13 Gradient-Based Learning Algorithms for Recurrent Networks and Their Computational Complexity

Ronald J. Williams
College of Computer Science, Northeastern University

David Zipser
Department of Cognitive Science, University of California, San Diego

INTRODUCTION

Learning in Recurrent Networks

Connectionist networks having feedback connections are interesting for a number of reasons. Biological neural networks are highly recurrently connected, and many authors have studied recurrent network models of various types of perceptual and memory processes. The general property making such networks interesting and potentially useful is that they manifest highly nonlinear dynamical behavior. One such type of dynamical behavior that has received much attention is that of settling to a fixed stable state, but probably of greater importance both biologically and from an engineering viewpoint are time-varying behaviors.

Here we consider algorithms for training recurrent networks to perform *temporal supervised learning tasks*, in which the specification of desired behavior is in the form of specific examples of input and desired output trajectories. One example of such a task is sequence classification, where the input is the sequence to be classified and the desired output is the correct classification, which is to be produced at the end of the sequence, as in some of the work reported by Mozer (1989; Chapter 5, this volume). Another example is sequence production, as studied by Jordan (1986), in which the input is a constant pattern and the corresponding desired output is a time-varying sequence. More generally, both the input and desired output may be time-varying, as in the prediction problems investigated by Cleeremans, Servan-Screiber, and McClelland (1989; Chapter 9, this volume) and the control problems studied by Nguyen and Widrow (Chapter 6, this volume). While limited forms of time-varying behaviors can be handled by using feedforward networks and tapped delay lines (e.g., Waibel et al., 1987),

recurrent networks offer a much richer set of possibilities for representing the necessary internal state. Because their internal state representation is adaptive rather than fixed, they can form delay line structures when necessary while also being able to create flip-flops or other memory structures capable of preserving a state over potentially unbounded periods of time. This point has been emphasized in (Williams, 1990) and similar arguments have been made by Mozer (1988; Chapter 5, this volume).

There are a number of possible reasons to pursue the development of learning algorithms for recurrent networks, and these may involve a variety of possible constraints on the algorithms one might be willing to consider. For example, one might be interested in understanding how biological neural networks learn to store and reproduce temporal sequences, which requires that the algorithm used be "biologically plausible," implying that the specific implementation of the algorithm map onto known neural circuitry in a reasonable way. Or, one might seek an algorithm which does not necessarily conform to known biological constraints but is at least implementable in entirely local fashion, requiring essentially no additional connectivity beyond that already present in the network to be trained. A still weaker constraint on the algorithm is that it allow a reasonable implementation in parallel hardware, even if that requires certain additional mechanisms within the overall system beyond those present in the network to be trained. These last two constraints are of some importance for attempts to create special-purpose hardware realizations of networks with on-line adaptation capabilities. Another possible constraint on the algorithm is that it be efficient when implemented in serial hardware. This constraint may be important for off-line development of networks which are useful for certain engineering applications, and it can also be important for cognitive modeling studies which are designed to examine the internal representations necessary to perform certain sequential tasks.

Overview of This Chapter

In this chapter we describe several gradient-based approaches to training a recurrent network to perform a desired sequential behavior in response to input. In characterizing these approaches as "gradient-based" we mean that at least part of the learning algorithm involves computing the gradient of some form of performance measure for the network in weight space, either exactly or approximately, with this result then used in some appropriate fashion to determine the weight changes. For the type of task investigated here, the performance measure is a simple measure of error between actual and desired output.

Because we deal here only with gradient-based learning algorithms, our primary focus will be on techniques for computing this exact or approximate gradient information. It is to be understood that there may be various alternative ways to use this gradient information in a particular learning algorithm, including simple proportional descent along the error gradient or the use of "momentum" or other more sophisticated acceleration techniques.

We discuss several approaches to performing the desired gradient computation, some based on the familiar backpropagation algorithm and some involving other ideas. Part of the intent of this chapter is to discuss the relationship between these various alternative approaches to gradient computation in recurrent networks. We begin by developing exact gradient computation algorithms, but later we note how they give rise to useful approximation strategies having more desirable computational features. For all these approaches to exact or approximate gradient computation we also provide an analysis of their computational requirements. The reader interested in performing digital computer simulation experiments of these various algorithms may find these analyses particularly helpful. In addition, we note some special architectures which readily lend themselves to specific hybrid strategies giving rise to conceptually and/or computationally simpler algorithms for exact gradient computation. Additional topics discussed are *teacher forcing*, a useful adjunct to all of the techniques discussed, and some experimental comparisons of the performance of some of the algorithms.

CONTINUAL VERSUS EPOCHWISE OPERATION

It is important to distinguish between two approaches to operating (and training) a recurrent network. In *epochwise operation* the network is run from some particular starting start until some stopping time is reached, after which the network is reset to its starting state for the next epoch. It is not essential that the state at the beginning of each epoch be the same; the important feature of this approach is that the state at the start of the new epoch is unrelated to the state at the end of the previous epoch. Because of this, an epoch boundary serves as a barrier across which "credit assignment" should not pass; erection of these barriers rules out any possibility that activity from one epoch might be relevant to producing the desired behavior for any later epoch.[1]

Note that an epoch in the sense used here is only loosely related to the corresponding notion sometimes used in the context of *batch training*, as distinguished from *incremental training*, of feedforward networks. The key issue in that case is when the weight updates are performed. In the batch approach to training a feedforward network, weight changes are performed only after a complete cycle of pattern presentations; in the incremental approach, weight

[1]Interestingly, these functions can be dissociated from one another. For example, one might imagine imposing no state reset at any time, while still allowing a learning algorithm to take advantage of occasional information provided by a teacher which effectively tells the learning system that no state reached prior to some particular time is relevant to producing correct performance at subsequent times.

changes are made after each pattern is presented. In the current terminology, a single epoch for the recurrent network corresponds to one training pattern for a feedforward network, so a network which operates epochwise may be trained using an incremental approach, in which weight changes are made at the end of each epoch, or a batch approach, in which weight changes are performed after several epochs.

In contrast, a network is considered to *operate continually* if neither "manual" state resets nor other such artificial credit-assignment barriers are available to a trainer of the network. The concept of a continually operating network would appear to be more appropriate for situations when on-line learning is required, although this introduces some subtleties when attempting to formalize the overall objective of learning. These subtleties are not present in the epochwise case because one can imagine that each epoch involves a potentially repeatable event, like the presentation of a single pattern to a feedforward network, with these individual events considered independent of one another. An additional subtlety in the continual operation case is due to the need to make weight changes while the network runs. Unlike the epochwise case, the continual operation case offers no convenient times at which to imagine beginning anew with different weight values.

As an example of the use of this distinction, consider the task of training a network to match the input-output behavior of a given finite state machine through observation of this behavior. A number of the training algorithms to be described in this chapter have been used for just such tasks. If one assumes that there is a distinguished start state and a set of distinguished final states in the machine to be emulated by the network, then it seems reasonable to train the network in an epochwise fashion. In this approach, whenever the machine being emulated is restarted in its start state after arriving in a final state, the network is reset to its start state as well. However, one might also consider trying to emulate finite state machines having no such distinguished states, in which case letting the network operate continually is more appropriate. In general, resetting the network to match a particular state of the machine being emulated is an additional mechanism for giving training information to the network, less informative than the extreme of giving complete state information (which would make the task easy), but more informative than giving only input-output information. In this case the training information helps learning during the time period shortly after the reset. There is also another difference between the continual operation case and the epochwise case which may be important. If transitions are added from the final states to the start state in the finite state machine emulation task, an epochwise task is turned into a continual-operation task. Note that a network trained to perform the epochwise version of the task is never required to make the transition to this distinguished state on its own, so one would not expect it to perform the same on the continual-operation version of the task as a network actually trained on that version. In particular, it may not be able to "reset itself" when appropriate.

While we include discussion of learning algorithms for networks which operate epochwise, much of our emphasis here is on algorithms especially appropriate for training continually operating networks.

FORMAL ASSUMPTIONS AND DEFINITIONS

Network Architecture and Dynamics

All the algorithms presented in this chapter are based on the assumption that the network consists entirely of semilinear units. More general formulations of these algorithms are possible, and it is straightforward to use the same approach to deriving them. Another assumption we make here is the use of discrete time. There are continuous-time analogs of all the approaches we discuss, some of which are straightforward to obtain and others of which involve more work.

Let the network have n units, with m external input lines.[2] Let $\mathbf{y}(t)$ denote the n-tuple of output of the units in the network at time t, and let $\mathbf{x}^{\text{net}}(t)$ denote the m-tuple of external input signals to the network at time t. We also define $\mathbf{x}(t)$ to be the $(m + n)$-tuple obtained by concatenating $\mathbf{x}^{\text{net}}(t)$ and $\mathbf{y}(t)$ in some convenient fashion. To distinguish the components of $\mathbf{x}$ representing unit outputs from those representing external input values where necessary, let U denote the set of indices k such that x_k, the kth component of $\mathbf{x}$, is the output of a unit in the network, and let I denote the set of indices k for which x_k is an external input. Furthermore, we assume that the indices on $\mathbf{y}$ and $\mathbf{x}^{\text{net}}$ are chosen to correspond to those of $\mathbf{x}$, so that

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

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. Note that one consequence of this notational convention is that $x_k(t)$ and $y_k(t)$ are two different names for the same quantity when $k \in U$. The general philosophy behind our use of this notation is that variables symbolized by x represent input and variables symbolized by y represent output. Since the output of a unit may also serve as input to itself and other units, we will consistently use x_k when its role as input is being emphasized and y_k when its role as output is being emphasized. Furthermore, this naming convention is intended to apply both at the level of individual units and at the level of the entire network. Thus, from the point of view of the network, its input is denoted $\mathbf{x}^{\text{net}}$ and, had it been it necessary for this exposition, we would have

[2] What we call *input lines* others have chosen to call *input units*. We avoid this terminology because we believe that they should not be regarded as units since they perform no computation. Another reasonable alternative might be to call them *input terminals*.

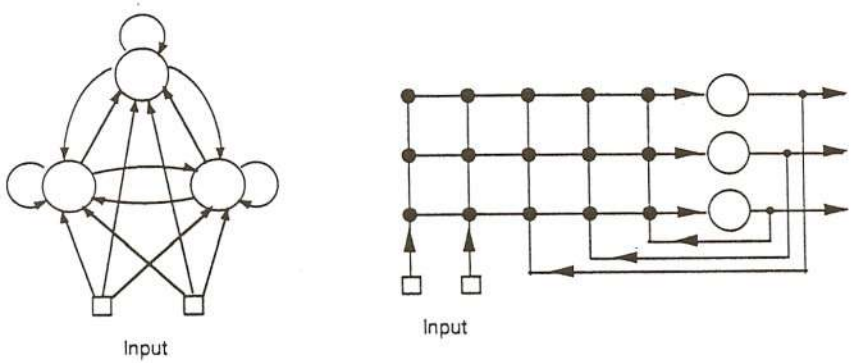

Figure 1. Two representations of a completely connected recurrent network having three units and two input lines. One input line might serve as a bias and carry the constant value 1. Any subset of these three units may serve as output units for the net, with the remaining units treated as hidden units. The 3 × 5 weight matrix for this network corresponds to the array of heavy dots in the version on the right.

denoted its output by $\mathbf{y}^{\text{net}}$ and chosen its indexing to be consistent with that of $\mathbf{y}$ and $\mathbf{x}$.

Let $\mathbf{W}$ 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 to 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 matrix contains as its ith element the bias for unit i. In general, our naming convention dictates that we regard the weight w_{ij} as having x_j as its "presynaptic" signal and y_i as its "postsynaptic" signal. Figure 1 shows a fully connected network having 3 units, 2 input lines, and a 3 × 5 weight matrix.

For the semilinear units used here it is convenient to also introduce for each k the intermediate variable $s_k(t)$, which represents the net input to the kth unit at time t. Its value at time $t + 1$ is computed in terms of both the state of and input to the network at time t by

$$s_k(t + 1) = \sum_{l \in U} w_{kl} y_l(t) + \sum_{l \in I} w_{kl} x_l^{\text{net}}(t) = \sum_{l \in U \cup I} w_{kl} x_l(t). \tag{2}$$

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 $\mathbf{x}$ and the corresponding indexing convention. Hereafter, we use only the latter form, thereby avoiding any explicit reference to $\mathbf{x}^{\text{net}}$ or its individual coordinates.

The output of such a unit at time $t + 1$ is then expressed in terms of the net input by

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

where f_k is the unit's squashing function. Throughout much of this chapter we make no particular assumption abut the nature of the squashing functions used by the various units in the network, except that we require them to be differentiable. In those cases where a specific assumption about these squashing functions is required, it will be assumed that all units use the logistic function.

Thus the system of Equations 2 and 3, where k ranges over U, constitute the entire discrete-time dynamics of the network, where the x_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 $t + 1$. We are thus treating every connection as having a one-time-step delay. It is not difficult to extend the analyses presented here situations where different connections have different delays. Later we make some observations concerning the specific case when some of the connections have no delay.

While the derivations we give throughout this chapter conform to the particular discrete-time dynamics given by Equations 2 and 3, it is worthwhile here to call attention to the use of alternative formulations obtained specifically from application of Euler discretization to continuous-time networks. For example, if we begin with the dynamical equations[3]

$$\tau_k \dot{y}_k(t) = -y_k(t) + f_k(s_k(t)), \tag{4}$$

where $s_k(t)$ is defined by Equation 2 as before, then discretizing with a sampling interval of Δt is easily shown to give rise to the discrete update equations

$$y_k(t + \Delta t) = \left(1 - \frac{\Delta t}{\tau_k}\right) y_k(t) + \frac{\Delta t}{\tau_k} f_k(s_k(t)). \tag{5}$$

Defining $\beta_k = \Delta t / \tau_k$ and altering the time scale so that $\Delta t = 1$, we then obtain the equations

$$y_k(t + 1) = (1 - \beta_k) y_k(t) + \beta_k f_k(s_k(t)), \tag{6}$$

and it is then clear that Equation 3 represents the special case when $\beta_k = 1$. It is straightforward to derive algorithms like those given throughout this chapter for these more general alternative forms of discrete-time dynamics if desired. The potential advantage of using such dynamics where $\beta_k \ll 1$ is that certain classes of task may be more readily learned by such systems, as has been observed by Tsung (1990).[4] The particular advantage possessed by such systems is that the

[3]Note that these particular equations are of essentially the same form as those considered by Pineda (Chapter 4, this volume), except that we assume that external input to the unit must pass through the squashing function.

[4]In fact, there is a strong similarity between Equation 6 and the form of recurrence Mozer (chapter 5, this volume) has used; some of his observations concerning the potential advantages of his focused architecture could be considered to apply more generally to any use of recurrence more like that found in continous-time systems.

gradient computation used in the learning algorithms to be described here falls off more gradually over time, which means that "credit assignment" is more readily spread over longer time spans than when $\beta = 1$.

Network Performance Measure

Assume that the task to be performed by the network is a *sequential supervised learning* task, meaning that certain of the units' output values are to match specified target values (which we also call *teacher signals*) at specified times. Once again, this is not the most general problem formulation to which these approaches apply, but it is general enough for our purposes here.

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 at time t. Then define a time-varying n-tuple $\mathbf{e}$ by

$$e_k(t) = \begin{cases} d_k(t) - y_k(t) & \text{if } k \in T(t) \\ 0 & \text{otherwise.} \end{cases} \quad (7)$$

Note that this formulation allows for the possibility that target values are specified for different units at different times. The set of units considered to be "visible" can thus be time varying. Now let

$$J(t) = -\frac{1}{2} \sum_{k \in U} [e_k(t)]^2 \quad (8)$$

denote the negative of the overall network error at time t. A natural objective of learning might be to maximize[5] the negative of the total error

$$J^{\text{total}}(t',t) = \sum_{\tau=t'+1}^{t} J(\tau) \quad (9)$$

over some appropriate time period $(t', t]$. The gradient of this quantity in weight space is, of course,

$$\nabla_{\mathbf{w}} J^{\text{total}}(t',t) = \sum_{\tau=t'+1}^{t} \nabla_{\mathbf{w}} J(\tau). \quad (10)$$

In general, we let t_0 denote some starting time at which the network has its state initialized. For a continually running network there are no other times at which the state is ever reinitialized in this way, but with epochwise training there will be other such times $t_1, t_2, t_3, \ldots$ marking epoch boundaries. Alternatively,

[5]The problem of minimizing error is treated here as a maximization problem because it eliminates the need for annoying minus signs in many of the subsequent formulas.

one might consider time to begin anew at t_0 whenever the state is reinitialized in a epochwise approach. Throughout this chapter, whether considering the case of a network operating epochwise or continually, we let t_0 denote the last time at which a state reset occurred. In the epochwise case we also use t_1 to indicate the end of the current epoch.

We now introduce some specific definitions designed to pin down the relationship between the various notions concerning continual and epochwise operation on the one hand and the use of gradient computation on the other. For purposes of this chapter, we make the following definitions. An *exact gradient computation algorithm* is one having the property that at every time step τ during which the network runs there is an interval $(t', t]$ containing τ such that the algorithm computes $\nabla_{\mathbf{w}} J^{\text{total}}(t', t)$ at time t, under the assumption that the network weights are fixed. Any such exact gradient algorithm is called *epochwise* if it is applied to a network operating in epochwise fashion and it computes $\nabla_{\mathbf{w}} J^{\text{total}}(t_0, t_1)$ at t_1, the end of the epoch. It is called *real time* if it computes $\nabla_{\mathbf{w}} J(t)$ at each time t. If, instead, an algorithm computes what is considered only an approximation to $\nabla_{\mathbf{w}} J^{\text{total}}(t', t)$ at time t (under the assumption that the weights are fixed) it will be regarded as an *approximate gradient computation algorithm*.

It must be emphasized that an "exact" gradient algorithm in this sense is only exact if the weights are truly fixed. Such an algorithm may not compute the exact gradient for the current setting of the weights if the weights are allowed to vary. When such an exact gradient algorithm is used to adjust the weights in a continually operating network, what it computes will thus generally be only an approximation to the desired true gradient. Later we discuss this issue further.

A *gradient-based learning algorithm* is a learning algorithm which bases its weight changes on the result of an exact or approximate gradient computation algorithm. The complete specification of such a learning algorithm must include not only how it computes such gradient information, but also how it determines the weight changes from the gradient and when these weight changes are made. Since the main focus of this chapter is on the gradient computation itself, we will generally remain noncommittal about both of these details for the learning algorithms we discuss, occasionally even blurring the distinction between the learning algorithm itself and the gradient computation portion of the algorithm.

One natural way to make the weight changes is along a constant positive multiple of the performance measure gradient, so that

$$\Delta w_{ij} = \eta \frac{\partial J^{\text{total}}(t',t)}{\partial w_{ij}}, \quad (11)$$

for each i and j, where η is a positive learning rate parameter. In those cases where we describe the empirical behavior of particular gradient-based learning algorithms this is the precise weight-change strategy used.

With regard to the timing of the weight changes, it is natural with a continually operating network to adjust the weights at the point when the appropriate

gradient has been computed, but, as already noted, for the epochwise case it may be appropriate to make weight adjustments only after multiple epochs. For purposes of this chapter, we consider an *epochwise learning algorithm* to be any learning algorithm appropriate for networks which operate epochwise and which has the property that weight updates are performed only at epoch boundaries, while a *real-time learning algorithm* is one in which weight updates can be performed at all time steps.

It is trivial to observe that any algorithm capable of computing the instantaneous performance gradient $\nabla_{\mathbf{W}} J(t)$ could be used in an epochwise manner by simply accumulating these values until time t_1 but we will discover below that this is not an efficient strategy.

Notation and Assumptions Used for Complexity Analyses

Here we summarize notation to be used in analyses of the computational complexity of the various algorithms to be discussed in this chapter. For completeness, we include some introduced earlier.

n = number of units
m = number of input lines
w_U = number of nonzero weights between units
w_A = number of adjustable weights
Δ_T = number of time steps between target presentations
n_T = average number of units given a target per time step and
L = total number of time steps

We also use the standard notation for describing the order of magnitude of the computational complexity of algorithms, where $O(\varphi(n))$ is the set of positive-integer-valued functions of n which are less than or equal to some constant positive multiple of $\varphi(n)$, $\Omega(\varphi(n))$ is the set of positive-integer-valued functions of n which are greater than or equal to some constant positive multiple of $\varphi(n)$, and $\Theta(\varphi(n)) = O(\varphi(n)) \cap \Omega(\varphi(n))$. Thus O is used to describe an upper bound on the order of magnitude of a quantity of interest, Ω is used to describe a lower bound on this order of magnitude, and Θ is used to describe the exact order of magnitude.

In all cases, we analyze the space complexity in terms of the number of real numbers stored and the time complexity in terms of the number of arithmetic operations required. For all the algorithms to be analyzed, the dominant computation is a form of inner product, so the operations counted are additions and multiplications, in roughly equal numbers. For the analyses presented here we ignore the computational effort required to run the dynamics of the network

(which, of course, must be borne regardless of the learning algorithm used), and we also ignore any additional computational effort required to actually update the weights according to the learning algorithm. Our measurement of the complexity is based solely on the computational requirements of the particular exact or approximate gradient computation method used by any such learning algorithm.

For any fixed n, the worst case for all the algorithms discussed here occurs when the network is fully connected and all weights are adaptable. In this case, $w_A = n(n + m)$ and $w_U = n^2$. In all cases below where we perform an analysis of the worst case behavior we restrict attention to classes of networks for which $m \in O(n)$ just to make the resulting formulas a little simpler. This assumption applies, for example, to the situation where a variety of networks are to be taught to perform a particular fixed task, in which case $m \in O(1)$, and it also applies whenever we might imagine increasing the number of units in a network in proportion to the size of the input pattern representation chosen. For our worst-case analyses, then, we will use the fact that w_A and w_U are both in $\Theta(n^2)$.

Note that expressing the complexity in terms of the quantities w_A and w_U assumes that the details of the particular algorithm are designed to take advantage of the limited connectivity through the use of such techniques as sparse matrix storage and manipulation. Alternatively, one could regard multiplication by zero and addition of zero as no-cost operations. A similar remark applies to the use of Δ_T and n_T. All the complexity results derived throughout this chapter are summarized in Tables 1 and 2, which appear near the end of this chapter.

BACK PROPAGATION THROUGH TIME

Here we describe an approach to computing exact error gradient information in recurrent networks based on an extension of the standard back-propagation algorithm for feedforward nets. Various forms of this algorithm have been derived by Werbos (1974), Rumelhart, Hinton, and Williams (1986), and Robinson and Fallside (1987), and continuous-time versions have been derived by Pearlmutter (1989) and Sato (1990a; 1990b). This approach is called *back propagation through time* (BPTT) for reasons that should become clear.

Unrolling a Network

Let $\mathcal{N}$ denote the network which is to be trained to perform a desired sequential behavior. Recall that we assume that $\mathcal{N}$ has n units and that it is to run from time t_0 up through some time t (where we take $t = t_1$ if we are considering an epochwise approach). As described by Rumelhart et al. (1986), we may "unroll" this network in time to obtain a feedforward network $\mathcal{N}^*$ which has a layer for each time step in the interval $[t_0, t]$ and n units in each layer. Each unit in $\mathcal{N}$ has a

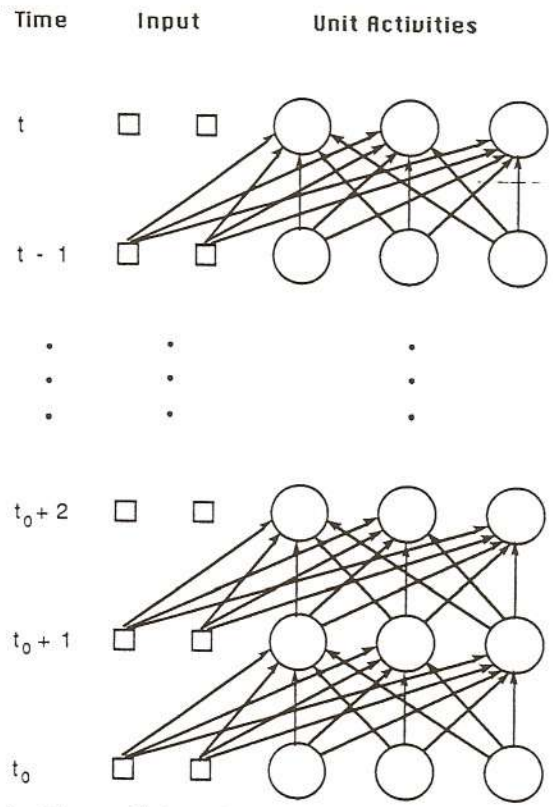

Figure 2. The unrolled version of the network shown in Figure 1 as it operates from time t_0 through time t. Each connection in the network is assumed to have a delay of one time step.

copy in each layer of $\mathcal{N}^*$, and each connection from unit j to unit i in $\mathcal{N}$ has a copy connecting unit j in layer τ to unit i in layer $\tau + 1$, for each $\tau \in [t_0, t)$. An example of this unrolling mapping is given in Figure 2. The key value of this conceptualization is that it allows one to regard the problem of training a recurrent network as a corresponding problem of training a feedforward network with certain constraints imposed on its weights. The central result driving the BPTT approach is that to compute $\partial J^{\text{total}}(t', t)/\partial w_{ij}$ in $\mathcal{N}$ one simply computes the partial derivatives of $J^{\text{total}}(t', t)$ with respect to each of the $t - t_0$ weights in $\mathcal{N}^*$ corresponding to w_{ij} and adds them up. Thus, the problem of computing the necessary negative error gradient information in the recurrent net $\mathcal{N}$ reduces to the problem of computing the corresponding negative error gradient in the feedforward network $\mathcal{N}^*$, for which one may use standard back propagation.

Straightforward application of this idea leads to two different algorithms, depending on whether an epochwise or continual operation approach is sought.

Detailed mathematical arguments justifying all the results described may be found in the Appendix.

Real-Time Back Propagation through Time

To compute the gradient of $J(t)$ at time t, we proceed as follows. First, we consider t fixed for the moment. This allows us the notational convenience of suppressing any reference to t in the following. We compute values $\epsilon_k(\tau)$ and $\delta_k(\tau)$ for $k \in U$ and $\tau \in (t_0, t]$ by means of the equations

$$\epsilon_k(t) = e_k(t), \tag{12}$$

$$\delta_k(\tau) = f_k'(s_k(\tau))\epsilon_k(\tau), \tag{13}$$

and

$$\epsilon_k(\tau - 1) = \sum_{l \in U} w_{lk}\delta_l(\tau). \tag{14}$$

These equations represent the familiar backpropagation computation. The process begins by using Equations 12 to determine the $\epsilon_k(t)$ values. We call this step *injecting error,* or, if we wish to be more precise, *injecting* $\mathbf{e}(t)$, at time t. Then the δ and ϵ values are obtained for successively earlier time steps (i.e., successively earlier layers in $\mathcal{N}^*$), through the repeated use of Equations 13 and 14. Figure 3 gives a schematic representation of this process.

When each unit in the network uses the logistic squashing function, the relation

$$f_k'(s_k(\tau)) = y_k(\tau)[1 - y_k(\tau)] \tag{15}$$

may be substituted in Equation 13. A corresponding observation applies to all the algorithms to be discussed throughout this chapter.

As described in the Appendix, $\epsilon_k(\tau)$ is just a mathematical shorthand for $\partial j(t)/\partial y_k(\tau)$ and $\delta_k(\tau)$ is just a mathematical shorthand for $\partial J(t)/\partial s_k(\tau)$. Thus $\epsilon_k(\tau)$ represents the sensitivity of the instantaneous performance measure $J(t)$ to small perturbations in the output of the kth unit at time τ, while $\delta_k(\tau)$ represents the corresponding sensitivity to small perturbations to that unit's net input at that time.[6]

Once the back-propagation computation has been performed down to time $t_0 + 1$, the desired gradient of instantaneous performance is computed by

[6]Note that all explicit references to ϵ could be eliminated by reexpressing the δ update equations entirely in terms of other δ values, resulting in a description of back propagation with which the reader may be more familiar. We have chosen to express the computation in this form for two reasons. One is that we will need to make explicit reference to these ϵ quantities later in this chapter; another is that it is useful to recognize that to back-propagate through a semilinear unit is to apply the chain rule through two stages of computation: application of the squashing function and weighted summation.

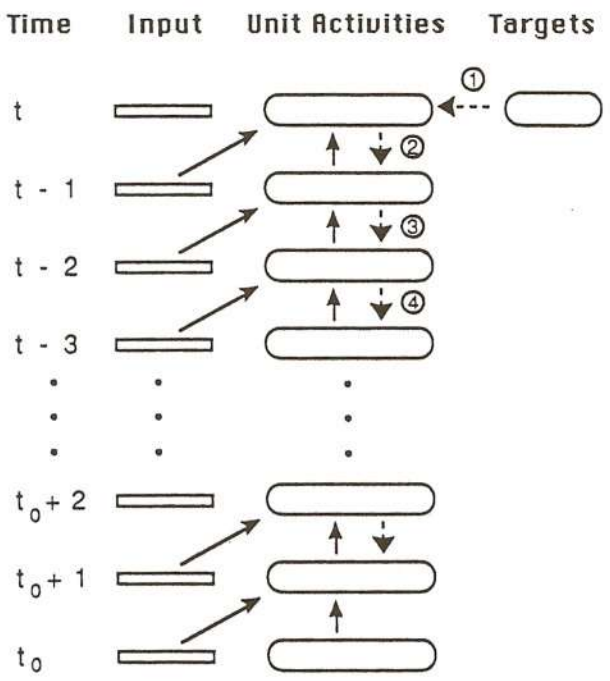

Figure 3. A schematic representation of the storage and processing required for real-time BPTT at each time step t. The history buffer, which grows by one layer at each time step, contains at time t all input and unit output values for every time step from t_0 through t. The solid arrows indicate how each set of unit output values is determined from the input and unit outputs on the previous time step. A backward pass, indicated by the dashed arrows, is performed to determine separate δ values for each unit and for each time step back to $t_0 + 1$. The first step is the injection of external error based on the target values for time step t, and all remaining steps determine virtual error for earlier time steps. Once the backward pass is complete the partial derivative of the negative error with respect to each weight can then be computed.

$$\frac{\partial J(t)}{\partial w_{ij}} = \sum_{\tau=t_0+1}^{t} \delta_i(\tau) x_j(\tau - 1). \tag{16}$$

To summarize, this algorithm, which we call *real-time back propagation through time* performs the following steps at each time t: (1) the current state of the network and the current input pattern is added to a history buffer which stores the entire history of network input and activity since time t_0; (2) error for the current time is injected and back propagation used to compute all the $\epsilon_k(\tau)$ and $\delta_k(\tau)$ values for $t_0 < \tau \leq t$; (3) all the $\partial J(t)/\partial w_{ij}$ values are computed; and (4) weights are changed accordingly. Because this algorithm makes use of potentially unbounded history storage, we will also sometimes denote it BPTT(∞). This algorithm is of more theoretical than practical interest, but later we discuss more practical approximations to it.

Epochwise Back Propagation through Time

An epochwise algorithm based on back propagation through time can be organized as follows. The objective is compute the gradient of $J^{\text{total}}(t_0, t_1)$, which can be obtained after the network has been run through the interval $[t_0, t_1]$. Essentially as before, we compute values $\epsilon_k(\tau)$ and $\delta_k(\tau)$ for $k \in U$ and $\tau \in (t_0, t_1]$, this time by means of the equations

$$\epsilon_k(t_1) = e_k(t_1), \tag{17}$$

$$\delta_k(\tau) = f'_k(s_k(\tau))\epsilon_k(\tau), \tag{18}$$

and

$$\epsilon_k(\tau - 1) = e_k(\tau - 1) + \sum_{l \in U} w_{lk}\delta_l(\tau). \tag{19}$$

These equations represent the familiar back-propagation computation applied to a feedforward network in which target values are specified for units in other layers than the last. The process begins at the last time step, using Equations 17 to determine the $\epsilon_k(t)$ values, and proceeds to earlier time steps through the repeated use of Equations 18 and 19. For this algorithm we speak of *injecting error* at time τ to mean the computational step of adding $e_k(\tau)$ to the appropriate sum when computing $\epsilon_k(\tau)$. The back-propagation computation for this case is essentially the same as that for computing the δ values for the real-time version, except that as one gets to layer τ one must inject error for that time step. Thus, not only are the δ values determined by a backward pass through the unrolled network, but the errors committed by the network are also taken into account in reverse order. Figure 4 gives a schematic representation of this process.

It is useful to regard the sum on the right-hand side of Equation 19 as a *virtual error* for unit k at time $\tau - 1$. We might also say that this unit has been given a *virtual target* value for this time step. Thus, in epochwise BPTT, virtual error is added to external error, if any, for each unit at each time step in the backward pass. Note that in real-time BPTT the only contribution to each ϵ is either external error, at the most recent time step, or virtual error, at all earlier time steps.

As with real-time BPTT, $\epsilon_k(\tau)$ is just a mathematical shorthand, this time for $\partial J^{\text{total}}(t_0, t_1)/\partial y_k(\tau)$; similarly, $\delta_k(\tau)$ is just a mathematical shorthand for $\partial J^{\text{total}}(t_0, t_1)/\partial s_k(\tau)$. Thus $\epsilon_k(\tau)$ represents the sensitivity of the overall performance $J^{\text{total}}(t_0, t_1)$ to small perturbations in the output of the kth unit at time τ, while $\delta_k(\tau)$ represents the corresponding sensitivity to small perturbations to that unit's net input at that time.

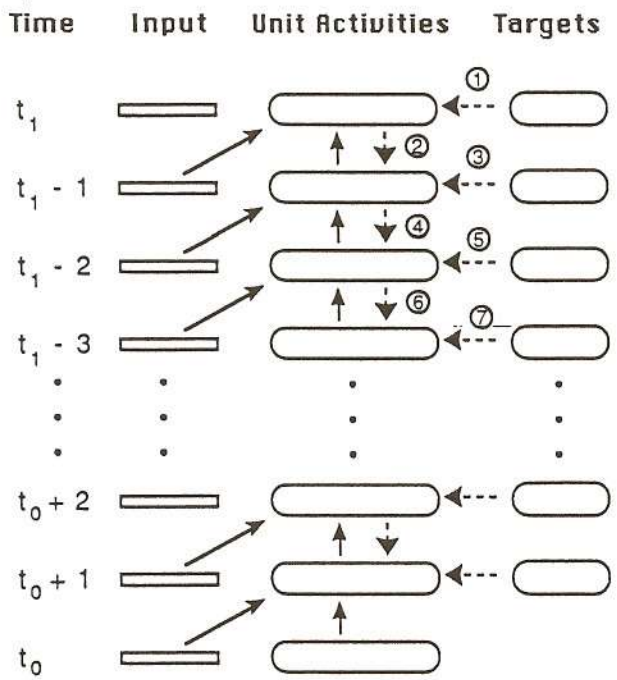

Figure 4. A schematic representation of the storage and processing required for epochwise BPTT. All input, unit output, and target values for every time step from t_0 and t_1 are stored in the history buffer. The solid arrows indicate how each set of unit output values is determined from the input and unit outputs on the previous time step. After the entire epoch is complete, the backward pass is performed as indicated by the dashed arrows. Each even-numbered step determines the virtual error from later time steps, while each odd-numbered step corresponds to the injection of external error. Once the backward pass has been performed to determine separate δ values for each unit and for each time step back to $t_0 + 1$, the partial derivative of the negative error with respect to each weight can then be computed.

Once the back-propagation computation has been performed down to time $t_0 + 1$, the desired gradient of overall performance is computed by

$$\frac{\partial J^{\text{total}}(t_0,t_1)}{\partial w_{ij}} = \sum_{\tau=t_0+1}^{t_1} \delta_i(\tau)x_j(\tau - 1). \tag{20}$$

Epochwise BPTT thus must accumulate the history of activity in (and input to) the network over the entire epoch, along with the history of target values (or, equivalently, the history of errors) over this epoch, after which the following steps are performed: (1) the back-propagation computation is carried out to obtain all the $\epsilon_k(\tau)$ and $\delta_k(\tau)$ values for $t_0 < \tau \leq t_1$; (2) all the $\partial J^{\text{total}}(t_0, t_1)/\partial w_{ij}$ values are computed; and (3) weights are changed accordingly. Then the network is reinitialized and this process repeated.

Epochwise BPTT Applied to Settling Networks

Although our main interest here is in the general problem of training networks to perform time-varying behaviors, the BPTT formulation leads to a simple algorithm for training settling networks with constant input, whenever certain assumptions hold. This algorithm, which is a discrete-time version of the algorithm described by Almeida (1987) and Pineda (1987; chapter 4, this volume) is obtained as follows.

First, suppose that a network is to be driven with constant input and that we have initialized it to a state which represents a fixed point for its dynamics. Suppose further that we intend to observe this state at the end of the epoch $[t_0, t_1]$ to compare it with some desired state. If we were to use epochwise BPTT for this situation, the appropriate equations would be

$$\epsilon_k(t_1) = e_k(t_1), \tag{21}$$

$$\delta_k(\tau) = f'_k(s_k(t_1))\epsilon_k(\tau), \tag{22}$$

and

$$\epsilon_k(\tau - 1) = \sum_{l \in U} w_{lk}\delta_l(\tau), \tag{23}$$

with weight changes determined by

$$\frac{\partial J^{\text{total}}(t_0,t_1)}{\partial w_{ij}} = \sum_{\tau=t_0+1}^{t} \delta_i(\tau)x_j(\tau - 1) = \sum_{\tau=t_0+1}^{t} \delta_i(\tau)x_j(t_1)$$

$$= x_j(t_1) \sum_{\tau=t_0+1}^{t} \delta_i(\tau). \tag{24}$$

Note that this last result takes into account the fact that all states and all input during the epoch are equal to their values at the end of the epoch. Thus there is no need to save the history of input and network activity in this case.

Now define

$$\epsilon^*_k(t) = \sum_{\tau=t+1}^{t_1} \epsilon_k(\tau) \tag{25}$$

and

$$\delta_k^*(t) = \sum_{\tau=t+1}^{t_1} \delta_k(\tau). \tag{26}$$

Then Equation 24 becomes

$$\frac{\partial J^{\text{total}}(t_0,t_1)}{\partial w_{ij}} = \delta_i^*(t_0)x_j(t_1). \tag{27}$$

Furthermore, it is easy to check by induction that

$$\epsilon_k^*(t_1) = e_k(t_1), \tag{28}$$

$$\delta_k^*(\tau) = f_k'(s_k(t_1))\epsilon_k^*(\tau), \tag{29}$$

and

$$\epsilon_k^*(\tau - 1) = e_k(t_1) + \sum_{l \in U} w_{lk}\delta_l^*(\tau). \tag{30}$$

Thus the δ^* and ϵ^* values may be interpreted as representing the δ and ϵ values obtained from performing epochwise BPTT from t_1 back to t_0 while injecting the constant error $\mathbf{e}(t_1)$ at each time step, while Equation 27 has the form of the usual feedforward backpropagation computation for determining the partial derivative of error with respect to any weight.

Now consider what happens in the limit as the epoch is made very long. In this case, the computation of the $\delta_i^*(t_0)$ values by means of Equations 28–30 can be viewed as a settling computation, assuming it converges. As it turns out, it can be shown that the BPTT computation given by Equations 21–23 will "die away" (meaning that the backpropagated quantities $\delta_k(\tau)$ and $\epsilon_k(\tau)$ will decrease to zero) exponentially fast as long as the network has reached a *stable* equilibrium state, which implies that the settling computation for the $\delta_i^*(t_0)$ values does indeed converge in this case.

The *recurrent back-propagation* (RBP) algorithm (Almeida, 1987; Pineda, 1987) for training settling networks having constant input consists of applying the following steps: (1) the network is allowed to settle (with the time at which settling has completed regarded as t_1); (2) the BPTT computation given by equations 28–30 is performed for as long as needed until the δ_i^* values converge; (3) all the $\partial J^{\text{total}}(t_0, t_1)/\partial w_{ij}$ values are computed using Equation 27; and (4) weights are changed accordingly. The appealing features of this algorithm are that it does not require the storage of any past history to implement and is entirely local. The reason it requires no history storage is that it implicitly assumes that all relevant past states and input are equal to their current values. This algorithm is thus applicable only to situations where both the desired and actual behaviors of the network are limited to stable settling.

The argument presented so far shows that RBP would compute the same thing as the BPTT computation given by Equations 21–23 over a very long epoch in which the network state is held constant at a stable equilibrium. Now, continue to assume that the input to the network is constant throughout the entire epoch, but assume instead that the network has settled to an equilibrium state from possibly some other starting state by the end of the epoch, at time t_1. Assume further that it has reached this equilibrium state long before t_1. Because the BPTT computation resulting from injecting error only at time t_1 dies away, as described earlier, even in this case RBP and this BPTT computation yield essentially the same result. That is, if error is injected only long after the network has arrived at its steady-state behavior, the full BPTT computation will also give the same result as RBP, because the BPTT computation dies away before reaching the transient portion of the network's behavior. This shows clearly that not only is RBP limited to training settling networks, but it is really only designed to directly influence their fixed points and cannot control their transient behaviors. In general, RBP is only capable of perturbing the equilibrium states already present in the network's dynamics.[7] On the other hand, as long as errors are injected within (or soon after) the transient behavior of a network, BPTT can directly influence such transient behavior.

These observations concerning the inability of even full BPTT to reach back into the transient behavior if error is injected too long after steady-state behavior is reached have some other interesting consequences for the problem of training continually operating networks, which we describe below when we discuss the teacher forcing strategy.

Computational Requirements of BPTT Algorithms

It is clear that to store the history of m-dimensional input to and n-dimensional activity of a network over h time steps requires $(m + n)h$ numbers. In addition, the number of target values over these h time steps is no greater than nh. Thus the gradient computation performed for epochwise BPTT has space complexity in $\Theta((m + n)h)$, where h represents the epoch length. However, for BPTT(∞) this history must continue to grow indefinitely. With L representing the total time over which the network is actually run, the space complexity of BPTT(∞) is thus in $\Theta((m + n)L)$.

To determine the number of arithmetic operations required for these algorithms, note that Equation 13 requires an evaluation of $f_k'(s_k(\tau))$ plus one multiplication for each k in U. For the logistic squashing function this amounts to two multiplications per unit for determining the δ values from the corresponding ϵ values. In general, the number of operations required for this part of the backpropagation computation is in $\theta(n)$. Application of Equation 14 for all $k \in U$ at each fixed τ clearly requires w_U multiplications and $w_U - 1$ additions, while

[7] However, as we discuss later, Pineda (1988; chapter 4, this volume) has shown that new equilibrium points can be created by combining RBP with the *teacher forcing* technique.

application of Equation 19 for all $k \in U$ at each fixed τ requires the same number of multiplications and up to n more additions and subtractions, depending on how many units have target values for that time step. As long as we assume $w_U \in \Omega(n)$, it follows that each stage of the back-propagation computation has time complexity in $\Theta(w_U)$, regardless of whether error is injected at all time steps during the backward pass, as in epochwise BPTT, or just at the last time step, as in real-time BPTT.

Now let $h = t - t_0$, where t represents the time at which BPTT is performed for either real-time or epochwise BPTT. (In the latter case, $t = t_1$.) It is clear that Equation 16, which must be evaluated once for each adaptable weight, requires h multiplications and $h - 1$ additions, leading to a total of $\Theta(w_A h)$ operations. Thus the total number of operations required to compute the gradient for one epoch in epochwise BPTT is in $\Theta(w_U H + w_A h)$.[8] Amortized across the h time steps of the epoch, the gradient computation for epochwise BPTT requires an average of $\Theta(w_U + w_A)$ operations per time step. For real-time BPTT, a back-propagation computation all the way back to t_0 must be performed any time a target is specified. Thus the total number of operations required over the entire training interval of length L is in $\Theta((w_U + w_A)T^2/\Delta_T)$, which is an average of $\Theta((w_U T + w_A)T/\Delta_T)$ operations per time step. These complexity results are summarized in Table 1.

The worst case for either of these algorithms for any fixed n is when the network is fully connected, all weights are adaptable, and target values are supplied at every time step, so that $\Delta_T = 1$. In this case, epochwise BPTT has space complexity in $\Theta(nh)$ and average time complexity per time step in $\Theta(n^2)$, while real-time BPTT has space complexity in $\Theta(nL)$ and average time complexity per time step in $\Theta(n^2 L)$, as shown in Table 2.

Note that when weights are changed throughout the course of operating the network, a variant of real-time BPTT is possible in which the history of weight values are saved as well and used for the backpropagation computation, by replacing w_{lk} by $w_{lk}(\tau)$ in Equation 14. For this algorithm, the storage requirements are in $\Theta((m + n + w_A)T)$ in the general case and in $\Theta(n^2 T)$ in the worst case.

While real-time BPTT could be used to train a network which is operated in epochwise fashion, it is clearly inefficient to do so because it must duplicate some computation which need only be performed once in epochwise BPTT. Epochwise BPTT computes $\nabla_{\mathbf{W}} J^{\text{total}}(t_0, t_1)$ without ever computing any of the gradients $\nabla_{\mathbf{W}} J(t)$ for individual time steps t.

[8] This assumes that there is some error to inject at the last time step. In general, it is also assumed throughout this analysis that the number of units given targets and the connectivity of the network are such that backpropagation "reaches" every unit. If this is not true, then the time complexity could be lower for an algorithm designed to take advantage of this.

THE REAL-TIME RECURRENT LEARNING ALGORITHM

While BPTT uses the backward propagation of error information to compute the error gradient, an alternative approach is to propagate activity gradient information forward. This leads to a learning algorithm which we have called *real-time recurrent learning* (RTRL). This algorithm has been independently derived in various forms by Robinson and Fallside (1987), Kuhn (1987), Bachrach (1988; Chapter 11, this volume), Mozer (1989; chapter 5, this volume), and Williams and Zipser (1989a), and continuous-time versions have been proposed by Gherrity (1989), Doya and Yoshizawa (1989), and Sato (1990a; 1990b).

The Algorithm

For each $k \in U$, $i \in U$, $j \in U \cup I$, and $t_0 \le t \le t_1$, we define

$$p_{ij}^k(t) = \frac{\partial y_k(t)}{\partial w_{ij}}. \tag{31}$$

This quantity 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 but assuming that the initial state of the network, the input over $[t_0, t)$, and the remaining weights are not altered.

From Equations 7 and 8 and use of the chain rule, we find that

$$\frac{\partial J(t)}{\partial w_{ij}} = \sum_{k \in U} e_k(t) p_{ij}^k(t) \tag{32}$$

for each $i \in U$ and $j \in U \cup I$. Also, differentiating Equations 2 and 3 for the network dynamics yields

$$p_{ij}^k(t+1) = f_k'(s_k(t+1)) \left[\sum_{l \in U} w_{kl} p_{ij}^l(t) + \delta_{ik} x_j(t) \right], \tag{33}$$

where δ_{ik} denotes the Kronecker delta. Furthermore,

$$p_{ij}^k(t_0) = \frac{\partial y_k(t_0)}{\partial w_{ij}} = 0 \tag{34}$$

since we assume that the initial state of the network has no functional dependence on the weights. These equations hold for all $k \in U$, $i \in U$, $j \in U \cup I$, and $t \ge t_0$.

Thus we may use Equations 33 and 34 to compute the quantities $\{p_{ij}^k(t)\}$ at each time step in terms of their prior values and other information depending on activity in the network at that time. Combining these values with the error vector $\mathbf{e}(t)$ for that time step via Equations 32 then yields the negative error gradient

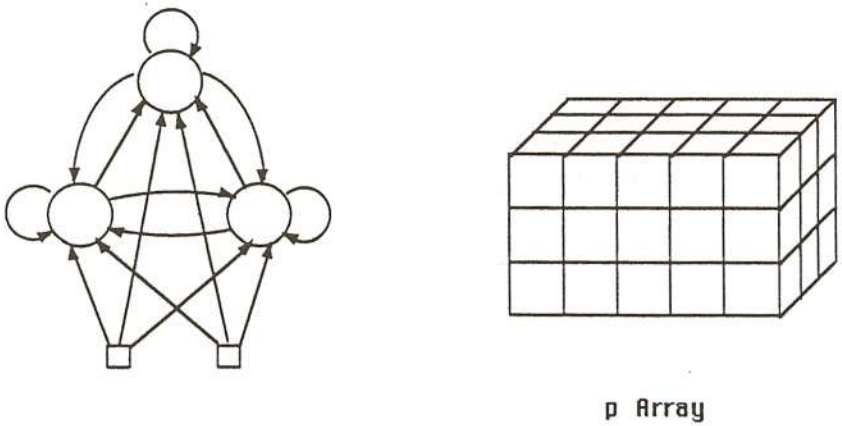

Figure 5. The data structures that must be updated on each time step to run the RTRL algorithm with the network of Figure 1. In addition to updating the three unit activities within the network itself on each time step (along with the 15 weights, if appropriate), the $3 \times 5 \times 3$ array of p_{ij}^k values must also be updated. It is assumed here that all 15 weights in the network are adjustable. In general, a p_{ij}^k value for each combination of adjustable weight and unit in the network must be stored and updated on each time step for RTRL.

$\nabla_{\mathbf{W}} J(t)$. Because the $p_{ij}^k(t)$ values are available at time t, the computation of this gradient occurs in real time. Figure 5 depicts the data structures that must be updated on each time step to run the RTRL algorithm with the network of Figure 1.

Computational Requirements

The computational requirements of the RTRL algorithm arise from the need to store and update all the p_{ij}^k values. To analyze these requirements, it is useful to view the triply indexed set of quantities p_{ij}^k as forming 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 w_A adjustable weights is nw_A. Furthermore, from Equation 33 it is clear that the number of multiplications and the number of additions required to update all the p_{ij}^k values are each essentially equal to $w_U w_A$. Note that this computation is performed on every time step, regardless of whether target values are specified for that time step.

In addition, Equation 32 requires one multiplication (and approximately one addition) at each time step for each unit given a target on that time step and each adjustable weight. This amounts to an average of $\Theta(n_T w_A)$ operations per time step. Thus the space complexity of the gradient computation for RTRL is in $\Theta(nw_A)$, and its average time complexity per time step is in $\Theta(w_U w_A)$, as indicated in Table 1. When the network is fully connected and all weights are adaptable, this algorithm has space complexity in $\Theta(n^3)$ and average time complexity per time step in $\Theta(n^4)$, as shown in Table 2.

While this time complexity is quite severe for serial implementation, part of the appeal of this algorithm is that it can run in $O(n)$ time per time step using $\Theta(n^3)$ processors. However, this raises the question of its communication requirements, especially in relation to the network being trained. Interestingly, update of the p_{ij}^k values can be carried out using a completely local communication scheme in the network being trained if one allows n-tuples to be communicated along network connections rather than single real numbers. The idea is to let each unit k store within it the set of numbers p_{ij}^k with (i, j) ranging over all weights in the network. If we regard this set of numbers as a vector $\mathbf{p}^k$, then the set of Equations 33 corresponding to each fixed value of k can be organized into a single vector update equation. In this way, one can imagine a network of units which pass not only their activations around, but also thee $\mathbf{p}^k$ vectors. However, the actual computation of $\nabla_{\mathbf{W}} J(t)$ by means of Equation 32 ultimately requires global access to the $\mathbf{p}^k$ vectors.

Without giving details, we note that the entire RTRL algorithm could be carried out in a more conventional scalar-value-passing network having, in addition to the n units of the network to be trained, an additional unit for each p_{ij}^k value and an additional unit for each connection in the network to be trained. Each unit in this last set would simultaneously gate numerous connections among the remaining units.

A HYBRID ALGORITHM

It is possible to formulate a hybrid algorithm incorporating aspects of both BPTT and the forward gradient propagation computation used in RTRL. This algorithm, first proposed in Williams 1989, and later described by Schmidhuber (1992), is interesting both because it helps shed light on the relationship between BPTT and RTRL and because it can yield exact error gradient information for a

continually running network more efficiently than any other method we know. The mathematical derivation of this algorithm is provided in the Appendix. Here we describe the steps of the algorithm and analyze its computational complexity.

The Algorithm

This algorithm involves a segmentation of time into disjoint intervals each of length $h = t - t'$, with weight changes performed only at the end of each such interval. By our definition, then, this is not a real-time algorithm when $h > 1$. Nor is it an epochwise algorithm, since it does not depend on the artificial imposition of credit-assignment boundaries and/or state resets. The segmentation into intervals is purely arbitrary and need have no relation to the task being performed. Over each such interval $[t', t]$ the history of activity of (and input to) the network is saved; at the end of this time period, a computation to be described is performed. Then the process is begun anew, beginning with collecting the history of the network activity starting at time t (which becomes the new value of t').

This algorithm depends on having all the values $p_{ij}^k(t')$, as used in RTRL, for the start of each time period. For the moment, we assume that these are available; later we describe how they are updated by this algorithm. Then the equations

$$\epsilon_k(\tau) = \begin{cases} e_k(t) & \text{if } \tau = t \\ e_k(\tau) + \sum_{l \in U} w_{lk}\delta_l(\tau + 1) & \text{if } \tau < t \end{cases} \tag{35}$$

and

$$\delta_k(\tau) = f_k'(s_k(\tau))\epsilon_k(\tau) \tag{36}$$

are used to compute all the values $\epsilon_k(\tau)$ for $t' \le \tau \le t$ and $\delta_k(\tau)$ for $t' < \tau \le t$. This computation is essentially identical to an epochwise BPTT computation over the interval $[t', t]$. In particular, note that each error vector $\mathbf{e}(\tau)$, for $t' < \tau \le t$, is injected along the backward pass. Once all these ϵ and δ values are obtained, the gradient of $J^{\text{total}}(t', t)$, the cumulative negative error over the time interval $(t', t]$, is computed by means of the equations

$$\frac{\partial J^{\text{total}}(t', t)}{\partial w_{ij}} = \sum_{l \in U} \epsilon_l(t')p_{ij}^l(t') + \sum_{\tau=t'}^{t-1} \delta_i(\tau + 1)x_j(\tau), \tag{37}$$

for each i and j.

Note that the second sum on the right-hand side is what would be computed for this partial derivative if one were to truncate the BPTT computation at time t', while the first sum represents a correction in terms of the p values used in RTRL. There are two special cases of this algorithm worth noting. When $t' = t$, the second sum in Equation 37 vanishes and we recover the RTRL Equation 32 expressing the desired partial derivatives in terms of the current p values. When

$t' = t_0$, the first sum in Equation 37 vanishes and we recover Equation 16 for the BPTT(∞) algorithm.

Thus far we have described how the desired error gradient is obtained, assuming that the p values are available at time t'. In order to repeat the same process over the next time interval, beginning at time t, the algorithm must also compute all the values $p_{ij}^r(t)$. For the moment, consider a fixed r in U. Suppose that we were to inject error $\mathbf{e}^r(t)$ at time t, where $e_k^r(t) = \delta_{kr}$ (the Kronecker delta), and use BPTT to compute $\partial J(t)/\partial w_{ij}$. It is clear from Equation 32 that the result would be equal to $p_{ij}^r(t)$. Thus this gives an alternative view of what these quantities are. For each r, the set of numbers $p_{ij}^r(t)$ represents the negative error gradient that would be computed by BPTT if unit r were given a target 1 greater than its actual value. Furthermore, we may use the same approach just used to compute the partial derivatives of an arbitrary error function to compute the partial derivatives of this particular imagined error function. Thus, to compute $p_{ij}^r(t)$ for all i and j, the algorithm first performs a BPTT computation using the equations[9]

$$\epsilon_k(\tau) = \begin{cases} \delta_{kr} & \text{if } \tau = t, \\ \sum_{l \in U} w_{lk}\delta_l(\tau + 1) & \text{if } \tau < t, \end{cases} \tag{38}$$

together with Equations 36, to obtain a set of values[10] $\epsilon_k(\tau)$ for $t' \le \tau \le t$ and $\delta_k(\tau)$ for $t' < \tau \le t$. These values are then used to compute $p_{ij}^r(t)$ for each i and j by means of the equations

$$p_{ij}^r(t) = \sum_{l \in U} \epsilon_l(t')p_{ij}^l(t') + \sum_{\tau=t'}^{t-1} \delta_i(\tau + 1)x_j(\tau). \tag{39}$$

In other words, to compute $p_{ij}^r(t)$, a 1 is injected at unit r at time t and BPTT performed back to time t', and the results substituted into Equation 39.

This process is repeated for each r in U in order to obtain all the p values for time t. Thus this algorithm involves a total of $n + 1$ different BPTT computations, one to compute the error gradient and n to update the p values. Because this algorithm involves both a forward propagation of gradient information (from time t' to time t) and backward propagation through time, we will denote this

[9]The reader is warned to avoid confusing the singly subscripted (and time-dependent) quantities denoted δ_l, which are obtained via back propagation, with the doubly subscripted Kronecker delta, such as δ_{kr}. Both uses of the symbol δ appear throughout the equations presented in this and the next section.

[10]The reader should understand that, although we are denoting the result of several different BPTT computations in the same way, the various sets of δ and ϵ values obtained from each BPTT computation are unrelated to each other. We have resisted introducing additional notation here which might make this clearer, on the grounds that it might clutter the presentation. A more precise formulation may be found in the Appendix.

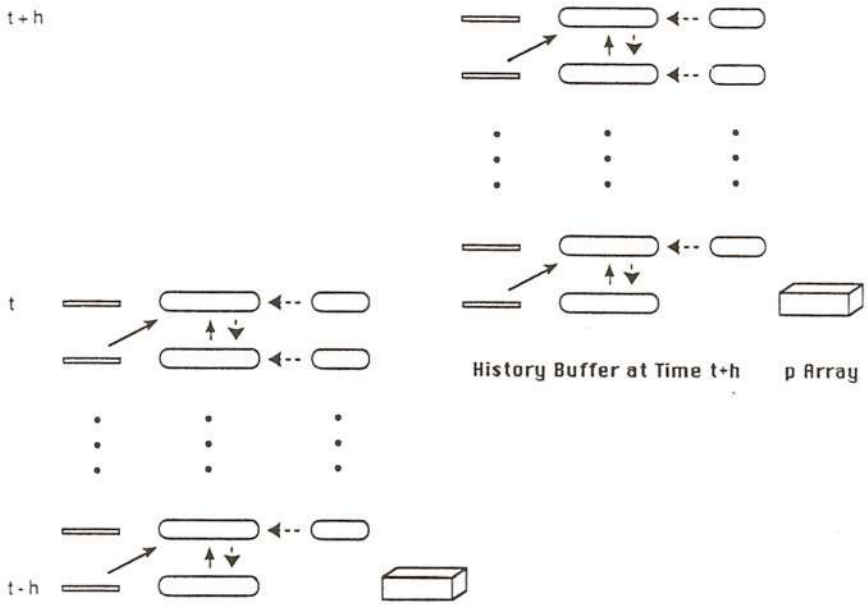

Figure 6. A schematic representation of the storage and processing required for the FP/BPTT(h) algorithm for two consecutive executions of the error gradient computation, one at time step t and the next at time step $t + h$. From time step $t - h$ through time step t the network input, activity, and target values are accumulated in the history buffer. At time t the cumulative error gradient is computed on the basis of one BPTT pass through this buffer, also using the p values stored for time step $t - h$. In addition, n separate BPTT passes, one for each unit in the network, are performed to compute the p values for time t. Each such BPTT pass begins with the injection of 1 as "error" at a single unit at the top level. Once the weights have been adjusted on the basis of the cumulative error gradient over the interval $(t - h, t]$ and the p values have been updated at time t, accumulation of the history begins anew over the interval $[t, t + h]$.

algorithm FP/BPTT(h), where $h = t - t'$ is the number of past states which are saved in the history buffer. Figure 6 gives a schematic representation of the storage and processing required for this algorithm.

Computational Requirements

This hybrid algorithm requires $\Theta(nw_A)$ storage for the p_{ij} values, like RTRL, and $\Theta((m + n)h)$ storage for the history of network input, activity, and teacher signals over the interval $[t', t]$, like epochwise BPTT. In addition, each BPTT

computation requires $\Theta(nh)$ storage for all the δ and ϵ values, but this space may be reused for each of the $n + 1$ applications of BPTT. Thus its overall storage requirements are in $\Theta(nw_A + (m + n)h)$.

To determine the number of arithmetic operations performed, note that each BPTT computation requires $\Theta((w_U + w_A)h)$ operations, and, for each such BPTT computation, equation 37, requiring $\Theta(n + h)$ operations, must be used for each adjustable weight, or w_A times. Thus the number of operations required for each of the $n + 1$ applications of BPTT requires $\Theta(w_U h + 2w_A h + nw_A) = \Theta(w_U h + w_A h + nw_A)$, giving rise to a total number of operations in $\Theta(nw_U h + nw_A h + n^2 w_A)$. Since this computation is performed every h time steps, the average number of operations per time step is in $\Theta(nw_U + nw_A + n^2 w_A/h)$. When the network is fully connected and all weights are adaptable, FP/BPTT(h) has space complexity in $\Theta(n^3 + nh)$ and average time complexity per time step in $\Theta(n^3 + n^4/h)$. Thus, by making h proportional to n, the resulting algorithm has worst-case space complexity in $\Theta(n^3)$ and time complexity per time step in $\Theta(n^3)$. Thee complexity results are summarized in Tables 1 and 2.

This means that of all exact gradient computation algorithms for continually operating networks, FP/BPTT(cn), where c is any constant, has superior asymptotic complexity properties. Its asymptotic space complexity is no worse than that of RTRL, and its asymptotic time complexity is significantly better. The reduction in time complexity in comparison to RTRL is achieved by only performing the update of the p_{ij}^k values after every cn time steps. The improvement in both time and space complexity over real-time BPTT over long training times is achieved because there is no need to apply BPTT further back than to the point where these p_{ij}^k values are available.

SOME ARCHITECTURE-SPECIFIC APPROACHES

Up to now, we have restricted attention to the case where every connection in the network is assumed to have a delay of one time step. It is sometimes useful to relax this assumption. In particular, a number of researchers have proposed specific mixed feedforward/feedback architectures for processing temporal data. In almost all of these architectures the feedforward connections are assumed to have no delay while the feedback connections are assumed to incorporate a delay of one time step. After briefly considering the case of arbitrary (but fixed) delays, we then focus in this section on exact gradient algorithms for certain classes of network architectures where all delays are 0 or 1.

Connection-Dependent Delays

To handle the general case in which various connections in the network have different delays, Equation 2 for the network dynamics must be replaced by

$$s_k(t) = \sum_{l \in U \cup I} w_{kl} x_l(t - \Delta_{kl}), \tag{40}$$

where Δ_{kl} represents the delay on the connection from unit (or input line) l to unit k. In general, we may allow each delay to be any nonnegative integer, as long as the subgraph consisting of all links having delay 0 is acyclic. This condition is necessary and sufficient to guarantee that there is a fixed ordering of the indices in U such that, for any t and k, $s_k(t)$ depends only on quantities $x_l(t')$ having the property that $t' < t$ or l comes before k in this ordering.

As an alternative to allowing multiple delays, one could instead transform any such setup into a form where all delays are 1 by adding "delay units" along paths having a delay larger than 1 and repeating computations along paths having delay 0, but this is generally undesirable in simulations. Because holding a value fixed in memory is a no-cost operation on a digital computer, it is always more efficient to simulate such a system by only updating variables when necessary. For example, in a strictly layered network having h layers of weights, although they both lead to the same result, it is clearly more efficient to update activity one layer at a time than to run one grand network update a total of h times. A similar observation applies to the backward pass needed for back propagation. Figure 7 illustrates a case where all links have delay 0 or 1 and shows a useful way to conceptualize the unrolling of this network.

Watrous and Shastri (1986) have derived a generalization of BPTT to this more general case, and it is straightforward to extend the RTRL approach as well. With a little more effort, the hybrid algorithm described here can also be generalized to this case. Rather than give details of these generalizations, we confine attention in the remainder of this section to some particular cases where all delays are 0 or 1 and describe some exact gradient computation algorithms for these involving both backward error propagation and forward gradient propagation. These cases represent modest generalizations of some specific mixed feedforward/feedback architectures which have been considered by various researchers.

Some Special Two-Stage Architectures

The architectures to be investigated here involve limited recurrent connections added to what would otherwise be a feedforward net. We regard these architectures as consisting of two stages, which we call a *hidden stage* and an *output stage*. The output stage must contain all units given targets, but it need not be confined to these. The hidden stage contains all units not in the output stage. As a minimum, each architecture has feedforward connections from the hidden stage to the output stage, and there may be additional feedforward connections within each stage as well. Thus, in particular, each stage may be a multilayer network. Let U_O denote the set of indices of units in the output stage, and let U_H denote the set of indices of units in the hidden stage.

Here we restrict attention to three classes of recurrent net which consist of this

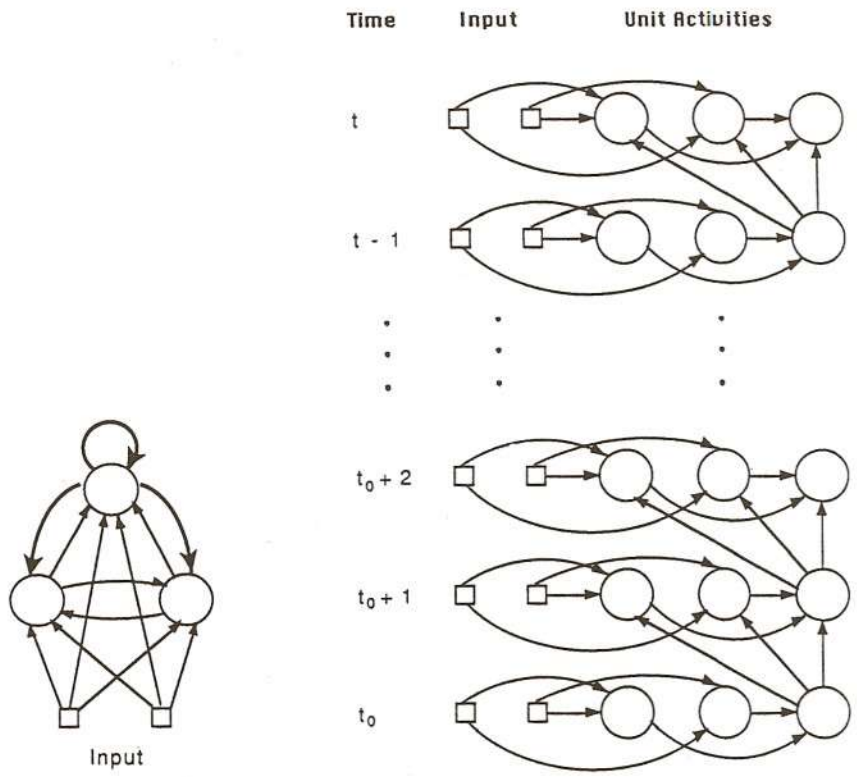

Figure 7. A network having connections with delays of 0 and 1 and its unrolling from time t_0 to t. The feedforward connections, indicated by the thinner arrows in the network itself, all have a delay of 0. These correspond to the within-level connections in the unrolled version. The feedback connections, indicated by the thicker arrows in the network, all have a delay of 1. These correspond to the connections from each level to the next level above it in the unrolled version. Other delays beside 0 and 1 are possible and would be represented by connections that skip levels. In the unrolled network, updating of activity is assumed to occur from left to right within each level and then upward to the next level. Thus a sequence of operations is performed within each single time step when computing the activity in the network. When errors are back-propagated, processing goes in the reverse direction, from higher levels to lower levels and from right to left within each level.

minimum feedforward connectivity plus some additional recurrent connections. In all cases, we assume that the feedforward connections have delay 0 and the added feedback connections have delay 1. For any given network which falls into one of these categories there may be many ways to decompose it into the two stages, and particular recurrent networks may be viewed as belonging to more

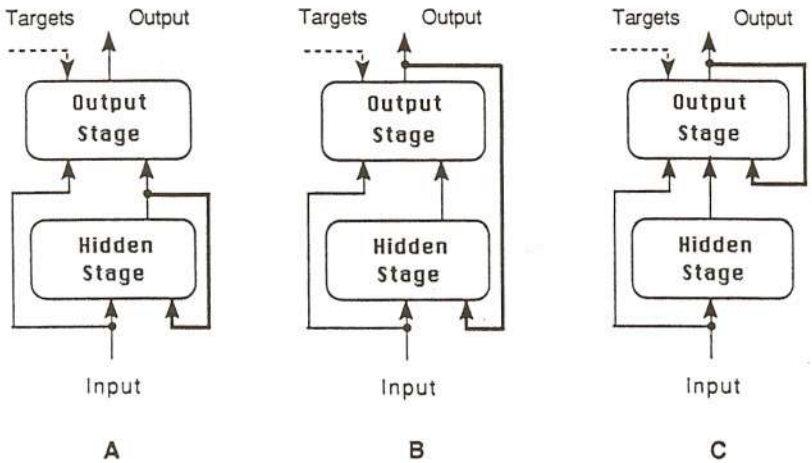

Figure 8. Three special architectures where all connections have delays of 0 or 1 time step. In each case the hidden stage and the output stage have only 0-delay feedforward connections within them. They may each consist of multilayer networks, for example. It is also assumed that there is no delay on the input connections or the feedforward connections from units in the hidden stage to units in the output stage. The output stage must contain all units which receive target values. Input may optionally feed directly to the output stage, as indicated. The feedback connections, indicated by the heavier arrows, all have a delay of 1 time step. The three possible feedback configurations are where (a) all feedback is confined to the hidden stage; (b) all feedback goes from the output stage to the hidden stage; and (c) all feedback is confined to the output stage. A specialized mixture of back propagation and RTRL is applicable to each of these architectures.

than one category, depending on which units are assigned to which stage. We consider feedback connections confined to one of three possibilities: internal feedback within the hidden stage, feedback from the output stage to the hidden stage, and internal feedback within the output stage. Figure 8 depicts these three architectures. In this section we omit discussion of the computational complexity of the algorithms described.

Hidden-to-Hidden Feedback Only

Figure 8a illustrates a general architecture in which all feedback connections are confined to the hidden stage. One example of this architecture is provided by the work of Elman (1988), who has considered a version in which the hidden stage and the output stage are each one-layer networks, with feedback connections provided between all units in the hidden stage. Cleeremans, Servan-Screiber, and McClelland (1989; Chapter 9, this volume) have also studied this

architecture extensively. One approach to creating an real-time, exact gradient algorithm for this architecture is to use a hybrid strategy involving both RTRL and back propagation. In this approach, the p_{ij}^k values need only be stored and updated for the hidden units, with back propagation used to determine other necessary quantities. Mathematical justification for the validity of this approach is based on essentially the same arguments used to derive the hybrid algorithm FP/BPTT(h).

The error gradient is computed by means of

$$\frac{\partial J(t)}{\partial w_{ij}} = \begin{cases} \delta_i(t) x_j(t) & \text{if } i \in U_O, \\ \sum_{l \in U_H} \epsilon_l(t) p_{ij}^l(t), & \text{if } i \in U_H, \end{cases} \tag{41}$$

where $\delta_i(t)$ is obtained by back propagation entirely within the hidden stage.

The p_{ij}^k values, for $k \in U_H$, are updated by means of the equations

$$p_{ij}^k(t) = f_k'(s_k(t)) \left[\sum_{l \in U_H} w_{kl} p_{ij}^l(t-1) + \delta_{ik} x_j(t-1) \right], \tag{42}$$

which are just the RTRL Equations 33 specialized to take into account the fact that w_{kl} is 0 if $l \in U_O$.

One noteworthy special case of this type of architecture has been investigated by Mozer (1989; chapter 5, this volume). For this architecture, the only connections allowed between units in the hidden stage are self-recurrent connections. In this case, p_{ij}^k is 0 except when $k = i$. This algorithm can then be implemented in an entirely local fashion by regarding each p_{ij}^i value as being stored with w_{ij}, because the only information needed to update p_{ij}^i is locally available at unit i. The algorithm described here essentially coincides with Mozer's algorithm except that his net uses a slightly different form of computation within the self-recurrent units.

Output-to-Hidden Feedback Only

Figure 8b illustrates a general architecture in which all feedback connections go from the output stage to the hidden stage. One example of this architecture is provided by the work of Jordan (1986), who has considered a version in which the hidden stage and the output stage are each one-layer networks, with feedback connections going from all units in the output stage to all units in the hidden stage. As in the preceding case, we consider a hybrid approach for this architecture involving both RTRL and back propagation. In this case, the p_{ij}^k values are only stored and updated for the output units. Mathematical justification for the validity of this approach is based on essentially the same arguments used to derive the hybrid algorithm FP/BPTT(h).

The error gradient is computed by means of the equation

$$\frac{\partial J(t)}{\partial w_{ij}} = \sum_{k \in U_O} e_k(t) p_{ij}^k(t), \tag{43}$$

which is just the RTRL Equation 32 specialized to take into account the fact that e_k is always 0 for $k \in U_H$.

The updating of the p values for units in the output stage is based on performing a separate backpropagation computation for each $k \in U_O$, in a manner very much like that used in the hybrid algorithm FP/BPTT(h). To compute $p_{ij}^k(t)$, for $k \in U_O$, inject a 1 as "error" at the kth unit and back-propagate all the way from the output stage, through the hidden stage, and through the feedback connections, right back to the output stage at the previous time step. Then compute

$$p_{ij}^k(t) = \sum_{l \in U_O} \epsilon_l(t-1) p_{ij}^l(t-1) + \delta_i(t) x_j(t - \Delta_{ij}), \tag{44}$$

where Δ_{ij} is 1 if $j \in U_O$ and 0 otherwise. The relevant $\delta_i(t)$ and $\epsilon_i(t-1)$ values are obtained from the back-propagation computation, with a new set obtained for each k.

Output-to-Output Feedback Only

Figure 8c illustrates a general architecture in which all feedback connections are confined to the output stage. Just as in the previous cases, we consider a hybrid approach in which the p_{ij}^k values need only be stored and updated for the output units, with back propagation used to determine other necessary quantities. As before, the error gradient is computed by means of Equation 43.

Updating of the p_{ij}^k values is performed using a slightly different mix of back propagation and forward gradient propagation than in the previous case. To derive this, we write the equation computing net input for a unit in the output stage as

$$s_k(t) = \sum_{l \in U_H \cup I} w_{kl} x_l(t) + \sum_{l \in U_O} w_{kl} x_l(t - \Delta_{kl}), \tag{45}$$

where Δ_{kl} is 0 if the connection from unit l to unit k is a feedforward connection within the output stage and 1 if it is a feedback connection. Singling out the first sum on the right-hand side of this equation, we define

$$s_k^*(t) = \sum_{l \in U_H \cup I} w_{kl} x_l(t). \tag{46}$$

It then follows that

$$p_{ij}^k(t) = f_k'(s_k(t)) \frac{\partial s_k^*(t)}{\partial w_{ij}} + f_k'(s_k(t)) \left[\sum_{l \in U_O} w_{kl} p_{ij}^l(t) + \delta_{ik} x_j(t - \Delta_{ij}) \right]. \tag{47}$$

If $i \in U_O$, the first term on the right-hand side of this equation is zero and the updating of p_{ij}^k thus proceeds using a pure RTRL approach. That is, for k and i in U_O, p_{ij}^k is updated by means of the equation

$$p_{ij}^k(t) = f_k'(s_k(t)) \left[\sum_{l \in U} w_{kl} p_{ij}^l(t) + \delta_{ik} x_j(t - \Delta_{ij}) \right]. \tag{48}$$

If $i \in U_H$, however, the first term on the right-hand side of Equation 47 is not necessarily zero, but it can be computed by injecting a 1 as "error" at the output of the kth unit and back-propagating directly into the hidden stage to the point where δ_i is computed. This backpropagation computation begins at the output of the kth unit and proceeds directly into the hidden stage, ignoring all connections to the kth unit from units in the output stage. Specifically, then, for each fixed $k \in U_O$, one such back-propagation pass is performed to obtain a set of $\delta_i(t)$ values for all $i \in U_H$. Then the p_{ij}^k values for this particular k are updated using

$$p_{ij}^k(t) = \delta_i(t) x_j(t) + f_k'(s_k(t)) \left[\sum_{l \in U_O} w_{kl} p_{ij}^l(t) + \delta_{ik} x_j(t - \Delta_{ij}) \right]. \tag{49}$$

One special case of this architecture is a network having a single self-recurrent unit as its only output unit, with a feedforward network serving as a preprocessing stage. In this case, there is a single value of p_{ij}^k to associate with each weight w_{ij}, and we may imagine that it is stored with its corresponding weight. Then only local communication is required to update these p values, and a single global broadcast of the error $e_k(t)$ (where k is the index of the output unit) is sufficient to allow error gradient computation. This may be viewed as a generalization of the single self-recurrent unit architecture studied by Bachrach (1988). One of the algorithms he investigated coincides with that described here.

APPROXIMATION STRATEGIES

Up to this point we have confined our attention to exact gradient computation algorithms. However, it is often useful to consider algorithms which omit part of the computation required to fully compute the exact gradient. There are actually several reasons why this can be advantageous, some of which we discuss later. The primary reason is to simplify the computational requirements.

Truncated Back Propagation through Time

A natural approximation to the full real-time BPTT computation is obtained by truncating the backward propagation of information to a fixed number of prior time steps. This is, in general, only a heuristic technique because it ignores

dependencies in the network spanning durations longer than this fixed number of time steps. Nevertheless, in those situations where the actual back-propagation computation leads to exponential decay in strength through (backward) time, which occurs in networks whose dynamics consist of settling to fixed points, this can give a reasonable approximation to the true error gradient. Even when this is not the case, its use may still be justified when weights are adjusted as the network runs simply because the computation of the "exact" gradient over a long period of time may be misleading since it is based on the assumption that the weights are constant. We call this algorithm *truncated back propagation through time*. With h representing the number of prior time steps saved, this algorithm will be denoted BPTT(h). Note that the discrepancy between the BPTT(h) result and the BPTT(∞) result is equal to the first sum on the right-hand side of Equation 37 for the FP/BPTT(h) algorithm. The processing performed by the BPTT(h) algorithm is depicted in Figure 9.

The computational complexity of this algorithm is quite reasonable as long as h is small. Its space complexity is in $\Theta((m + n)h)$ and the average number of arithmetic operations required per time step is in $\Theta((w_U + w_A)h/\Delta_T)$. The worst case for this algorithm for any fixed n is when the network is fully connected, all

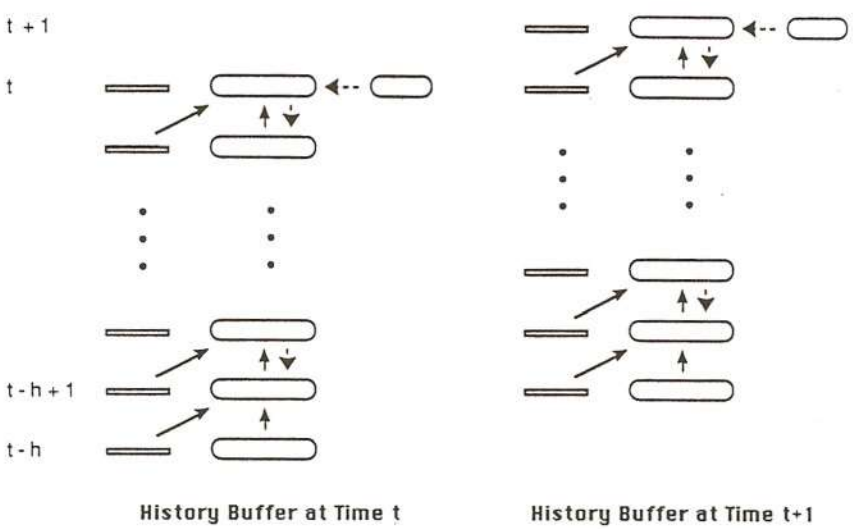

Figure 9. A schematic representation of the storage and processing required for the BPTT(h) algorithm for two consecutive executions of the error gradient computation, one at time step t and the next at time step $t + 1$. The history buffer always contains the current network input, activity, and target values, along with the values of network input and activity for the h prior time steps. The BPTT computation requires injection of error only for the current time step and is performed anew at each subsequent time step.

weights are adaptable, and target values are supplied at every time step, so that $\Delta_T = 1$. In this case the algorithm requires $\Theta(nh)$ space and $\Theta(n^2h)$ time. These complexity results are summarized in Tables 1 and 2.

A number of researchers (Watrous & Shastri, 1986; Elman, 1988; Cleeremans, Servan-Schreiber, & McClelland, 1989, chapter 9, this volume) have performed experimental studies of learning algorithms based on this approximate gradient computation algorithm. The architecture studied by Elman and by Cleeremans et al. is an example of the two-stage type described earlier with hidden-to-hidden feedback only, but the learning algorithm used in the recurrent hidden stage is BPTT(1).

A More Efficient Version of Truncated Back Propagation through Time

Interestingly, it is possible to devise a more efficient approximate gradient computation algorithm for continually operating networks by combining aspects of epochwise BPTT with the truncated BPTT approach, as has been noted in Williams, (1989). Note that in the truncated BPTT algorithm, BPTT through the most recent h time steps is performed anew each time the network is run through an additional time step. More generally, one may consider letting the network run through h' additional time steps before performing the next BPTT computation. In this case, if t represents a time at which BPTT is to be performed, the algorithm computes an approximation to $\nabla_W J^{\text{total}}(t - h', t)$ by taking into account only that part of the history over the interval $[t - h, t]$. Let us denote this algorithm BPTT(h; h'). Thus BPTT(h) is the same as BPTT(h; 1), and BPTT(h; h) is the epochwise BPTT algorithm, which, of course, is not an exact gradient algorithm unless there are state resets at the appropriate times. Figure 10 depicts the processing performed by the BPTT(h; h') algorithm.

In general, whenever it can be assumed that backpropagating through the most recent $h - h' + 1$ time steps gives a reasonably close approximation to the result that would be obtained from back-propagating all the way back to t_0, then this algorithm should be sufficient. The storage requirements of this algorithm are essentially the same as those of BPTT(h), but, because it computes the cumulative error gradient by means of BPTT only once every h' time steps, its average time complexity per time step is reduced by a factor of h'. Thus, its average time complexity per time step is in $\Theta((w_U + w_A)h/h')$ in general and in $\Theta(n^2h/h')$ in the worst case, as indicated in Tables 1 and 2. In particular, when h' is some fixed fraction of h, the worst-case time complexity per time step for this algorithm is in $\Theta(n^2)$. Furthermore, it is clear that making h/h' small makes the algorithm more efficient. Thus, a practical approximate gradient computation algorithm for continually operating networks may be obtained by choosing h and h' so that $h - h'$ is large enough that a reasonable approximation to the true gradient is obtained and so that h/h' is reasonably close to 1.

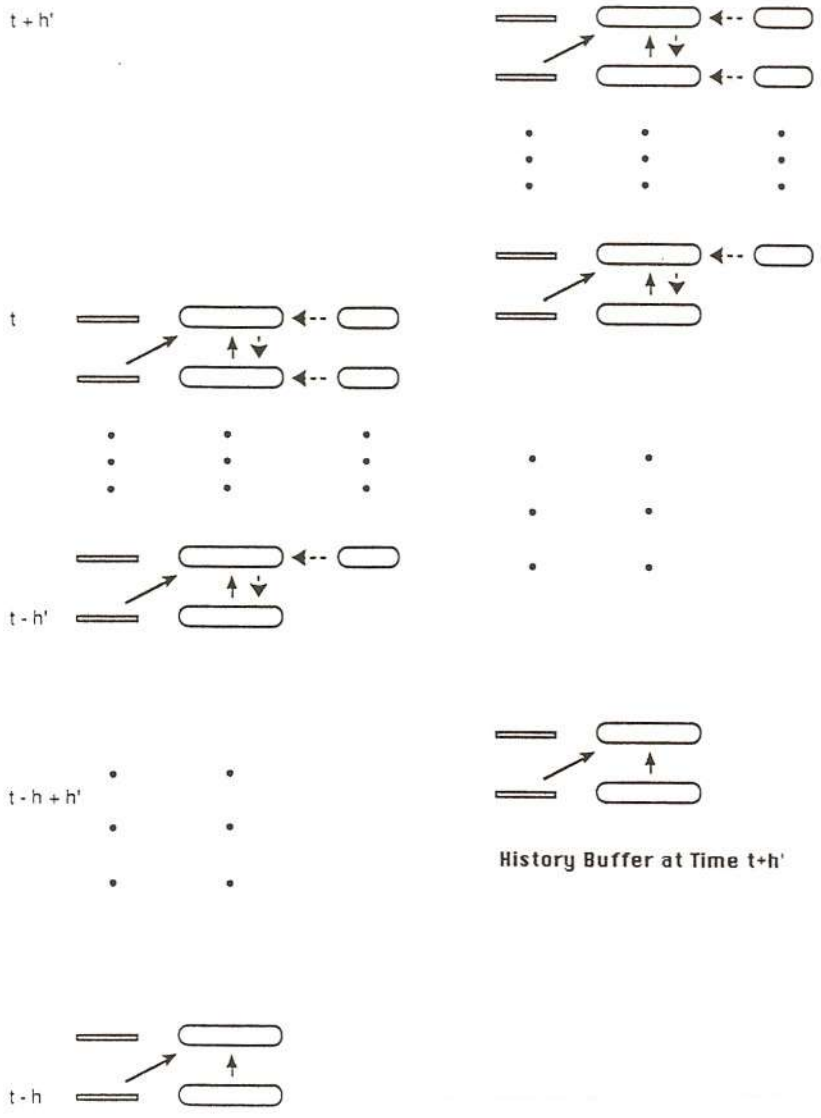

Figure 10. A schematic representation of the storage and processing required for the BPTT(h; h') algorithm for two consecutive executions of the error gradient computation, one at time step t and the next at time step $t + h'$. The history buffer always contains the values of the network input and activity for the current time step as well as for the h prior time steps. It also contains target values for the most recent h' time steps, including the current time step. The BPTT computation thus requires the injection of error only at the h' uppermost levels in the buffer. This figure illustrates a case where $h' < h/2$, but it is also possible to have $h' \geq h/2$.

Subgrouping in Real-Time Recurrent Learning

The RTRL approach suggests another approximation strategy which is designed to reduce the complexity of the computation and which also has some intuitive justification. While truncated BPTT achieves a simplification by ignoring long-term *temporal* dependencies in the network's operation, this modification to RTRL, proposed in Zipser (1989), achieves its simplification by ignoring certain *structural* dependencies in the network's operation.

This simplification is obtained by viewing a recurrent network for the purpose of learning as consisting of a set of smaller recurrent networks all connected together. Connections within each subnet are regarded as the recurrent connections for learning, while activity flowing between subnets is treated as external input by the subnet which receives it. The overall physical connectivity of the network remains the same, but now forward gradient propagation is only performed within the subnets. Note that this means that each subnet must have at least one unit which is given target values.

More precisely, in this approach the original network is regarded as divided into g equal-sized subnetworks, each containing n/g units (assuming that n is a multiple of g, as we will throughout this discussion). Each of these subnetworks needs to have at least one target, but the way the targets are distributed among the subnetworks is not germane at this point. Then Equations 33 and 32 of the RTRL algorithm are used to update the p_{ij}^k values and determine the appropriate error gradient, except that the value of p_{ij}^k is regarded as being fixed at zero whenever units i and k belong to different subnetworks. If we regard each weight w_{ij} as belonging to the subnetwork to which unit i belongs, this amounts to ignoring $\partial y_k / \partial w_{ij}$ whenever the kth unit and weight w_{ij} belong to different subnets. The computational effect is that RTRL is applied to g decoupled subnetworks, each containing n/g units. We denote this algorithm RTRL(g). Clearly, RTRL(1) is the same as RTRL. Figure 11 illustrates how RTRL is simplified by using the subgrouping strategy.

The number of nonzero p_{ij}^k values to be stored and updated for this algorithm is nw_A/g. To analyze its time requirements, we assume for simplicity that every subnetwork has the same number of adjustable weights and that every unit receives input from the same number of units, which implies that each subnetwork then contains w_A/g adjustable weights and w_U/g^2 within-group weights. But then Equation 33 for updating the p_{ij}^k values requires $\Theta((w_U/g^2)(w_A/g))$ operations within each subnetwork on each time step, or a total of $\Theta(w_U w_A/g^2)$ operations on each time step. In addition, the average number of operations required for Equation 32 per time step is $n_T w_A/g$. Altogether, then, the time complexity of this algorithm per time step is in $\Theta(w_U w_A/g^2 + n_T w_A/g)$.

To examine the worst-case complexity, assume that the network is fully connected, all weights are adaptable, and n_T is in $\Theta(n)$. In this case RTRL(g) has space complexity in $\Theta(n^3/g)$ and average time complexity per time step in $\Theta(n^4/g^2 + n^3/g) = \Theta(n^4/g^2)$ (since $g \leq n$). In particular, note that if g is

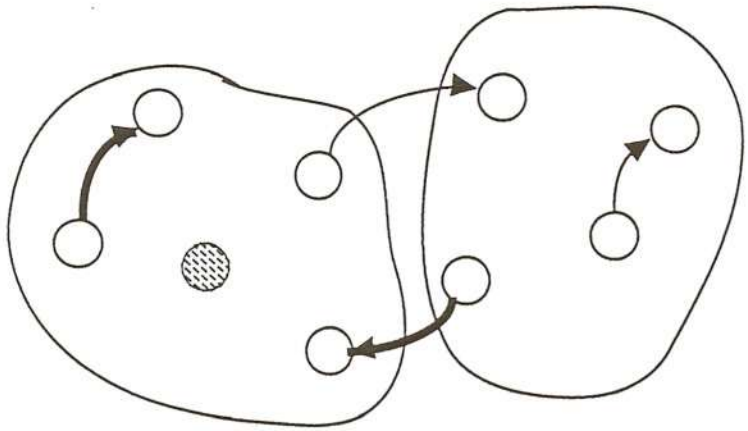

Figure 11. A network divided into two subnetworks for subgrouped RTRL. The full RTRL algorithm requires keeping track of the sensitivity of each unit in the network with respect to each weight in the network. When subgrouping is used, each unit only pays attention to its sensitivity to weights on connections terminating in the group to which it belongs. Thus, among the four connections shown, only those two indicated with the heavy lines are considered when computing the sensitivity of the unit indicated by the shading to variations in the weights.

increased in proportion to n, which keeps the size of the subnets constant, the resulting algorithm has, in the worst case, space and time complexity per time step both in $\Theta(n^2)$. These complexity results are summarized in Tables 1 and 2.

One strategy which avoids the need for assigning specific target values to units from each subgroup is to add a separate layer of output units with 0-delay connections from the entire recurrent network to these output units, which are the only units given targets. This is then an example of a two-stage architecture having only hidden-to-hidden recurrence, and the training method described earlier for such networks, involving both back propagation and RTRL, can be modified so that the full RTRL is replaced by subgrouped RTRL. This approach amounts to giving the recurrent network virtual targets by means of back propagation from the output units.

Note also that this subgrouping strategy could be used to advantage in the hybrid algorithm FP/BPTT(h). Such an approximation algorithm would provide an interesting blend of aspects of both truncated BPTT and subgrouped RTRL.

TEACHER FORCING

An interesting strategy that has appeared implicitly or explicitly in the work of a number of investigators studying supervised learning tasks for recurrent nets

(Doya & Yoshizawa, 1989; Jordan, 1986; Narendra & Parthasarathy, 1990; Pineda, 1988; Rohwer & Renals, 1989; Williams & Zipser, 1989a; 1989b) is to replace, during training, 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 target value exists. We call this intuitively sensible technique *teacher forcing*.

Formally, the dynamics of a teacher-forced-network during training are given by Equations 2 and 3, as before, but where $\mathbf{x}(t)$ is now defined by

$$x_k(t) = \begin{cases} x_k^{\text{net}}(t) & \text{if } k \in I, \\ d_k(t) & \text{if } k \in T(t), \\ y_k(t) & \text{if } k \in U \backslash T(t) \end{cases} \tag{50}$$

rather than by Equation 1. Because $\partial d_k(t)/\partial w_{ij} = 0$ for all $k \in T(t)$ and for all t, this leads to very slight differences in the resulting gradient computations, giving rise to slightly altered algorithms. It is an easy exercise to rework the computations given earlier for BPTT and RTRL using these modified dynamics. We omit the details and content ourselves here with a description of the results.

The one simple change necessary to incorporate teacher forcing into any version of BPTT is that the back-propagation computation from later times must be "blocked" at any unit in the unrolled network whose output has been set to a target value. Equivalently, any unit given an external target value at a particular time step should be given no virtual error for that time step. More precisely, for real-time BPTT or any of its variants, Equation 14 must be replaced by

$$\epsilon_k(\tau - 1) = 0 \tag{51}$$

whenever $k \in T(\tau - 1)$ for any $\tau \leq t$. Similarly, for epochwise BPTT, Equation 19 must be replaced by

$$\epsilon_k(\tau - 1) = e_k(\tau - 1) \tag{52}$$

whenever $k \in T(\tau - 1)$ for any $\tau \leq t_1$.

In the case of RTRL, the one simple change required to accommodate teacher forcing is to treat the value of $p_{ij}^l(t)$ as zero for any $l \in T(t)$ when computing $p_{ij}^k(t + 1)$ via Equation 33. Equivalently, Equation 33 is replaced by

$$p_{ij}^k(t + 1) = f_k'(s_k(t)) \left[\sum_{l \in U \backslash T(t)} w_{kl} p_{ij}^l(t) + \delta_{ik} x_j(t) \right]. \tag{53}$$

There seem to be several ways that teacher forcing can be useful. For one thing, one might expect that teacher forcing could lead to faster learning because it enables learning to proceed on what amounts to the assumption that the network is performing all earlier parts of its task correctly. In this way, all learning effort is focused on the problem of performing correctly at a particular time step given that the performance is correct on all earlier time steps. When teacher forcing provides this benefit, one would expect that its absence would simply

slow down learning but not prevent it altogether. It may also play a useful, or even critical, role in situations where there is some approximation involved. For example, when using subgrouping in RTRL, it has sometimes been found to make the difference between success and failure.

Beyond these potential benefits of teacher forcing is what we now recognize as its sometimes essential role in the training of continually operating networks. One such situation we have studied involves training networks to oscillate autonomously using RTRL. If the network starts with small enough weights, its dynamical behavior will consist of settling to a single point attractor from any starting state. Furthermore, assuming that the learning rate is reasonably small, it will eventually converge to its point attractor regardless of where it was started. Once it has stayed at this attractor sufficiently long the task can never be learned by moving along the negative error gradient in weight space because this error gradient information only indicates what direction to move to alter the fixed point, not what direction would change the overall dynamical properties. This is the same phenomenon described earlier in our discussion of the relationship between BPTT and the recurrent back-propagation algorithm for training settling networks. The gradient of error occurring long after the transient portion has passed contains no information about the overall dynamics of the network. Applying BPTT or RTRL to such a network is then equivalent to applying RBP; the only effect is that the point attractor is moved around. A network being trained to oscillate will thus simply adjust its weights to find the minimum error between its constant output and the desired oscillatory trajectory without ever becoming an oscillator itself.

We believe that this is a particular case of a much more general problem 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 moving arbitrarily close to zero over time). The information lost when the network has fallen into its attractor includes information which might tell the weights where to move to perform the desired task. As long as the network is moving along a transient, there is some gradient information which can indicate the desired direction in which to change the weights; once the network reaches its steady-state behavior, this information disappears.

Another example of this justification for the use of teacher forcing is provided by the work of Pineda (1988; chapter 4, this volume), who has combined it with RBP as a means of attempting to add new stable points to an associative memory network. Without teacher forcing, RBP would just move existing stable point around without ever creating new ones.

Still another class of examples where teacher forcing is obviously important is where the weights are correct to perform the desired task but the network is currently operating in the wrong region of its state space. For example, consider a network having several point attractors which happens to be currently sitting on the wrong attractor. Attempting to get it onto the right attractor by adjusting the weights alone is clearly the wrong strategy. A similar case is an oscillator network faced with a teacher signal essentially identical to its output except for being 180° out of phase. Simulation of such problems using RTRL without teacher forcing leads to the result that the network stops oscillating and produces constant output equal to the mean value of the teacher signal. In contrast, teacher forcing provides a momentary phase reset which avoids this problem.

The usefulness of teacher forcing in these situations is obviously related to the idea that both the network weights and initial conditions determine the behavior of the network at any given time. Error gradient information in these learning algorithms allows control over the network weights, but one must also gain control over the initial conditions, in some sense. By using desired values to partially reset the state of the net at the current time one is helping to control the initial conditions for the subsequent dynamics.

There are situations for which teacher forcing is clearly not applicable or may be otherwise inappropriate. It is certainly not applicable when the units to be trained do not feed their output back to the network, as in one of the special two-stage architectures discussed earlier. Furthermore, a gradient algorithm using teacher forcing is actually optimizing a different error measure than its unforced counterpart, although any setting of weights giving zero error for one also gives zero error for the other. This means that, unless zero error is obtained, the two versions of a gradient algorithm need not give rise to the same solutions. 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 minimum-error solution for the unforced network.

A simple solution is the problem of attempting to train a single unit to perform a sequence consisting of n zeros alternating with n ones. It is not hard to see that when $n \geq 2$ the best least-squares fit to this training data is achieved when the unit produces the constant output 0.5 at all times. This is the behavior to which a gradient algorithm will essentially converge for this problem if teacher forcing is not used. Such a solution is achieved by setting the unit's bias and recurrent weight to zero. Note that this actually makes 0.5 a global attractor for this dynamical system; if the output were somehow perturbed to some other value momentarily, it would converge back to 0.5 (in one time step, in this case).

However, when teacher forcing is used, the behavior tends toward creating point attractors for the output of the unit at $1/n$ and $1 - 1/n$. When $n = 2$ this is identical to the solution obtained without teacher forcing, but for $n \geq 3$ it is quite different. When $n \geq 3$, the weights obtained using teacher forcing lead to bistable behavior, with an output of 0.5 representing an unstable critical point separating the two basins of attraction for the system.

Teacher forcing leads to such a result because it emphasizes *transitions* in the training data. According to the training data, a correct output of either 0 or 1 is followed by that same value $1 - 1/n$ of the time and by the opposite value $1/n$ of the time; the result obtained using teacher forcing simply represents the mini-

mum mean-square error for such transition data. In this particular problem only the transitions between successive output values are relevant because there are no other state variables potentially available to record the effect of earlier output values. More generally, teacher forcing attempts to fit transitions from the collections of all prior correct output values to the next correct output value, subject to the ability of the net to capture the relevant distinctions in its state of activity.

Pineda (1980; chapter 4, this volume) has pointed out some other potential problems with teacher forcing. One of these is that it may create trajectories which are not attactors but repellers. One potential way around this and other difficulties with teacher forcing is to consider a slight generalization in which $x_k(t)$ is set equal to $y_k(t) + \beta e_k(t)$ for $k \in U$, where $\beta \in [0, 1]$ is a constant. Teacher forcing uses $\beta = 1$ while $\beta = 0$ represents its absence. But other values of β represent a mix of the two strategies. For this generalization, the correct gradient computation involves attenuating the virtual error back-propagated from later times by the factor $1 - \beta$ in BPTT or multiplying $p_{ij}^l(t)$ by $1 - \beta$ before propagating the activity gradient forward in RTRL. A related strategy is to use teacher forcing intermittently rather than on every time step when target values are available. This has been tested by Tsung (1990) and found useful for dealing with the somewhat different but related problem of training network trajectories that vary extremely slowly.

Finally, we note that Rohwer (1990) has expanded on this idea of teacher forcing to develop an interesting new epochwise learning algorithm based on computation of the gradient of performance with respect to unit activities rather than network weights.

EXPERIMENTAL STUDIES

The important question to be addressed in studies of recurrent network learning algorithms, whatever the constraints to which they must conform, is how much total computational effort must be expended to achieve the desired performance. For many of the algorithms described here an analysis of the amount of computation required per time step has been presented, but this must be combined with knowledge of the number of time steps required and success rate obtained when training particular networks to perform particular tasks. Any speed gain from performing a simplified computation on each time step is of little interest unless it allows successful training without inordinately prolonging the training time.

To examine the relative performance of some of the more computationally attractive approximation algorithms for continually operating networks described here, both subgrouped RTRL and truncated BPTT were tested for their ability to train fully recurrent networks to emulate the finite state machine part of a Turing machine for balancing parentheses, a task that had previously been shown to be learnable by RTRL (Williams & Zipser, 1989b). For this task the network receives as input the same tape mark that the Turing machine "sees," and is trained to produce the same outputs as the Turing machine for each cell of the tape that it visits. There are four output lines in the version of the problem used here. They code for the direction of movement, the character to be written on the tape, and whether a balanced or unbalanced final state has been reached. It had previously been found that a fully recurrent network with 12 units was the smallest that learned the Turing machine task. Although this could be formulated as an epochwise task by resetting the network every time the Turing machine halts and begins anew, the network was allowed to run continually, with transitions from a halt state to the start state being considered part of the state transition structure which the network had to infer.

To test the subgrouping strategy on this task, a 12-unit fully connected network was divided for learning into four subnets of 3 units each, with one unit in each subnet designated as an output unit. The full RTRL algorithm allowed the network to learn the task with or without teacher forcing about 50% of the time after seeing fewer than 100,000 cells of the Turing machine tape. The RTRL(4) algorithm also allowed the network to learn the task about 50% of the time in fewer than 100,000 Turing machine cycles, but only in the teacher forcing mode. The subdivided network never learned the task without teacher forcing.

To test the truncation strategy on this task, BPTT(h) was tried, with various values of h.[11] No teacher forcing was used. It was found that with $h \leq 4$, BPTT(h) was successful in training the network only about 9% of the time, while BPTT(9) succeeded more than 80% of the time. The fact that BPTT(9) succeeded more often than the various RTRL algorithms, including the version with no subgrouping, may indicate that the error committed in computing an exact gradient as if the weights had been constant throughout the past may outweigh the error committed by discarding all effects of activity and input in the distant past. On the other hand, it might also represent a beneficial effect of failing to follow the exact gradient and thereby avoiding becoming trapped at a local optimum.

The relative actual running times of the these algorithms on a single-processor machine were also compared. It was found that BPTT(9) ran 28 times faster on this task than RTRL, while RTRL(4) ran 9.8 times faster than RTRL.

In another set of studies (Williams & Peng, 1990), BPTT(16;8) was found to succeed as often as BPTT(9) on this task, while running twice as fast.[12] Note that BPTT(16;8) is thus well over 50 times faster than RTRL on this task.

[11] For these studies the variant in which past weight values are stored in the history buffer was used.

[12] Careful analysis of the computational requirements of BPTT(9) and BPTT(16;8), taking into account the fixed overhead of running the network in the forward direction that must be borne by any algorithm, would suggest that one should expect about a factor of 4 speedup when using BPTT(16;8). Because this particular task has targets only on every other time step, the use of BPTT(9) here really amounts to using BPTT(9;2), which therefore reduces the speed gain by essentially one factor of 2.

TABLE 1
Order of Magnitude of Space and Time Requirements for the Various General-Purpose Algorithms Discussed Here

Algorithm	*Space*	*Average Time Per Time Step*
Epochwise BPTT	$\Theta((m + n)h)$	$\Theta(w_U + w_A)$
BPTT (∞)	$\Theta((m + n)L)$	$\Theta((w_U + w_A)L/\Delta_T)$
RTRL	$\Theta(nw_A)$	$\Theta(w_U w_A)$
FP/BPTT(h)	$\Theta(nw_A + (m + n)h)$	$\Theta(nw_U + nw_A + n^2 w_A/h)$
FP/BPTT(cn)	$\Theta(nw_A + cn(m + n))$	$\Theta(nw_U + nw_A + nw_A/c)$
BPTT(h)	$\Theta((m + n)h)$	$\Theta((w_U + w_A)h/\Delta_T)$
BPTT($h;h'$)	$\Theta((m + n)h)$	$\Theta((w_U + w_A)h/h')$
BPTT($h;ch$)	$\Theta((m + n)h)$	$\Theta(w_U + w_A)$
RTRL(g)	$\Theta(nw_A/g)$	$\Theta(w_U w_A/g^2 + n_T w_A/g)$
RTRL(cn)	$\Theta(w_A)$	$\Theta(w_U w_A/cn^2 + n_T w_A/n)$

Here c denotes a constant and the meaning of all the other symbols used is summarized in the third section. For the variant of BPTT(h) in which past weight values are saved, the space requirements are in $\Theta(w_A h)$.

TABLE 2
Worst-Case Complexity for the Various General-Purpose Algorithms Discussed Here Expressed in Terms of the Number of Units n

Algorithm	*Space*	*Average Time Per Time Step*
Epochwise BPTT	$\Theta(nh)$	$\Theta(n^2)$
BPTT(∞)	$\Theta(nL)$	$\Theta(n^2 L)$
RTRL	$\Theta(n^3)$	$\Theta(n^4)$
FP/BPTT(h)	$\Theta(n^3 + nh)$	$\Theta(n^3 + n^4/h)$
FP/BPTT(cn)	$\Theta(n^3)$	$\Theta(n^3)$
BPTT(h)	$\Theta(nh)$	$\Theta(n^2 h)$
BPTT($h;h'$)	$\Theta(nh)$	$\Theta(n^2 h/h')$
BPTT($h;ch$)	$\Theta(nh)$	$\Theta(n^2)$
RTRL(g)	$\Theta(n^3/g)$	$\Theta(n^4/g^2)$
RTRL(cn)	$\Theta(n^2)$	$\Theta(n^2)$

These results are based on the assumption that m, the number of input lines, is in $O(n)$. Here c denotes a constant. For the variant of BPTT(h) in which past weight values are saved, the worst-case space requirements are in $\Theta(n^2 h)$.

DISCUSSION

In this chapter we have described a number of gradient-based learning algorithms for recurrent networks, all based on two different approaches to computing the gradient of network error in weight space. The existence of these various techniques, some of them quite reasonable in terms of their computational requirements, should make possible much more widespread investigation of the capabilities of recurrent networks.

In the introduction we noted that investigators studying learning algorithms for such networks might have various objectives, each of which might imply different constraints on which algorithms might be considered to meet these objectives. Among the possible constraints one might wish to impose on a learning algorithm are biological plausibility and locality of communication. Feedforward back propagation is generally regarded as biologically implausible, but its requirement for reverse communication along only the connections already in place allows it to be considered a locally implementable algorithm, in the sense that it does not require a great deal of additional machinery beyond the network itself to allow implementation of the algorithm. Except in very restricted cases involving severely limited architectures or extreme approximations, the algorithms described here cannot be considered biologically plausible as learning algorithms for real neural networks, nor do they enjoy the locality of feeforward back propagation.

However, many of the algorithms discussed here can be implemented quite reasonably and efficiently in either vector parallel hardware or special-purpose parallel hardware designed around the storage and communication requirements of the particular algorithm. Several of these algorithms are quite well suited for efficient serial implementation as well. Thus one might expect to see these algorithms used especially for off-line development of networks having desired temporal behaviors in order to study the properties of these networks. Some of these techniques have already been used successfully to fit models of biological neural subsystems to data on the temporal patterns they generate (Arnold & Robinson, 1989; Lockery, Fang, & Sejnowski, 1990; Tsung, Cottrell, & Selverston, 1990; Anastasio, 1991) and a number of studies have been undertaken to apply these methods to develop networks which carry out various language processing or motor control tasks as a means of understanding the information processing strategies involved (Elman, 1988; Jordan, 1986; Mozer, 1989, chapter 5, this volume; Cleeremans, Servan-Screiber, and McClelland, 1989, chapter 9, this volume; Smith & Zipser, 1990). One might also expect to see specific engineering applications of recurrent networks developed by these methods as well.

Thus there is much that can be done with the currently available algorithms for training recurrent networks, but there remains a great deal of room for further development of such algorithms. It is already clear that more locally implement-

able or biologically plausible algorithms remain to be found, and algorithms with improved overall learning times are always desirable. It seems reasonable to conjecture that such algorithms will have to be more architecture specific or task specific than the general-purpose algorithms studied here.

Of particular importance are learning algorithms for continually operating networks. Here we have described both "exact" and approximate gradient algorithms for training such networks. However, by our definition, the exact algorithms compute the true gradient at the current value of the weights only under the assumption that the weights are held fixed, which cannot be true in a continually operating learning network. This problem need not occur in a network which operates epochwise; when weight changes are only performed between epochs, an exact gradient algorithm can compute the true gradient of some appropriate quantity.

Thus all the algorithms described here for continually operating networks are only capable of computing approximate gradient information to help guide the weight updates. The degree of approximation involved with the "exact" algorithms depends on the degree to which past history of network operation influences the gradient computation and the degree to which the weights have changed in the recent past. Truncated BPTT alleviates this particular problem because it ignores all past contributions to the gradient beyond a certain distance into the past. Such information is also present in RTRL, albeit implicitly, and Gherrity (1989) has specifically addressed this issue by incorporating into his continuous-time version of RTRL an exponential decay on the contributions from past times. For the discrete-time RTRL algorithm described here, this is easily implemented by multiplying all the p_{ij}^k values by an attenuation factor less than 1 before computing their updated values. Unlike truncated BPTT, however, this does not reduce the computational complexity of the algorithm.

Another way to attempt to alleviate this problem is to use a very low learning rate. The effect of this is make the constant-weight approximation more accurate, although it may slow learning. One way to view this issue is in terms of time scales, as noted by Pineda (chapter 4, this volume). The accuracy of the gradient computation provided by an exact algorithm in our sense depends on the extent to which the time scale of the learning process is decoupled from the time scale of the network's operation by being much slower. In general, with the learning rate set to provide sufficiently fast learning, these time scales may overlap. This can result in overall dynamical behavior which is determined by a combination of the dynamics of the network activation and the dynamics of the weight changes brought about by the learning algorithm. At this point one leaves the realm of gradient-based learning algorithms and enters a realm in which a more general control-theoretic formulation is more appropriate. A particular issue here of some importance is the overall stability of such a system, as emphasized in the theory of *adaptive control* (Narendra & Annaswamy, 1989). It is to be expected that satisfactory application of the techniques described here to situations requiring on-line adaptation of continually operating recurrent networks will depend on gaining further understanding of these questions.

It is useful to recognize the close relationship between some of the techniques discussed here and certain approaches which are well known in the engineering literature. In particular, the specific backward error propagation and forward gradient propagation techniques which we have used here as the basis for all the algorithms investigated turn out to have their roots in standard optimal-control-theoretic formulations dating back to the 1960s. For example, leCun (1988) has pointed to the work of Bryson and Ho (1969) in optimal control theory as containing a description of what can now be recognized as error back propagation when applied to multilayer networks. Furthermore, it is also clear that work in that tradition also contains the essential elements of the back-propagation-through-time approach. The idea of back-propagating through time, at least for a linear system, amounts to running forward in time what is called in that literature the *adjoint system*. The two-point boundary-value problems discussed in the optimal control literature arise from such considerations. Furthermore, the idea of propagating gradient information forward in time, used as the basis for RTRL, was proposed by McBride and Narendra (1965), who also noted that use of the adjoint system may be preferable when on-line computation is not required because of its lower computational requirements. The teacher forcing technique has its counterpart in engineering circles as well. For example, it appears in the adaptive signal processing literature as an "equation error" technique for synthesizing linear filters having an infinite impulse response (Widrow & Stearns, 1985).

In work very similar in spirit to that we have presented here, Piche (1994) has shown how various forms of back propagation through time and forward gradient computation may be derived in a unified manner from a standard Euler-Lagrange optimal-control-theoretic formulation. Furthermore, he also discusses the computational complexity of the various algorithms described. Included among the algorithms covered by his analysis are some of those we have described in Section 7 for special architectures.

Finally, we remark that the techniques we have discussed here are far from being the only ones available for creating networks having certain desired properties. We have focused here specifically on those techniques which are based on computation of the error gradient in weight space, with particular emphasis on methods appropriate for continually operating networks. As described earlier in the discussion of the teacher forcing technique, Rohwer (1990) has proposed an epochwise approach based on computation of the error gradient with respect to unit activities rather than network weights. Also, another body of techniques has been developed by Baird (1989) for synthesizing networks having prescribed dynamical properties. Unlike the algorithms discussed here, which are designed to gradually perturb the behavior of the network toward the target behavior as it runs, these algorithms are intended to be used to "program in" the desired

dynamics at the outset. Another difference is that these techniques are currently limited to creating networks for which external input must be in the form of momentary state perturbations rather than more general time-varying forcing functions.

ACKNOWLEDGMENT

R. J. Williams was supported by Grant IRI-8703566 from the National Science Foundation. D. Zipser was supported by Grant I-R01-M445271-01 from the National Institute of Mental Health and grants from the System Development Foundation.

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. II* (pp. 609–618).

Anastasio, T. J. (1991). Neural network models of velocity storage in the horizontal vestibuloocular reflex. *Biological Cybernetics, 64,* 187–196.

Arnold, D. & Robinson, D. A. (1989). A learning neural-network model of the oculomotor integrator. *Society of Neuroscience Abstracts, 15,* part 2, 1049.

Bachrach, J. (1988). *Learning to represent state.* Unpublished master's thesis. University of Massachusetts, Amherst, Department of Computer and Information Science.

Baird, B. (1989). A bifurcation theory approach to vector field programming for periodic attractors. *Proceedings of the International Joint Conference on Neural Networks. I* (pp. 381–388).

Bryson, A. E., Jr. & Ho, Y-C. (1969). *Applied optimal control.* New York: Blaisdell.

Cleeremans, A., Servan-Schreiber, D., & McClelland, J. L. (1989). Finite-state automata and simple recurrent networks. *Neural Computation, 1,* 372–381.

Doya, K. & Yoshizawa, S. (1989). Adaptive neural oscillator using continuous-time back-propagation learning. *Neural Networks, 2,* 375–385.

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

Gherrity, M. (1989). A learning algorithm for analog, fully recurrent neural networks. *Proceedings of the International Joint Conference on Neural Networks. I* (pp. 643–644).

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).

Kuhn, G. (1987). *A first look at phonetic discrimination using a connectionist network with recurrent links* (SCIMP Working Paper No. 4/87). Princeton, NJ: Communications Research Division, Institute for Defense Analyses.

leCun, Y. (1988). *A theoretical framework for back-propagation* (Technical Report CRG-TR-88-6). Toronto: University of Toronto, Department of Computer Science.

Lockery, S., Fang, Y., & Sejnowski, T. (1990). Neural network analysis of distributed representations of dynamical sensory-motor transformations in the leech. In *Advances in neural information processing systems, 2.* San Mateo, CA: Morgan Kaufmann.

McBride, L. E., Jr., & Narendra, K. S. (1965). Optimization of time-varying systems. *IEEE Transactions on Automatic Control, 10,* 289–294.

Mozer, M. C. (1989). A focused back-propagation algorithm for temporal pattern recognition. *Complex Systems, 3,* 349–381.

Narendra, K. S., & Annaswamy, A. M. (1989). *Stable adaptive systems.* Englewood Cliffs, NJ: Prentice-Hall.

Narendra, K. S., & Parthasarathy, K. (1990). Identification and control of dynamic systems using neural networks. *IEEE Transactions on Neural Networks, 1,*4–27.

Pearlmutter, B. A. (1989). Learning state space trajectories in recurrent neural networks. *Neural Computation, 1,* 263–269.

Piche, S. W. (1994). Steepest descent algorithms for neural network controllers and filters. *IEEE Transactions on Neural Networks, 5.*

Pineda, F. J. (1987). Generalization of backpropagation to recurrent neural networks. *Physical Review Letters, 18,* 2229–2232.

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

Pineda, F. J. (1989). Recurrent backpropagation and the dynamical approach to adaptive neural computation. *Neural Computation, 1,* 161–172.

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

Rohwer, R. (1990). The "moving targets" training algorithm. In L. B. Almeida & C. J. Wellekens (Eds.), *Proceedings of the EURASIP Workshop on Neural Networks,* Sesimbra, Portugal. *Lecture Notes in Computer Science* (vol. 412, p. 100). New York: Springer-Verlag.

Rohwer, R., & Renals, S. (1989). Training recurrent networks. In L. Personnaz & G. Dreyfus (Eds.), *Neural networks from models to applications.* Paris: I.D.E.S.T.

Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). Learning internal representations by error propagation. In D. E. Rumelhart, J. L. McClelland, & the PDP Research Group, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition. Vol. 1: Foundations.* Cambridge: MIT Press/Bradford Books.

Sato, M. (1990a). A real time learning algorithm for recurrent analog neural networks. *Biological Cybernetics, 62,* 237–241.

Sato, M. (1990b). A learning algorithm to teach spatiotemporal patterns to recurrent neural networks. *Biological Cybenertics, 62,* 259–263.

Schmidhuber, J. (1992). A fixed size storage $O(n^3)$ time complexity learning algorithm for fully recurrent continually running networks. *Neural Computation 4,* 243–248.

Smith, A. W., & Zipser, D. (1990). Learning sequential structure with the real-time recurrent learning algorithm. *International Journal of Neural Systems, 1,* 125–131.

Tsung, F. S. (1990). Learning in recurrent finite difference networks. In D. S. Tourtetzky, J. L. Elman, T. J. Sejnowski, & G. E. Hinton (Eds.), *Proceedings of the 1990 Connectionist Models Summer School.* San Mateo, CA: Morgan Kaufmann.

Tsung, F. S., Cottrell, G. W., & Selverston, A. (1990). Some experiments on learning stable network oscillations. *Proceedings of the International Joint Conference on Neural Networks,* San Diego, CA.

Waibel, A., Hanazawa, T., Hinton, G., Shikano, K., & Lang, K. (1987). *Phoneme recognition using time-delay neural networks* (Technical Report TR-I-0006). Japan: Advanced Telecommunications Research Institute.

Watrous, R. L., & Shastri, L. (1986). *Learning phonetic features using connectionist networks: An experiment in speech recognition* (Technical Report MS-CIS-86-78). Philadelphia: University of Pennsylvania.

Werbos, P. J. (1974). *Beyond regression: new tools for prediction and analysis in the behavioral sciences.* Unpublished doctoral dissertation. Harvard University.

Werbos, P. J. (1988). Generalization of backpropagation with application to a recurrent gas market model. *Neural Networks, 1,* 339–356.

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

Williams, R. J. (1990). Adaptive state representation and estimation using recurrent connectionist networks. In: W. T. Miller, R. S. Sutton, & P. J. Werbos (Eds.) *Neural Networks for Control.* Cambridge; MIT Press/Bradford Books.

Williams, R. J. (1989). *Complexity of exact gradient computation algorithms for recurrent neural networks* (Technical Report NU-CCS-89-27). Boston: Northeastern University, College of Computer Science.

Willaims, R. J., & Peng, J. (1990). An efficient gradient-based algorithm for on-line training of recurrent network trajectories. *Neural Computation, 2,* 490–501.

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

Williams, R. J., & Zipser, D. (1989b). Experimental analysis of the real-time recurrent learning algorithm. *Connection Science, 1,* 87–111.

Zipser, D. (1989). A subgrouping strategy that reduces complexity and speeds up learning in recurrent networks. *Neural Computation, 1,* 552–558.

A. APPENDIX

A.1. Preliminaries

For completeness, we first summarize some of the definitions and assumptions from the main text. Given a network with n units and m input lines, we define an $(m + n)$-tuple $\mathbf{x}(t)$ and index sets U and I such that $x_k(t)$, the kth component of $\mathbf{x}(t)$, represents either the output of a unit in the network at time t, if $k \in U$, or an external input to the network at time t, if $k \in I$. When $k \in U$, we also use the notation $y_k(t)$ for $x_k(t)$. For each $i \in U$ and $j \in U \cup I$ we have a unique weight w_{ij} on the connection from unit or input line j to unit i.

Letting $T(l)$ 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 at time t, we also define a time-varying n-tuple $\mathbf{e}(t)$ whose kth component is

$$e_k(t) = \begin{cases} d_k(t) - y_k(t) & \text{if } k \in T(t), \\ 0 & \text{otherwise.} \end{cases} \tag{A.1}$$

We then define the two functions

$$J(t) = \frac{1}{2} \sum_{k \in U} [e_k(t)]^2 \tag{A.2}$$

and

$$J^{\text{total}}(t', \text{itt}) = \sum_{\tau = t'+1}^{t} J(\tau), \tag{A.3}$$

where $t_0 \leq t' < t$, with t_0 denoting some fixed starting time.

For purposes of analyzing the backpropagation-through-time approach, we replace the dynamical Equations 2 and 3 by the equations

$$s_k(t + 1) = \sum_{l \in U \cup I} w_{kl}(t) x_l(t), \tag{A.4}$$

$$y_k(t + 1) = f_k(s_k(t + 1)), \tag{A.5}$$

and

$$w_{ij}(t) = w_{ij}, \tag{A.6}$$

for all $k \in U$, $i \in U$, $j \in U \cup I$, which give rise to equivalent dynamics for the s_k and y_k values. These equations can be viewed as representing the multilayer computation performed in the unrolled version $\mathcal{N}^*$ of the original arbitrary net $\mathcal{N}$, where t represents a layer index in $\mathcal{N}^*$ rather than a time index in $\mathcal{N}$.

Now suppose we are given a differentiable function F expressed in terms of $\{y_k(\tau) | k \in U, t' < \tau \leq t\}$, the outputs of the network over the time interval $(t', t]$. Note that while F may have an *explicit* dependence on some $y_k(\tau)$, it may also have an *implicit* dependence on this same value through later output values. To avoid the resulting ambiguity in interpreting partial derivatives like $\partial F / \partial y_k(\tau)$, we introduce variables $y_k^*(\tau)$ such that $y_k^*(\tau) = y_k(\tau)$ for all $k \in U$ and $\tau \in (t_0, t]$ and treat F as if it were expressed in terms of the variables $\{y_k^*(\tau)\}$ rather than the variables $\{y_k(\tau)\}$.[13]

Then, for all $k \in U$, define

$$\epsilon_k(\tau; F) = \frac{\partial F}{\partial y_k(\tau)} \tag{A.7}$$

for all $\tau \in [t_0, t]$ and define

$$\delta_k(\tau; F) = \frac{\partial F}{\partial s_k(\tau)} \tag{A.8}$$

for all $\tau \in (t_0, t]$. Also, define

$$e_k(\tau; F) = \frac{\partial F}{\partial y_k^*(\tau)} \tag{A.9}$$

for all $\tau \in (t_0, t]$. Note that $e_k(\tau; F) = 0$ whenever $\tau \leq t'$ because we assume that F has no explicit dependence on the output of the network for times outside the interval $(t', t]$. Finally, for $i \in U$, $j \in U \cup I$, $k \in U$, and $\tau \in [t_0, t]$, define

[13] To see why this is necessary, consider, for example, the two possible interpretations of $\partial F / \partial x$ given that $F(x, y) = x + y$ and $y = x$. The confusion occurs because the variable named "x" represents two different function arguments according to a strict use of the mathematical chain rule, a problem easily remedied by introducing additional variable names to eliminate such duplication. Werbos (1974, 1988), in addressing this same problem, uses the standard partial derivative notation to refer to explicit dependencies only, introducing the term *ordered derivative,* denoted in a different fashion, for a partial derivative which takes into account all influences. Our use of partial derivatives here corresponds to this latter notion.

$$p_{ij}^k(\tau) = \frac{\partial y_k(\tau)}{\partial w_{ij}}, \tag{A.10}$$

with

$$p_{ij}^k(t_0) = 0 \tag{A.11}$$

for all such i, j, and k since we assume that the initial state of the network has no functional dependence on the weights.

A.2 Derivation of the Back-Propagation-through-Time Formulation

Since F depends on $y_k(\tau)$ only through $y_k^*(\tau)$ and the variables $s_l(\tau + 1)$, as l ranges over U, we have

$$\frac{\partial F}{\partial y_k(\tau)} = \frac{\partial y_k^*(\tau)}{\partial y_k(\tau)} \frac{\partial F}{\partial y_k^*(\tau)} + \sum_{l \in U} \frac{\partial s_l(\tau + 1)}{\partial y_k(\tau)} \frac{\partial F}{\partial s_l(\tau + 1)}, \tag{A.12}$$

from which it follows that

$$\epsilon_k(\tau;F) = \begin{cases} e_k(t;F) & \text{if } \tau = t, \\ e_k(\tau;F) + \sum_{l \in U} w_{lk} \delta_l(\tau + 1;F) & \text{if } \tau < t. \end{cases} \tag{A.13}$$

Also, for all $\tau \leq t$,

$$\frac{\partial F}{\partial s_k(\tau)} = \frac{dy_k(\tau)}{ds_k(\tau)} \frac{\partial F}{\partial y_k(\tau)}, \tag{A.14}$$

so that

$$\delta_k(\tau; F) = f_k'(s_k(\tau))\epsilon_k(\tau; F). \tag{A.15}$$

In addition, for any appropriate i and j,

$$\frac{\partial F}{\partial w_{ij}} = \sum_{\tau=t_0}^{t-1} \frac{\partial F}{\partial w_{ij}(\tau)} \frac{\partial w_{ij}(\tau)}{\partial w_{ij}} = \sum_{\tau=t_0}^{t-1} \frac{\partial F}{\partial w_{ij}(\tau)}, \tag{A.16}$$

and, for any τ,

$$\frac{\partial F}{\partial w_{ij}(\tau)} = \frac{\partial F}{\partial s_i(\tau + 1)} \frac{\partial s_i(\tau + 1)}{\partial w_{ij}(\tau)} = \delta_i(\tau + 1;F)x_j(\tau). \tag{A.17}$$

Combining these last two results yields

$$\frac{\partial F}{\partial w_{ij}} = \sum_{\tau=t_0}^{t-1} \delta_i(\tau + 1;F)\mathrm{ix_j}(\tau). \tag{A.18}$$

Equations A.13, A.15, and A.18 represent the back-propagation-through-time computation of $\partial F/\partial w_{ij}$ for any differentiable function F expressed in terms of the outputs of individual units in a network of semilinear units. With $F = J(t)$, these specialize to the real-time BPTT Equations 12–16 because $e_k(t; J(t)) = e_k(t)$ and $e_k(\tau; J(t)) = 0$ for $\tau < t$. Similarly, Equations 17–20 for epochwise BPTT are obtained by setting $t = t_1$ and $F = J^{\text{total}}(t_0, t_1)$ and observing that $e_k(\tau; J^{\text{total}}(t_0, t_1)) = e_k(\tau)$ for all $\tau \leq t_1$.

A.3. Derivation of the Hybrid Formulation

Continuing on from Equation A.16, we may write

$$\frac{\partial F}{\partial w_{ij}} = \sum_{\tau=t_0}^{t'-1} \frac{\partial F}{\partial w_{ij}(\tau)} + \sum_{\tau=t'}^{t-1} \frac{\partial F}{\partial w_{ij}(\tau)}. \tag{A.19}$$

But the first sum on the right-hand side of this equation may be rewritten as

$$\sum_{\tau=t_0}^{t'-1} \frac{\partial F}{\partial w_{ij}(\tau)} = \sum_{\tau=t_0}^{t'-1} \sum_{l \in U} \frac{\partial F}{\partial y_l(t')} \frac{\partial y_l(t')}{\partial w_{ij}(\tau)} = \sum_{l \in U} \frac{\partial F}{\partial y_l(t')} \sum_{\tau=t_0}^{t'-1} \frac{\partial y_l(t')}{\partial w_{ij}(\tau)}$$

$$= \sum_{l \in U} \frac{\partial F}{\partial y_l(t')} \frac{\partial y_l(t')}{\partial w_{ij}} = \sum_{l \in U} \epsilon_l(t';F)p_{ij}^l(t').$$

Incorporating this result and equation A.17 into Equation A.19 yields

$$\frac{\partial F}{\partial w_{ij}} = \sum_{l \in U} \epsilon_l(t';F)p_{ij}^l(t') + \sum_{\tau=t_0}^{t'-1} \delta_i(\tau + 1;F)x_j(\tau). \tag{A.20}$$

This last result, together with equations (66) and (68), represents the basis for the hybrid FP/BPTT algorithm described in the text. For that algorithm we apply Equation A.20 a total of $n + 1$ times, first to $F = J^{\text{total}}(t', t)$, and then to $F = y_k(t)$ for reach $k \in U$. That is, back propagation through time, terminating at time step t', is performed $n + 1$ different times. When $F = J^{\text{total}}(t', t)$, this computation yields the desired gradient of $J^{\text{total}}(t', t)$, assuming that the values $p_{ij}^l(t')$, for all appropriate i, j, and k, are available. Performing the back propagation with $F = y_k(t)$ yields the values p_{ij}^k for all appropriate i and j, so this must be performed anew for each k to yield the entire set of p_{ij}^k values for use in the next time interval.

Not surprisingly, this hybrid formulation can be s hown to subsume both the BPTT and RTRL formulations. In particular, the pure BPTT Equation A.18 is the special case where $t' = t_0$. Likewise, if we let $F = J(t)$ and $t' = t$, we see that the second sum vanishes and the result is

$$\frac{\partial F}{\partial w_{ij}} = \sum_{l \in U} e_l(t) p_{ij}^l(t), \tag{A.21}$$

while letting $F = y_k(t)$ and $t' = t - 1$ yields

$$p_{ij}^k(t) = \sum_{l \in U} w_{kl} f_k'(s_k(t)) p_{ij}^l(t-1) + \delta_{ik} f_i'(s_i(t)) x_j(t-1)$$

$$= f_k'(s_k(t)) \left[\sum_{l \in U} w_{kl} p_{ij}^l(t-1) + \delta_{ik} x_j(t-1) \right]. \tag{A.22}$$