we make here is the use of discrete time. It is straightforward to extend this approach to continuous time models of computation, but we omit the details. For a semilinear unit, its output at the next discrete time step is expressed in terms of the net input by

$$y_k(t+1) = f_k[s_k(t)], \tag{3}$$

where f_k is the unit's squashing function. For the moment, we make no particular assumption about the nature of this squashing function (other than its differentiability).

Thus the system of equations (2) and (3), where k ranges over U, constitutes the entire (discrete-time) dynamics of the network, where the z_k values are defined by equation (1). Note that the external input at time t does not influence the output of any unit until time $t+1$.

Now that we have specified the dynamics of the network, we need to consider how we might adapt the weights in order to improve its peformance over time. The fundamental approach to deriving such an adaptation scheme, just as with the more familiar backpropagation algorithm, is to specify some measure of network performance and to compute its gradient in weight space. In the case of a dynamical trajectory, a general performance measure will be some function of the output **y** of the network over time. In particular, for the simulations reported here, we wanted the trajectory of some subset of the components of **y** to match specified values at specified times. To formulate this property, let $T(t)$ denote the set of indices $k \in U$ for which there exists a specified target value $d_k(t)$ that the output of the kth unit should match. Then define a time-varying n-tuple **e** by

$$e_k(t)=\begin{cases} d_k(t)-y_k(t) & \text{if } k\in T(t) \\ 0 & \text{otherwise.}\end{cases} \tag{4}$$

Note that this formulation allows for the possibility that target values are specified for different units at different times (as does the usual backpropagation-through-time formulation). The set of units considered to be 'visible' can thus be time-varying. Now let

$$\mathcal{J}(\tau)=\tfrac{1}{2}\sum_{k\in U}[e_k(\tau)]^2 \tag{5}$$

denote the network error at time τ. Assume that the network is run starting at time t_0 up to some final time t_1. We would like to minimize the total error

$$\mathcal{J}_{\text{total}}(t_0,t_1)=\sum_{\tau=t_0+1}^{t_1}\mathcal{J}(\tau) \tag{6}$$

over this trajectory, so we want to compute the gradient of this total error measure in weight space.

Note that

$$\mathcal{J}_{\text{total}}(t_0,t+1)=\mathcal{J}_{\text{total}}(t_0,t)+\mathcal{J}(t+1) \tag{7}$$

so that its gradient satisfies

$$\nabla_{\mathbf{w}}\mathcal{J}_{\text{total}}(t_0,t+1)=\nabla_{\mathbf{w}}\mathcal{J}_{\text{total}}(t_0,t)+\nabla_{\mathbf{w}}\mathcal{J}(t+1). \tag{8}$$

Thus as the trajectory unfolds over time, we can simply accumulate the values of the vector $\nabla_{\mathbf{w}}\mathcal{J}$ at each time step until the final time step. Since the weight change rule we seek adjusts $\mathbf{W}$ along a fixed positive multiple of $-\nabla_{\mathbf{w}}\mathcal{J}_{\text{total}}(t_0,t_1)$, the same observation applies to the weight changes themselves.

In other words, for each weight w_{ij} in the network we accumulate the value of

$$\Delta w_{ij}(t)=-\alpha\frac{\partial\mathcal{J}(t)}{\partial w_{ij}} \tag{9}$$

at each time step t along the trajectory, where α is some fixed positive learning rate. After the network has run through this trajectory, we alter each weight w_{ij} by

$$\sum_{t=t_0+1}^{t_1}\Delta w_{ij}(t). \tag{10}$$

Thus we want an algorithm that computes

$$-\frac{\partial\mathcal{J}(t)}{\partial w_{ij}}=\sum_{k\in U}e_k(t)\frac{\partial y_k(t)}{\partial w_{ij}} \tag{11}$$

at each time step t. Since the value of $e_k(t)$ is known at time t for each $k\in U$, all that remains is to find a way to compute the remaining factor, $\partial y_k(t)/\partial w_{ij}$, at time step t.

Before describing how this factor is computed, it may be helpful to gain an intuitive understanding of its meaning. Essentially, $\partial y_k(t)/\partial w_{ij}$ measures the sensitivity of the value of the output of the kth unit at time t to a small increase in the value of w_{ij}, taking into account the effect of such a change in the weight over the entire trajectory from t_0 to t. It is assumed, however, that the initial state of the network

$\mathbf{y}(t_0)$, the input over (t_0,t), and the remaining weights are not altered when determining this sensitivity.

Another observation concerning the quantity $\partial y_k(t)/\partial w_{ij}$ is that it does not depend at all on the teacher signal or the discrepancy between the desired and actual performance of the network. This is the main reason why one could expect to compute it directly from the network's actual operation, without any knowledge of the errors that the network may eventually commit sometime in the future.

To compute this factor, we simply differentiate equations (2) and (3), yielding

$$\frac{\partial y_k(t+1)}{\partial w_{ij}} = f'_k[s_k(t)]\left[\sum_{l \in U \cup I} w_{kl}\frac{\partial z_l(t)}{\partial w_{ij}} + \delta_{ik} z_j(t)\right] \tag{12}$$

where δ_{ik} denotes the Kronecker delta

$$\delta_{ik} = \begin{cases} 1 & \text{if } i=k \\ 0 & \text{otherwise.} \end{cases} \tag{13}$$

Now, since we assume that the input signals do not depend on the network weights, it follows that

$$\frac{\partial z_l(t)}{\partial w_{ij}} = \begin{cases} 0 & \text{if } l \in I \\ \dfrac{\partial y_l(t)}{\partial w_{ij}} & \text{if } l \in U, \end{cases} \tag{14}$$

so equation (12) becomes

$$\frac{\partial y_k(t+1)}{\partial w_{ij}} = f'_k[s_k(t)]\left[\sum_{l \in U} w_{kl}\frac{\partial y_l(t)}{\partial w_{ij}} + \delta_{ik} z_j(t)\right]. \tag{15}$$

Also, since we assume that the initial state of the network has no functional dependence on the weights, we have

$$\frac{\partial y_k(t_0)}{\partial w_{ij}} = 0. \tag{16}$$

These equations hold for all $k \in U$, $i \in U$, and $j \in U \cup I$.

Thus if we create a dynamical system with variables $\{p^k_{ij}\}$ for all $k \in U$, $i \in U$ and $j \in U \cup I$, and dynamics given by

$$p^k_{ij}(t+1) = f'_k[s_k(t)]\left[\sum_{l \in U} w_{kl} p^l_{ij}(t) + \delta_{ik} z_j(t)\right], \tag{17}$$

with initial conditions

$$p^k_{ij}(t_0) = 0, \tag{18}$$

we see that

$$p^k_{ij}(t) = \frac{\partial y_k(t)}{\partial w_{ij}} \tag{19}$$

for every time step t and all appropriate i, j and k.

The precise algorithm then consists of computing, at each time step t from t_0 to t_1, the quantities $p^k_{ij}(t)$, using equations (17) and (18), and then using the discrepancies $e_k(t)$ between the desired and actual outputs to compute the weight changes

$$\Delta w_{ij}(t) = \alpha \sum_{k \in U} e_k(t) p^k_{ij}(t). \tag{20}$$

The overall correction to be applied to each weight w_{ij} in the net is then given by

$$\Delta w_{ij} = \sum_{t=t_0+1}^{t_1} \Delta w_{ij}(t). \tag{21}$$

In the case when each unit in the network uses the logistic squashing function, as in the experiments reported here, the appropriate algorithm uses the fact that

$$f_k'[s_k(t)] = y_k(t+1)[1-y_k(t+1)] \tag{22}$$

in equation (17).

Real-time Recurrent Learning

The above algorithm was derived on the assumption that the weights remained fixed throughout the trajectory from t_0 to t_1. In order to allow on-line training, however, it is useful to deviate from this assumption and actually make the weight changes while the network is running. The actual algorithm we used in the experiments reported here did not make the weight change prescribed by equation (21) at the end of the trajectory, but instead immediately added $\Delta w_{ij}(t)$, as computed by equation (20), to each weight w_{ij} at time step t. Thus the resulting algorithm does not truly lead to weight changes along the negative gradient of $J_{\text{total}}(t_0, t_1)$, since the weights themselves are actually altered over the course of the trajectory. We call this algorithm for training the dynamical behaviors of arbitrary recurrent networks the *real-time recurrent learning* (RTRL) algorithm. Making the weight changes at each time step in RTRL rather than at the end of the trajectory is similar to the philosophy of training a feedforward net in an incremental fashion on a fixed sequence of patterns rather than using a batch approach which updates weights along the true gradient of total error. While the fixed-pattern-sequence incremental algorithm is not guaranteed to follow this gradient, it is known to work well in practice, presumably because the use of a small enough learning rate leads to a net weight update whose direction is a close enough approximation to the true gradient. A similar observation applies to RTRL.

One potential problem with this algorithm is that the observed trajectory may itself depend on the variation in the weights caused by the learning algorithm, which can be viewed as providing another source of feedback in the system. To avoid this, one wants the time scale of the weight changes to be much slower than the time scale of the network operation, meaning that the learning rate must be sufficiently small. On the other hand, this on-line version has the advantage that it is not necessary to define epoch boundaries during the training of the network. As the weights gradually change, the continuing trajectory is automatically a function of the new, approximately constant weights. The length of what one might consider an epoch in this version of the algorithm is determined by the time scale of the weight changes, and these epochs all overlap and blend together. A further property of this modified algorithm is that one need not explicitly consider the ending time t_1 for the trajectory being trained. This algorithm can be run continually over an indefinite time period.

It is useful to view the triply indexed set of quantities p_{ij}^k as a matrix, each of whose rows corresponds to a weight in the network and each of whose columns corresponds to a unit in the network. Looking at the update equations it is not hard to see that, in general, we must keep track of the values p_{ij}^k even for those k corresponding to units that never receive a teacher signal. Thus we must always have n columns in this matrix. However, if the weight w_{ij} is not to be trained (as would happen, for example, if we constrain the network topology so that there is no connection from unit j to unit i), then it is not necessary to compute the value p_{ij}^k for any $k \in U$. This means that this matrix need only have a row for each adaptable weight in the network, while having a column for each unit. Thus the minimal number of p_{ij}^k values needed to store and update for a general network having n units and r adjustable weights is nr. In the case where every weight is adaptable, there are $n^3 + mn^2$ such p_{ij}^k values.

Teacher-forced Real-time Recurrent Learning

There is an interesting variant of the algorithm given above that seems to be helpful in some training tasks such as stable oscillation. The idea is to replace the actual output $y_k(t)$ of a unit by the teacher signal $d_k(t)$ in subsequent computation of the behavior of the network, whenever such a value exists. We call this technique *forcing the network with the teacher signal*, or *teacher-forcing* for short. To describe this algorithm more precisely, let the *free-running state* of the network at time t be $\mathbf{y}(t)$, and define the *teacher-forced state* of the network at time t to be $\mathbf{y}(t) + \mathbf{e}(t)$. The idea is to base the future activity of the network on the teacher-forced state of the network rather than the free-running state. The dynamics of the network are thus altered during the training phase, and the corresponding learning algorithm will necessarily be somewhat different.

The modified dynamics of the network during training can be described formally as follows: Recall that $T(t)$ is the set of indices $k \in U$ for which $d_k(t)$ exists. Then, in equation (2), let

$$z_k(t) = \begin{cases} x_k(t) & \text{if } k \in I \\ d_k(t) & \text{if } k \in T(t) \\ y_k(t) & \text{if } k \in U - T(t). \end{cases} \tag{23}$$

The dynamics of the network during training are then given by equations (2) and (3), this time using this new definition of $\mathbf{z}(t)$ rather than the one in equation (1). To derive a learning algorithm for this situation, we once again differentiate both sides of equation (3) with respect to w_{ij}, yielding equation (12), as before. This time, however, note that

$$\frac{\partial z_l(t)}{\partial w_{ij}} = \begin{cases} 0 & \text{if } k \in I \\ 0 & \text{if } k \in T(t) \\ \dfrac{\partial y_l(t)}{\partial w_{ij}} & \text{if } k \in U - T(t). \end{cases} \tag{24}$$

Thus we find that

$$\frac{\partial y_k(t+1)}{\partial w_{ij}} = f_k'[s_k(t)] \left[\sum_{l \in U - T(t)} w_{kl} \frac{\partial y_l(t)}{\partial w_{ij}} + \delta_{ik} z_j(t) \right] \tag{25}$$

for this situation. This means that for the values $p_{ij}^k(t)$ to be equal to $\partial y_k(t)/\partial w_{ij}$ we must alter their dynamics to use

$$p_{ij}^k(t+1) = f_k'[s_k(t)] \left[\sum_{l \in U - T(t)} w_{kl} p_{ij}^l(t) + \delta_{ik} z_j(t) \right], \tag{26}$$

rather than equation (17), with the same initial conditions (equation 18), as before.

Note that equation (26) is the same as equation (17) if we treat the values of $p^l_{ij}(t)$ as zero for all $l \in T(t)$ when computing $p^k_{ij}(t+1)$. Thus in this variant of the algorithm, we perform essentially the same computations as before, except that we set the appropriate p^k_{ij} values to zero after they have been used to compute the Δw_{ij} values but before computing any of the $p^k_{ij}(t+1)$ values.

The steps of this algorithm are then: (a) compute the free-running state of the network at time t based on the teacher-forced state of and input to the network at time $t-1$; (b) compute the $p^k_{ij}(t)$ values just as with the earlier version of the algorithm; (c) compute the weight updates for time t just as with the earlier version of the algorithm; (d) set to zero all values $p^k_{ij}(t)$ for which the teacher value $d_k(t)$ exists; (e) set the teacher-forced state of the network at time t, in preparation for the next time step.

This teacher-forced version of the algorithm is thus essentially the same as the earlier one, with two simple alterations: using the teacher-forced state to compute future activity in the network and setting the appropriate p^k_{ij} values to zero after they have been used to compute the Δw_{ij} values.

The teacher-forced version of the algorithm appears to be required for adjusting the weights in a network in such a way that the dynamical behavior of the network is altered in a qualitative manner, such as creating new attractors or changing the form of existing ones in discontinuous ways. As will be described below, it has been found to be crucial for training networks to oscillate. However, there are also situations for which it is clearly of no use or otherwise inappropriate. One obvious case where it is of no use is when the units to be trained do not feed their output back to the network, as in the networks used by Elman (1988). Furthermore, note that the error measures being minimized by RLTR and teacher-forced RTRL are different in general, although any setting of the weights which gives zero for one measure also gives zero error for the other. This means that unless one obtains zero error when using the teacher-forced version, the solution found need not give minimum squared error between the desired and actual trajectories of the free-running network. In fact, it is easy to devise examples where the network is incapable of matching the desired trajectory and the result obtained using teacher forcing is far different from a solution giving minimum squared error for the free-running network.

It should be pointed out that this technique of forcing the network with the teacher signal, although not described using this terminology, appears implicitly or explicitly in the work of others. For example, Jordan's (1986) algorithm for training networks to producing sequential patters uses it, and Pineda's (1988) method for creating content-addressable memories is a special case in which the teacher signal is a constant. This idea also appears in the adaptive signal processing literature as an 'equation error' technique for synthesizing linear filters having an infinite impulse response (Widrow & Stearns, 1985, pp. 250–253).

Simulation Experiments

One of our main goals in the simulation experiments described here was to determine the ability of the algorithm to solve problems with a minimum of *a priori* information. In this spirit we used the same uniform network architecture for each case. The starting network consisted of a set of n units fully interconnected with initially random weights. All units in the network received all the inputs. The only way units were distinguished was that during training only the subset of units whose outputs were to be trained contributed to the values of the error vector $\mathbf{e}$. The unforced version of the algorithm was used for all cases except those that involved the learning of oscillations.

Exclusive OR With Delay

Exclusive OR (XOR) is a nonlinearly separable Boolean function requiring at least two processing cycles to compute. It has been extensively studied with feedforward backpropagation networks. Here we wanted to demonstrate that a recurrent network could learn to do XOR continuously with two simultaniously changing inputs by organizing itself into a feedforward network with the appropriate number of layers. In a network receiving input on each update cycle, the teacher must be delayed by at least two cycles relative to the input that is bing XORed. If the teacher is delayed more than two cycles, the network must develop some internal mechanism, such as additional layers, to take care of this added delay. This formulation of XOR learning differs in a significant way from that used in the case of feedforward networks. Here the network is being trained to carry out the XOR operation continuously. This means that several different partially complete computations are present in the network at the same time in a pipelined fashion.

To see if our algorithm could deal with this problem we started with fully connected networks, as described earlier, using two inputs and a bias. Each input line received a continuous stream of randomly chosen 0s and 1s. In this, and all other Boolean examples, the outputs were trained to be 0 or 1. A network was considered to have successfully learned a taks if its outputs were correct for a sufficiently long testing period (typically 1000 cycles for XOR problems). The criterion of correctness was that outputs that should be 1 were greater than or equal 0.5 and outputs that should be 0 were less than 0.5.

A single unit of the network was taught on each cycle to output the XOR of the input occurring two or more cycles previously. A learning rate of 4.0, which seemed near optimal, was used. When the delay between input and teacher was two cycles, a network with three or more units learned the task. For a delay of three cycles, four or more units were required, and for a delay of four cycles, five or more units were needed. The learning behavior of a three-unit network with two cycles of delay between input and teacher is quite analogous to the standard feedforward case. The weights that would provide a feedforward solution become large in magnitude while the recurrent weights become small. In the example shown in Table I, the familiar solution using two single line recognizers in the first layer and an OR in the second layer was found. An interesting distribution of training trial lengths was observed. They could be easily divided into three length classes: 68% of less than 1000 cycles, average 688 cycles; 28% between 9000 and 76,000 cycles, average 30,792; and 6% greater than 200,000 cycles. The middle class is of interest because it seems to represent training trials that found what would be a local minimum for a feedforward network, but had an escape path because of the existence of the otherwise redundant recurrent connections. The choice of 200,000 cycles is arbitrary and some of these networks may eventually learn.

When an additional unit is added to the network and the delay between input and teacher is increased to three cycles, the algorithm finds a solution that incorporates an additional layer to provide the required delay. A typical example, which took 1510 cycles to learn, is shown in Table II. In this case the first layer, units 0 and 1, is a NOR and an AND. The second layer, unit 2, is also a NOR, completing the XOR. The last layer, unit 3, is just a follower of unit 2, which provides the required delay. The five-unit network with a delay of four requires still more cycles to learn. The example shown in Table III required 5618 learning trials. The weights in Table III are actually those found when an additional 100,000 training trials were run after reaching the

Table I. Weight matrix for XOR with two-cycle delay

U	B	I		R			T
0	−2.7	5.1	−5.4	0.1	0.2	−0.3	−
1	−2.8	−6.1	5.8	0.3	0.0	−0.6	−
2	−3.6	−0.1	−0.1	7.8	7.8	−0.3	+

NOTE: The columns labeled U, B, I, R and T give, respectively: unit number, bias weight, input weights, recurrent weights and teaching status where '+' indicates the existence of target values and '−' indicates that the unit never had targets. The same format is used for Tables II, III, IV, V and VIII.

correctness criteria using a threshold of 0.5. The extra training was carried out so that asymptotic values of the weights were achieved. Note that while the feedforward weights are large, some recurrent weights still have nontrivial magnitudes. This is generally the case, but its significance is not yet clear.

Table II. Weight matrix for XOR with three-cycle delay

U	B	I		R				T
0	2.5	−6.3	−6.3	−0.7	−0.9	−0.5	−0.1	−
1	−7.0	5.0	4.9	−1.6	−1.8	0.3	0.1	−
2	3.3	−0.5	−0.3	−7.6	−7.3	2.2	−0.9	−
3	−4.0	−0.3	−0.3	1.0	1.4	9.3	−1.4	+

These results show that strictly layered feedforward problems form a subset of those that can be learned by the RTRL algorithm. They also demonstrate that the netwok can learn the inherent temporal relations of the problem without any explicit temporal information.

Table III. Weight matrix for XOR with four-cycle delay

U	B	I		R					T
0	1.8	0.2	0.1	1.9	−6.3	−0.2	−6.3	0.3	−
1	−2.3	5.4	−5.3	0.2	−1.9	0.0	−1.6	−0.1	−
2	4.6	0.2	0.2	−9.2	−1.0	−0.2	−1.1	−0.4	−
3	−2.2	−5.4	5.4	0.2	−1.8	0.2	−2.0	−0.5	−
4	−4.6	−0.3	−0.3	−1.1	−0.5	14.3	−0.4	−0.8	+

Learning Internal State

Here we consider a very simple sequential recognition task which is essentially identical to Bachrach's (1988) 'bus driver' problem. The idea is for the network to recognize that two particular input events have happened in prescribed order, regardless of the number of intervening events. This example is intended to demonstrate the power of the learning algorithm and to clearly illustrate a type of task for which a simple recurrent net is clearly much better suited than any alternative approach based on the use of a feedforward net with past inputs stored in tapped delay lines.

The network for this task consists of two units, one of which is arbitrarily selected to serve as the output unit for the net. There are several input lines to the network, which we consider to be labeled with letters of the alphabet. The *a* and *b* input lines sever a special purpose, with all others serving as distractors. For purposes of illustration we describe explicitly a version with two distractors; we have successfully simulated versions with up to eight distractors. For the two-distractor case, there are a total of four input lines, labeled *a*, *b*, *c* and *d*. On any given time step, a randomly chosen input line is given a value of 1 and all others are given the value 0. Thus the input patterns can be considered to correspond to a single letter of the alphabet, encoded in a local manner. The desired output for the network can be summarized by the rule: On the time step immediately following the first *b* after an *a*, emit a 1; otherwise, emit a 0.

It should be clear that one could hand-design a solution to this problem in the following manner. One of the units would serve as a flop-flop that is set by the occurrence of *a* in the input stream and is reset by the occurrence of *b*. To be such a flip-flop, this unit should have a highly positive feedback weight to itself and a negative bias of half the magnitude, together with a strongly positive weight on the *a* input line, a strongly negative weight on the *b* line, and zero weight on all other input lines. The other unit would be the output unit, serving as a simple AND gate between the output of the flip-flop and the *b* input line.

One actual solution obtained by the real-time recurrent learning algorithm is displayed in Figure 1. The weights shown were obtained after 3000 time steps, using a learning rate of 5.0. Their initial values were chosen by uniform random generation from the interval $[-1,1]$.

This solution can be described in the following manner. First, the nonoutput unit does indeed serve as a flip-flop, but with its set and reset inverted from the more intuitive approach described earlier. The large negative weight to this unit from the *a* line causes it to take on its low value (i.e. be reset) whenever *a* occurs. The somewhat large positive weight to this unit from the *b* line causes a *b* event to contribute to its being set to its high value. In addition, there is a moderately large positive weight from the output unit to this unit. This causes the flip-flop also to tend to be set to its high value whenever the output of the network was large on the previous time step. It is not immediately clear why this should be the case; it appears to be a result of the fact that the most frequent correct value for the flip-flop is to be in its high state right after the output of the network is 1. The only time this is incorrect is when the very next input is *a*. Interestingly, we see that the strongly negative weight of the flip-flop from the *a* line will override this tendency in this case, allowing the flip-flop to behave correctly in all cases.

It is clear that the output unit cannot come on unless the *b* line is on and the flip-flop is at its low state. It is also clear that the conjunction of these conditions is sufficient to make it come on except for the presence of the strongly negative self-weight on this unit. This self-weight seems to help it avoid coming on two time steps in a row, which is, indeed, an implicit constraint on the correct operation of the network. Also, note that the moderately strong negative weights to this unit from the *a*, *c* and *d* lines help to insure that only the presence of a *b* will trigger the output unit.

Thus the solution found by the algorithm has certain essential elements of the solution that one might handcraft for this problem. At the same time, there are curious additional features that the algorithm seems to have devised, most notably the

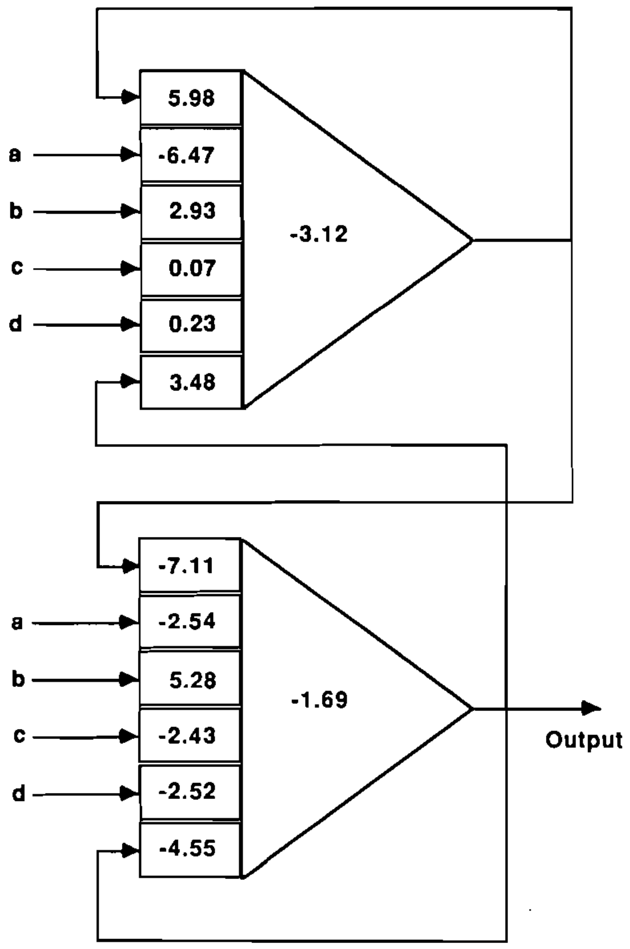


Figure 1. Network for recognizing *a* followed by *b*. Inputs *c* and *d* are distractors. The output unit was the only one with a target. The bias is shown inside the units.

moderate-to-strong weights from the output unit to itself and to the flip-flop unit. It would be interesting to investigate why this more complex solution was developed. One possibility is that during the early stages of learning, these additional strong weights are important because they help compensate for weaknesses in the immature network's operation. For example, the negative self-weight on the output unit may play a crucial role before the flip-flop comes to act in crisp bistable fashion.

It is instructive to compare the recurrent net approach taken here with a possible alternative that is sometimes proposed for dealing with time-varying input. In particular, suppose that one were to approach this task through the use of a feedforward architecture with tapped delay lines on the input. First of all, it is clear that because of the finite length of the delay lines there will always be patterns that such a network will fail to recognize properly, namely, those in which the important information spans a length of time greater than the delay lines can retain. This happens as the delay between an *a* and the first *b* following it increases. Thus an approach using a feedforward net along with tapped delay lines on the input may be computationally inadequate for certain types of tasks.

Even more interesting is to ignore for the moment the computational inadequacy of such an approach to this task[2] and consider the learning effort involved. When there are m input lines, each of which is run through a length k delay line, the effective input patterns to the network are of dimension mk. Because there must be at least this many adjustable weights in the network, it should be clear that increasing k means that more training data must be supplied to avoid spurious generalization from a limited training set. Thus while increasing k lessens the computational inadequacy, it also lessens the generalization ability. This should also be clear from the particular nature of the delay line representation; even if the net learns that it should produce a 1 when b follows a with delays of 1, 2 and 4, this experience does not carry over in any useful way to the case when the delay is 3.

In the particular recurrent network described here, there are a total of 14 adjustable weights. The simplest feedforward network using tapped delay lines of length k on the input that could perform a limited version of this task would consist of a single unit receiving input from these tapped delay lines; no hidden units are required. There would thus be $4k+1$ adjustable weights in this network. Clearly, when $k>4$, the number of weights in the recurrent version is less, and if we were to set k sufficiently high (say $k>10$) so that the performance of the feedforward version may be any reasonable approximation to that attainable by the recurrent network, the number of weights would be so much larger that many more training examples would be required for the feedforward version than for the recurrent version.

Delayed Nonmatch to Sample

In this task, the network must remember a cued input pattern and then compare it to subsequent input patterns, outputting a 0 if they match and a 1 if they do not. This taks is similar to the previous one in that an event must be remembered for an arbitrary time. However, here the memory must be able to store a pattern associated with the event, not just the fact that it occurred. We investigated a simple version of this task using a network with two input lines. One line represents the pattern and is set to 0 or 1 at random on each cycle. The other line is the cue that being set to 1 indicates that the corresponding bit on the pattern line must be remembered and used for matching until the next occurrence of the cue. The cue frequency was determined randomly, typically with a cue probability on each cycle of 0.8, so that intervals between cues could be arbitrarily long while the average intercue interval was short enough to facilitate training. A delay of one cycle was inserted after each cue to allow time for strobing in the pattern bit. A minimum teaching delay of two cycles between input and output is required for the matching computation, which is essentially an XOR, so the first teaching cycle occurred three cycles after the cue. Teaching was then continuous until the next cue; examine the traces in Table VI to clarify this task.

Analysis showed that the nonmatch-to-sample task can be carried out by four units, which is probably the minimum. Networks of four units learn the task using the unforced algorithm, but it is a difficult problem. Success in less than 200,000 cycles occurs only about 20% of the time, and the average number of cycles needed is about 90,000. With five or more units, the problem is much easier, never requiring more than 200,000 cycles and averaging about 20,000 cycles. No attempt was made to optimize the learning rate or other parameters so this performance might be improved.

Perhaps the most interesting aspect of this task is the way the remembered pattern was represented. Sometimes a local representation was used with a single bistable unit recording the pattern or its complement, while the other units did the required logic. On other runs, a distributed representation appeared in which no single unit was devoted exclusively to either storage or logic. Examples of a solution from each of

Table IV. Weight matrix for nonmatch network with local representations of stored bit

U	B	I		R				T
0	−3.6	1.7	5.9	9.6	−0.2	−9.5	−0.6	−
1	1.3	−5.8	4.0	−7.5	0.8	7.6	0.2	−
2	−9.6	7.7	6.8	2.3	−0.3	−1.9	0.0	−
3	−6.9	−0.1	−2.7	3.4	12.9	18.1	−0.4	+

these classes for networks with four units is shown in Tables IV and V. In the case of the local solution in Table IV, unit 0 stores the complement of the cued input. The network is still quite complex because of the requirements for strobing. Note in this regard that unit 0 is set to a high level on the cycle in which the cue appears no matter what the pattern bit, learning to use the cue as a strobe is probably what makes this a difficult problem. In the distributed case, none of the units stably record the pattern bit. This can be seen by examining the trace of unit activities in Table VI. When the stored bit is a 0, unit 0 is 0 and unit 1 is the complement of both the current input and the next output. Unit 2 is nearly complementary to unit 1, but intermediate values must be taken into account. When the stored bit is a 1, units 1 and 2 are 0 and unit 0 is the complement of the input and next output. This form of distributed memory is interesting because the pattern to be remembered is not simply distributed over multiple units but is used to configure the way the network responds.

Table V. Weight matrix for nonmatch network with distributed representation of stored bit

U	B	I		R				T
0	2.7	−6.9	1.7	2.3	−5.3	−4.7	−1.1	−
1	−2.9	−15.6	5.7	−1.0	6.0	3.8	0.3	−
2	−9.1	4.6	−13.8	0.9	7.3	9.3	−1.6	−
3	−2.8	−0.1	−4.8	7.6	−7.1	6.9	−1.0	+

Learning to be a Turing Machine

The previous two problems demonstrated that the RTRL algorithm could train networks to be simple state-preserving machines. To see if networks could be trained to be finite-state machines with significant power we tried to teach a network to emulate a special-purpose Turing machine (TM). There are various approaches to teaching a network to emulate a TM. We used a procedure in which the network 'looks over the shoulder' of the finite-state machine (FSM) part of the TM. The network sees the same input from the tape as the FSM and is trained to produce the FSM output. The network does not get to see the internal states of the FSM so it must invent its own. If the network learns to emulate correctly the FSM, it will be able to perform correctly on tapes of any length. Because the logic required to compute next states and outputs is, in general, nonlinearly separable, the network will require two cycles for each external cycle of the TM. That is, the TM will have to dwell on each tape location for two cycles of the network.

Table VI. Time trace activity of nonmatch networks

Local							Distributed						
P	C	A				T	P	C	A				T
1	1	0.9	0.0	0.9	0.0		0	1	0.0	0.9	0.0	0.0	
1	0	0.1	0.0	0.1	0.9		0	0	0.0	0.9	0.1	0.0	
0	0	0.0	0.9	0.0	0.0	0	1	0	0.0	0.0	0.9	0.0	0
1	0	0.1	0.0	0.1	0.9	1	0	0	0.1	0.6	0.5	0.9	1
0	0	0.0	0.8	0.0	0.0	0	1	0	0.0	0.0	0.9	0.0	0
0	1	0.9	0.9	0.0	0.7		1	0	0.0	0.0	0.9	0.9	1
0	0	0.9	0.0	0.0	0.9		0	0	0.0	0.7	0.2	0.9	1
1	0	0.9	0.0	0.5	0.0	0	1	0	0.0	0.0	0.8	0.0	0
1	1	0.9	0.0	0.9	0.9		1	0	0.0	0.0	0.9	0.9	1
0	0	0.0	0.8	0.0	0.9		0	0	0.0	0.7	0.1	0.9	1
0	0	0.0	0.8	0.0	0.9	1	0	0	0.0	0.9	0.0	0.0	0
0	0	0.0	0.8	0.0	0.9	1	1	1	0.2	0.0	0.0	0.1	
0	1	0.8	0.9	0.0	0.8		0	0	0.9	0.0	0.0	0.3	
0	0	0.9	0.0	0.0	0.9		1	0	0.0	0.0	0.0	0.9	1
1	0	0.9	0.0	0.5	0.0	0	0	0	0.8	0.0	0.0	0.0	0
0	0	0.6	0.1	0.0	0.9	1	0	0	0.9	0.0	0.0	0.9	1
0	0	0.8	0.0	0.0	0.0	0	1	0	0.0	0.0	0.0	0.9	1
							1	0	0.0	0.0	0.0	0.0	0
							1	0	0.0	0.0	0.0	0.0	0
							0	0	0.9	0.0	0.0	0.0	0

NOTE: Time increases downward. The columns labeled P, C, A and T represent the pattern, the cue, the truncated activity value and the value of the target when present. The activities shown are from trained networks that are no longer being taught.

We trained a network to emulate a TM that parses parentheses. Given a tape marked with an arbitrary length string of left and right parentheses, with a blank cell at each end of the string, the TM must decide whether or not the string consists entirely of sets of balanced parentheses. In the particular version of the problem used here (Brady, 1977), the alphabet of tape marks consisted of (,), *, and blank. The outputs of the FSM were of two kinds, a move that could be left, right or none, and an action that could be to write a *, indicate balanced, indicate unbalanced or do nothing. The FSM used has four internal states and operates according to the rules given in Table VII. The reading head of the TM was started over the parenthesis at the left end of the string in internal state 1. The TM then proceeded to move back and forth along the tape while the network was taught to produce the same outputs as the TM. For teaching purposes, the TM must function continuously, so rather than permanently halting when a balanced–unbalanced decision is made, the TM goes into a halt state for one step during which a fresh tape is prepared with the reading head again at the left end of the parenthesis string.

Generating the tapes to be used for training presented a series of problems. We did not create a fixed training set but rather generated a new tape each time one was called for. These tapes were constructed at random using the rules described below. The training set had to include tapes of arbitrary length to prevent the network from finding an idiosyncratic solution that only worked for tapes less than some maximum length. An exponential tape length distribution was used so that most of the tapes were short to allow frequent final-decision events, while still providing some long tapes to prevent solutions that only worked below a maximum length. For practical reasons there had to be a maximum tape length during training, but this was made sufficiently

Table VII. The state transition table for the FSM part of the TM used to train the network

State	Input	Next state	Function	Direction
0	00	1	00	00
0	01	1	00	00
0	10 (impossible)			
0	11 (impossible)			
1	00	3	00	10
1	01	1	00	01
1	10	2	11	10
1	11	1	00	01
2	00	0	10	00
2	01	1	11	01
2	10	2	00	10
2	11	2	00	10
3	00	0	01	00
3	01	0	10	00
3	10	3	00	10
3	11	3	00	10

Input: 00=blank; 01='('; 10=')'; 11='*'.
Function: 00=nothing; 01=balanced; 10=unbalanced; 11=write *.
Direction: 00=no movement; 01=right; 10=left.

long to prevent unwanted solutions. A maximum length of 30 worked fine. Note that the number of cycles needed to parse a tape is generally many times the length of the tape. Once an even numbered length had been chosen randomly, a balanced string of this length was generated by picking left and right parentheses at random, subject to the condition that the number of left parentheses was always greater than or equal to the number of right parentheses. One third of the time this string was used; the rest of the time it was randomly altered to produce an unbalanced tape. This was done by reversing one or more randomly selected parentheses. The probability of reversing n ($n>0$) parentheses was 0.5 to the nth power.

Once the network being trained made no errors, using a threshold of 0.5, for some long continuous period, typically 10,000 TM cycles, training was stopped and the network tested on a randomly generated set of tapes, up to 10 times longer than the maximum length used in training. A uniform rather than an exponential distribution of tape lengths was used for testing to expose the network to many longer strings than it had ever seen in training. The networks were considered to have learned if no errors occurred, using a 0.5 threshold, during at least 50,000 TM cycles.

Our preliminary analysis indicated that 15 units would suffice and that nonlinearly separable logic was required. Determining the true minimum number of units for a complex task like this, particularly when the units can take advantage of intermediate values, is daunting. In training trials, networks of 15 units always learned the task in less than 100,000 TM cycles; the average (of three cases) was about 16,500 TM cycles till no more errors occurred. The minimum size network that was observed to learn the task was 12 units. The number of TM cycles needed by 12-unit networks averaged about the same as for the 15-unit network, but there were occasional failures to find a solution. Networks with 15 units, given only one cycle of network update for each TM cycle, never learned the problem, indicating that the logic involved is indeed likely to be nonlinearly separable because this would require two layers of units and thus two cycles of processing. No attempt was made to optimize the learning rate or the other parameters involved in learning so the figures given here could probably be improved.

The connection weights for a 12-unit network that learned the parenthesis parsing problem in 12,973 TM cycles is shown in Table VIII. The pattern of weights is extremely complex, with relatively few large weights and much recurrent interconnection. It is clear that the output values are used in the representation or computation of state, because significant connections exist between the units representing output and the other units of the network. In fact most of the large connection weights involve either inputs to or outputs from the output units. The small magnitude weights cannot be ignored, as can be seen in the case of unit 7, which has no large weights but is involved in an important recognition task.

Because the network must go through two update cycles for each TM cycle, it would be expected to have a complex state behavior. This appears to be the case, as can be seen by examining the traces of activity shown in Table IX. Intermediate values are used extensively, even by the output units, during the first of the two network update cycles, on which no teaching occurs. Another interesting feature is that the activity patterns just after a state has been entered are different from the patterns that become established when a particular state exists for many cycles. This can be seen in Table IXC. Note that to run this network as a TM, it must be started in an appropriate state; that is, unlike a feedforward network, the connection matrix alone is not enough to specify function. We tried starting the network using random initial activations on all units. We were quite surprised to find the network quickly found good values and started to work correctly. It rarely took more than two network cycles for this to occur and the worst case observed was eight network cycles. This observation suggests that valid activity configurations are strong attractors. A study of the state structure of the network using cluster analysis techniques has been begun. The initial results of this study indicate that single states in the original TM are often represented by split states in the network, the largest splitting corresponding to a distinction on the basis of the previous state. Still finer splitting of states is detectable and can be associated with still earlier states. Thus the network has gratuitously developed a memory of more of its past function than is strictly required for the task.

Learning to Oscillate

An interesting class of behaviors to study with any algorithm designed to train arbitrary network dynamics is that of oscillation. Here we describe three simple network oscillation tasks that we have studied. We have used RTRL with and without teacher forcing on these tasks and we have found that only the version with teacher forcing is capable of solving these problems in general. In the following, we describe both the tasks performed and our understanding of why teacher forcing seems to be necessary for them.

The first oscillation task involves a single logistic unit whose desired behavior is to produce alternating 0s and 1s. Giving the unit a strongly negative self-weight and a bias of half the magnitude clearly leads to such oscillatory behavior. Given a sufficiently large learning rate, the teacher-forced version of real-time backpropagation can solve this problem very quickly (in less than 10 iterations). This should not be surprising since the use of teacher forcing essentially decomposes the problem into the one-layer feedforward problem of complementing a single Boolean variable. What is

Table VIII. Weight matrix for parenthesis balancing Turing machine

U	B	I							R						T
0	−2.7	0.1	0.8	−1.5	−1.0	0.1	0.2	1.6	1.0	0.1	0.6	1.3	0.6	−2.6	0.4	−
1	−3.2	0.2	5.1	−1.5	−0.4	0.2	−0.3	0.6	0.9	0.4	−1.2	−0.6	−1.6	−4.8	1.7	−
2	−1.6	1.3	3.6	0.0	0.4	−1.2	−2.7	0.7	1.5	−0.9	0.0	−1.3	0.2	−0.8	−2.3	−
3	−3.5	4.6	2.2	0.3	−0.9	−2.7	−1.4	−2.5	−0.7	−1.8	0.0	−1.7	−0.6	2.8	−4.0	−
4	1.3	−4.8	−4.5	0.7	3.1	−0.5	−1.7	3.1	3.2	−2.9	0.1	1.4	−2.5	−2.9	−1.8	−
5	−2.9	−8.1	3.8	−0.2	−1.2	−0.4	−1.3	−1.1	−2.0	−0.4	0.2	−2.0	2.1	3.8	−3.2	−
6	0.6	−4.9	−0.8	0.8	1.6	0.1	0.0	−1.2	−1.3	0.4	0.3	−1.4	−0.5	−2.9	1.2	−
7	0.2	−0.7	−1.3	0.3	−1.1	0.6	0.4	−1.5	−0.7	−0.4	−0.9	0.0	−1.2	−1.0	0.3	−
8	−2.4	1.8	−2.3	−0.3	−3.5	1.4	−3.8	−0.8	12.0	−4.8	−1.0	−1.8	5.0	2.5	−2.1	+
9	2.3	4.3	−4.8	−1.0	−2.9	−4.0	−4.8	−4.5	9.2	−1.9	−1.6	−5.2	2.3	−2.2	2.9	+
10	−0.6	5.7	−6.3	−2.3	−4.4	0.0	4.1	−6.1	−4.1	4.4	0.3	1.2	4.4	1.6	2.2	+
11	−1.0	−3.7	6.5	1.8	6.2	−1.0	−4.9	−4.9	2.3	−0.1	−1.4	−2.3	−3.8	−1.7	−6.7	+

Table IX. Time trace of the activity of a 12-unit network trained to emulate TM doing the parentheses balancing problem

S	I	A												T			
(A) TAPE:|00|01|01|10|10|00|=||(|(|)|)||																	
1	01	0.0	0.7	0.7	0.0	0.1	0.2	0.8	0.0	0.0	0.0	0.0	0.9				
	01	0.1	0.9	0.2	0.0	0.0	0.0	0.9	0.1	0.0	0.0	0.0	0.9	0	0	0	1
1	01	0.0	0.9	0.2	0.0	0.0	0.0	0.9	0.0	0.0	0.0	0.0	0.9				
	01	0.0	0.9	0.2	0.0	0.0	0.0	0.9	0.0	0.0	0.0	0.0	0.9	0	0	0	1
1	10	0.0	0.1	0.0	0.0	0.0	0.0	0.2	0.1	0.0	0.9	0.9	0.0				
	10	0.0	0.0	0.2	0.9	0.0	0.0	0.0	0.0	0.9	0.9	0.9	0.0	1	1	1	0
2	01	0.0	0.0	0.0	0.0	0.0	0.9	0.0	0.0	0.0	0.0	0.9	0.0				
	01	0.0	0.0	0.9	0.5	0.9	0.9	0.0	0.0	0.9	0.9	0.0	0.9	1	1	0	1
1	11	0.9	0.9	0.3	0.0	0.0	0.0	0.0	0.0	0.9	0.2	0.2	0.0				
	11	0.0	0.1	0.8	0.6	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.9	0	0	0	1
1	10	0.1	0.1	0.0	0.0	0.0	0.0	0.0	0.5	0.0	0.9	0.9	0.0				
	10	0.0	0.0	0.2	0.9	0.0	0.0	0.0	0.0	0.9	0.9	0.9	0.0	1	1	1	0
2	11	0.0	0.0	0.1	0.7	0.0	0.0	0.0	0.0	0.3	0.0	0.9	0.0				
	11	0.0	0.0	0.4	0.9	0.0	0.0	0.0	0.0	0.0	0.0	0.9	0.0	0	0	1	0
2	11	0.0	0.0	0.3	0.9	0.0	0.0	0.0	0.0	0.0	0.0	0.9	0.0				
	11	0.0	0.0	0.3	0.9	0.0	0.0	0.0	0.0	0.0	0.0	0.9	0.0	0	0	1	0
2	01	0.0	0.0	0.1	0.2	0.1	0.9	0.0	0.1	0.0	0.0	0.1	0.2				
	01	0.2	0.8	0.8	0.0	0.4	0.1	0.1	0.1	0.9	0.9	0.0	0.9	1	1	0	1
1	11	0.5	0.7	0.3	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.3	0.0				
	11	0.0	0.2	0.9	0.9	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.9	0	0	0	1
1	11	0.2	0.9	0.0	0.0	0.0	0.0	0.0	0.2	0.0	0.0	0.9	0.0				
	11	0.0	0.0	0.9	0.9	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.9	0	0	0	1
1	11	0.2	0.9	0.0	0.0	0.0	0.0	0.0	0.3	0.0	0.0	0.9	0.0				
	11	0.0	0.0	0.9	0.9	0.1	0.0	0.0	0.0	0.0	0.0	0.0	-0.9	0	0	0	1
1	00	0.1	0.1	0.0	0.0	0.0	0.0	0.8	0.7	0.0	0.0	0.9	0.0				
	00	0.0	0.0	0.0	0.0	0.9	0.5	0.1	0.1	0.0	0.0	0.9	0.0	0	0	1	0
3	11	0.0	0.1	0.9	0.9	0.2	0.0	0.0	0.0	0.9	0.5	0.0	0.0				
	11	0.5	0.6	0.1	0.0	0.0	0.0	0.0	0.1	0.0	0.0	0.9	0.0	0	0	1	0
3	11	0.0	0.0	0.9	0.9	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.9				
	11	0.2	0.9	0.0	0.0	0.0	0.0	0.0	0.3	0.0	0.0	0.9	0.0	0	0	1	0
3	11	0.0	0.0	0.9	0.9	0.1	0.0	0.0	0.0	0.0	0.0	0.0	0.9				
	11	0.2	0.9	0.0	0.0	0.0	0.0	0.0	0.3	0.0	0.0	0.9	0.0	0	0	1	0
3	11	0.0	0.0	0.9	0.9	0.1	0.0	0.0	0.0	0.0	0.0	0.0	0.9				
	11	0.2	0.9	0.0	0.0	0.0	0.0	0.0	0.3	0.0	0.0	0.9	0.0	0	0	1	0
3	00	0.0	0.0	0.1	0.1	0.9	0.4	0.3	0.1	0.0	0.0	0.0	0.9				
	00	0.4	0.2	0.0	0.0	0.9	0.0	0.5	0.1	0.0	0.9	0.0	0.0	0	1	0	0
0	00	0.1	0.0	0.2	0.0	0.8	0.0	0.4	0.0	0.0	0.0	0.1	0.0				
	00	0.1	0.0	0.1	0.0	0.9	0.0	0.3	0.2	0.0	0.0	0.0	0.0	0	0	0	0
(B) TAPE:|00|01|01|10|01||=||(|(|)|(||																	
1	01	0.3	0.8	0.8	0.0	0.2	0.4	0.2	0.0	0.0	0.0	0.0	0.6				
	01	0.1	0.9	0.5	0.0	0.3	0.0	0.8	0.1	0.0	0.0	0.0	0.9	0	0	0	1
1	01	0.1	0.9	0.2	0.0	0.0	0.0	0.9	0.0	0.0	0.0	0.0	0.9				
	01	0.0	0.9	0.2	0.0	0.0	0.0	0.9	0.0	0.0	0.0	0.0	0.9	0	0	0	1
1	10	0.0	0.1	0.0	0.0	0.0	0.0	0.2	0.1	0.0	0.9	0.9	0.0				
	10	0.0	0.0	0.2	0.9	0.0	0.0	0.0	0.0	0.9	0.9	0.9	0.0	1	2	1	0
2	01	0.0	0.0	0.0	0.0	0.0	0.9	0.0	0.0	0.0	0.0	0.9	0.0				
	01	0.0	0.0	0.9	0.5	0.9	0.9	0.0	0.0	0.9	0.9	0.0	0.9	1	1	0	1
1	11	0.9	0.9	0.3	0.0	0.0	0.0	0.0	0.0	0.9	0.2	0.2	0.0				
	11	0.0	0.1	0.8	0.6	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.9	0	0	0	1
1	01	0.1	0.9	0.0	0.0	0.0	0.0	0.7	0.4	0.0	0.0	0.0	0.0				
	01	0.0	0.6	0.8	0.0	0.0	0.3	0.8	0.0	0.0	0.0	0.0	0.9	0	0	0	1
1	00	0.0	0.2	0.0	0.0	0.5	0.0	0.9	0.3	0.0	0.7	0.5	0.0				
	00	0.0	0.0	0.0	0.0	0.7	0.3	0.2	0.0	0.0	0.0	0.9	0.0	0	0	1	0

Table IX—*contd.*

S	I	A												T			
(B) TAPE:| 00| 01| 01| 10| 10| 00| =| | (| (|) |) | |—*contd.*																	
3	01	0.0	0.1	0.8	0.1	0.8	0.9	0.0	0.0	0.6	0.0	0.0	0.7				
	01	0.8	0.9	0.4	0.0	0.8	0.0	0.0	0.0	0.9	0.0	0.0	0.0	1	0	0	0
0	01	0.1	0.5	0.7	0.0	0.9	0.0	0.4	0.0	0.0	0.0	0.0	0.9				
	01	0.3	0.9	0.3	0.0	0.1	0.0	0.7	0.0	0.0	0.0	0.0	0.0	0	0	0	0
(C) TAPE: || *11*| *11*| *11*| *11*| *11*| *11*| *11*|| =| | *| *| *| *| *| *| *| |																	
1	11	0.3	0.8	0.9	0.4	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0				
	11	0.0	0.7	0.7	0.2	0.0	0.0	0.0	0.1	0.0	0.0	0.0	0.9	0	0	0	1
1	11	0.1	0.9	0.3	0.0	0.0	0.0	0.0	0.1	0.0	0.0	0.2	0.0				
	11	0.0	0.5	0.9	0.8	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.9	0	0	0	1
1	11	0.1	0.9	0.0	0.0	0.0	0.0	0.0	0.2	0.0	0.0	0.9	0.0				
	11	0.0	0.0	0.9	0.9	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.9	0	0	0	1
1	11	0.2	0.9	0.0	0.0	0.0	0.0	0.0	0.3	0.0	0.0	0.9	0.0				
	11	0.0	0.0	0.9	0.9	0.1	0.0	0.0	0.0	0.0	0.0	0.0	0.9	0	0	0	1
1	11	0.2	0.9	0.0	0.0	0.0	0.0	0.0	0.3	0.0	0.0	0.9	0.0				
	11	0.0	0.0	0.9	0.9	0.1	0.0	0.0	0.0	0.0	0.0	0.0	0.9	0	0	0	1
1	11	0.2	0.9	0.0	0.0	0.0	0.0	0.0	0.3	0.0	0.0	0.9	0.0				
	11	0.0	0.0	0.9	0.9	0.1	0.0	0.0	0.0	0.0	0.0	0.0	0.9	0	0	0	1
1	11	0.2	0.9	0.0	0.0	0.0	0.0	0.0	0.3	0.0	0.0	0.9	0.0				
	11	0.0	0.0	0.9	0.9	0.1	0.0	0.0	0.0	0.0	0.0	0.0	0.9	0	0	0	1
1	00	0.1	0.1	0.0	0.0	0.0	0.0	0.8	0.7	0.0	0.0	0.9	0.0				
	00	0.0	0.0	0.0	0.0	0.9	0.5	0.1	0.1	0.0	0.0	0.9	0.0	0	0	1	0
3	11	0.0	0.1	0.9	0.9	0.2	0.0	0.0	0.0	0.9	0.5	0.0	0.0				
	11	0.5	0.6	0.1	0.0	0.0	0.0	0.0	0.1	0.0	0.0	0.9	0.0	0	0	1	0
3	11	0.0	0.0	0.9	0.9	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.9				
	11	0.2	0.9	0.0	0.0	0.0	0.0	0.0	0.3	0.0	0.0	0.9	0.0	0	0	1	0
3	11	0.0	0.0	0.9	0.9	0.1	0.0	0.0	0.0	0.0	0.0	0.0	0.9				
	11	0.2	0.9	0.0	0.0	0.0	0.0	0.0	0.3	0.0	0.0	0.9	0.0	0	0	1	0
3	11	0.0	0.0	0.9	0.9	0.1	0.0	0.0	0.0	0.0	0.0	0.0	0.9				
	11	0.2	0.9	0.0	0.0	0.0	0.0	0.0	0.3	0.0	0.0	0.9	0.0	0	0	1	0
3	11	0.0	0.0	0.9	0.9	0.1	0.0	0.0	0.0	0.0	0.0	0.0	0.9				
	11	0.2	0.9	0.0	0.0	0.0	0.0	0.0	0.3	0.0	0.0	0.9	0.0	0	0	1	0
3	11	0.0	0.0	0.9	0.9	0.1	0.0	0.0	0.0	0.0	0.0	0.0	0.9				
	11	0.2	0.9	0.0	0.0	0.0	0.0	0.0	0.3	0.0	0.0	0.9	0.0	0	0	1	0
3	11	0.0	0.0	0.9	0.9	0.1	0.0	0.0	0.0	0.0	0.0	0.0	0.9				
	11	0.2	0.9	0.0	0.0	0.0	0.0	0.0	0.3	0.0	0.0	0.9	0.0	0	0	1	0
3	00	0.0	0.0	0.1	0.1	0.9	0.4	0.3	0.1	0.0	0.0	0.0	0.9				
	00	0.4	0.2	0.0	0.0	0.9	0.0	0.5	0.1	0.0	0.9	0.0	0.0	0	1	0	0
0	00	0.1	0.0	0.2	0.0	0.8	0.0	0.4	0.0	0.0	0.0	0.1	0.0				
	00	0.1	0.0	0.1	0.0	0.9	0.0	0.3	0.2	0.0	0.0	0.0	0.0	0	0	0	0

NOTE: The tape being processed is shown in coded and uncoded form at the top of the table. The columns labeled S, I, A and T represent the state of the FSM, the current tape symbol, the truncated value of the activities one cycle after seeing the inputs, and the value of the targets. The activities shown are from trained networks that are no longer being taught. The values of state and target are presented for reference. In each trace, the reading head of the TM is placed on the left-most nonblank cell of the tape before the first cycle of network update. Time increases downward.

interesting is how this compares with the behavior of the unforced version of the algorithm. Starting with small random weights, the unforced version will never solve this problem. This simulation experience is confirmed by the complete mathematical analysis one can perform for this particular case. While we omit discussion of the details of this analysis, we can describe the overall conclusion. The problem is a fundamental one in which the weights need to be adjusted across a bifurcation boundary, but the gradient itself cannot yield the necessary information (because it is zero or very close to zero).[3] However, if one is free to adjust both weights and initial conditions this problem disappears, at least in some cases. Something like this appears to be at the heart of the success of the use of teacher forcing: By using desired values to partially 'reset' the state of the net, one is helping to control the initial conditions for the subsequent dynamics.

For the particular case of trying to make a single unit oscillate, giving the weights small, random values guarantees that the network dynamics is that of settling to a stable state. When this is the case, the only way to move the weights into a region where oscillation occurs is by moving the weights across a bifurcation boundary, and thus the problem just described arises.

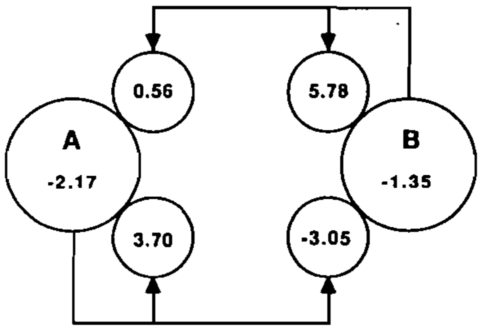
Also instructive in this regard is the following. Suppose we set the weights appropriately so that the unit will oscillate as desired, but once the network begins running we give it a teacher signal that is exactly out of phase with its actual operation. In this case, the unforced version of the algorithm will actually move the weights away from their correct values, towards values that try to make the unit settle to a compromise value of 0.5. In contrast, the forced version causes the weights to move in the wrong direction on only a single time step; after that, the unit locks into phase with the teacher signal and there is essentially no further need for weight adjustment.

The second Boolean oscillation task we have studied involves a pair of logistic units, where the desired behavior is for one of the units (chosen arbitrarily) to produce the sequence 0, 0, 1, 1, 0, 0, 1, 1, and so on. The behaviors of the forced and unforced versions of the algorithm on this task show a close correspondence with their behaviors on the single unit oscillator task. Using teacher forcing, the network learns to perform this task (within 0.1 of each desired value) after about 100 iterations with a learning rate of 5.0 and initial weights chosen uniformly from $[-1,1]$. Without teacher forcing, the network essentially never learns the correct behavior, once again because it starts out as a settling net. Likewise, occasional shifts in the phase of the teacher signal cause essentially no problems for the teacher-forced version of the algorithm but wreak havoc on the unforced version, even when the weights are initially correct.

Sine Wave Oscillation

It is straightforward to show that a pair of appropriately connected linear units can produce sine wave oscillation. It is not so clear whether, or how well, a pair of logistic units can produce sine wave oscillation. Using the forced version of the algorithm we tried to teach logistic units to oscillate sinusoidally. Stable, sine-like oscillation could be obtained for sine frequencies above about 25 network cycles (i.e. ticks) per cycle with training of 30,000 ticks or less. Much lower sine wave frequencies could not be learned in reasonable teaching times. Typically 3000 to 4000 ticks were required before stable oscillation was observed. The start of stable oscillation, unsupported by continued teaching, was an abrupt event. Before stable oscillation is established the network damps quickly to constant values.

An example of the kind of oscillation obtained with a pair of logistic units taught using a sine wave with a frequency of 25 ticks per cycle and minimum and maximum values of 0.0 and 1.0 is shown in Figure 2. Unit B, which received forced teaching, has nearly the correct amplitude but a distorted wave form. The other unit produces an almost perfect sine wave but with half the trained amplitude. The frequency of the free-running logistic network is about 10% lower than the trained frequency. These results are typical but we have not studied a wide range of learning rates, amplitudes, or frequencies.



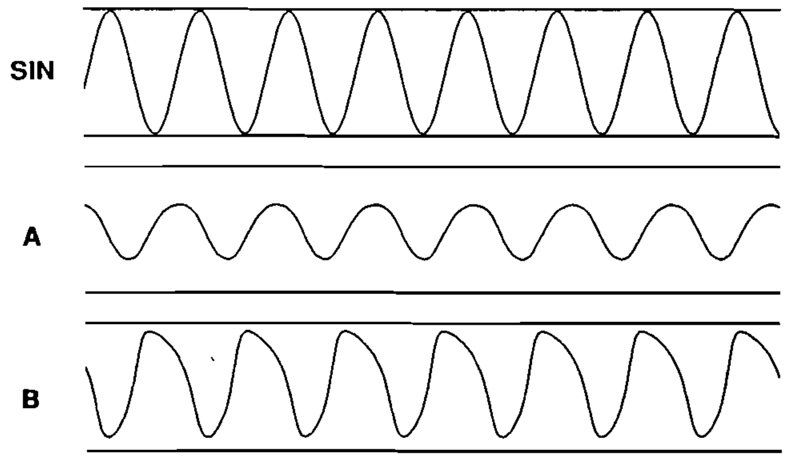


Figure 2. Sine wave network. Unit B was trained to a 25 tick per cycle sine wave. The graphs show the training sine wave and the outputs of units A and B in the absence of a teacher after stable oscillation had been established.

Discussion

Our primary goal here has been to derive a learning algorithm to train completely recurrent, continually updated networks to learn temporal tasks. Our emphasis has been on using inform starting configurations that contain no *a priori* information about the temporal nature of the task. In most cases we have used statistically derived training sets that have not been extensively optimized to promote learning. The results of the simulation experiments presented here demonstrate that the algorithm has sufficient generality and power to work under these conditions. It is likely that when knowledge of the temporal nature of the task is incorporated into the starting networks, still better results will be possible.

The algorithm we have described here is nonlocal in the sense that, for learning, each weight must have access to both the complete recurrent weight matrix **W** and the whole error vector **e**. This makes it unlikely that this algorithm, in its current form, can serve as the basis for learning in actual neurophysiological networks. The algorithm is, however, inherently quite parallel so that computation speed would benefit greatly from parallel hardware. As it stands, the dependence of computation time on the product of the number of units squared and the number of weights significantly limits the size of the networks that can be efficiently studied in serial computers.

The solutions found by the algorithm are often dauntingly obscure, particularly for complex tasks involving internal state. This observation is already familiar in work with feedforward networks. This obscurity has often limited our ability to analyze the solutions in sufficient detail. In the simpler cases, where we can discern what is going on, an interesting kind of distributed representation can be observed. Rather than only remembering a pattern in a fixed local or distributed group of units, the networks sometimes incorporate the data that must be remembered into their functioning in such a way that there is no stable pattern that represents it. This gives rise to dynamic internal representations that are, in a sense, distributed in both space and time. The existence of such patterns in the brain could greatly complicate the analysis of the representational mechanisms used there.

Notes

1. This derivation has been presented in shorter form in Williams & Zipser (1989).
2. After all, one could argue that situations in which the first *b* does not occur until, say, 20 or more time steps after an *a* are so rare that such a network, which bases its computation on the last 20 inputs, could have a very low error responsibility.
3. We emphasize that this is a problem for any gradient algorithm for adjusting the weights, not just RTRL. In particular, it occurs with the backpropagation-through-time algorithm as well if the state of the network at the start of an epoch is allowed to be equal to what it was at the end of the previous epoch, or if the teacher signal is provided only after the network has reached its steady state behavior.

References

Almeida, L.B. (1987) A learning rule for asynchronous perceptrons with feedback in a combinatorial environment. *Proceedings of the IEEE First International Conference on Neural Networks*, Vol. II, pp. 609–618.

Bachrach, J. (1988) *Learning to represent state.* Unpublished master's thesis. University of Massachusetts, Amherst, MA.

Brady, J.M. (1977) *The Theory of Computer Science: a Programming Approach.* London: Chapman & Hall.

Elman, J.L. (1988) *Finding structure in time* (CRL Tech. Rep. 8801). La Jolla, CA: University of California, San Diego, Center for Research in Language.

Jordan, M.I. (1986) Attractor dynamics and parallelism in a connectionist sequential machine. *Proceedings of the Eighth Annual Conference of the Cognitive Science Society*, pp. 531–546.

Mozer, M.C. (1988) *A focused back-propagation algorithm for temporal pattern recognition* (Tech. Rep.). University of Toronto, Departments of Psychology and Computer Science.

Pearlmutter, B.A. (1988) *Learning state space trajectories in recurrent neural networks: a preliminary report* (Tech. Rep. AIP-54). Pittsburgh, PA: Carnegie Mellon University, Department of Computer Science.

Pineda, F.J. (1988) Dynamics and architecture for neural computation. *Journal of Complexity*, **4**, 216–245.

Robinson, A.J. & Fallside, F. (1987) *The utility driven dynamic error propagation network* (Tech. Rep. CUED/F-INFENG/TR.1). Cambridge, UK: Cambridge University Engineering Department.

Rohwer, R. & Forrest, B. (1987) Training time-dependence in neural networks. *Proceedings of the IEEE First International Conference on Neural Networks.*

Rumelhart, D.E., Hinton, G.E. & Williams, R.J. (1986) Learning internal representations by error propagation. In D. E. Rumelhart, J. L. McClelland and the PDP Research Group (Eds) *Parallel*

Distributed Processing: Explorations in the Microstructure of Cognition. Vol. 1. Foundations. Cambridge, MA: MIT Press/Bradford Books.

Servan-Schreiber, D., Cleeremans, A. & McClelland, J.L. (1988) *Encoding sequential structure in simple recurrent networks* (Tech. Rep. CMU-CS-88-183). Carnegie Mellon University, Department of Computer Science.

Stornetta, W.S., Hogg, T. & Huberman, B.A. (1977) A dynamical approach to temporal pattern processing. *Proceedings of the IEEE Conference on Neural Information Processing Systems*, pp. 750–759.

Widrow, B. & Stearns, S.D. (1985) *Adaptive Signal Processing.* Englewood Cliffs, NJ: Prentice-Hall.

Williams, R.J. & Zipser, D. (1989) A learning algorithm for continually running fully recurrent neural networks. *Neural Computation*, **1**, 270–280.