ResearchGate

See discussions, stats, and author profiles for this publication at: https://www.researchgate.net/publication/2343555

An Efficient Gradient-Based Algorithm for On-Line Training of Recurrent Network Trajectories

Article *in* Neural Computation · December 1990

DOI: 10.1162/neco.1990.2.4.490 · Source: CiteSeer

CITATIONS
664

READS
3,212

2 authors, including:

Jing Peng
Montclair State University
162 PUBLICATIONS **4,285** CITATIONS

SEE PROFILE

All content following this page was uploaded by Jing Peng on 20 May 2015.

The user has requested enhancement of the downloaded file.

An Efficient Gradient-Based Algorithm for On-Line Training of Recurrent Network Trajectories*

Ronald J. Williams and Jing Peng
College of Computer Science
Northeastern University
Boston, MA 02115

Appears in *Neural Computation, 2*, pp. 490-501, 1990.

Abstract

A novel variant of a familiar recurrent network learning algorithm is described. This algorithm is capable of shaping the behavior of an arbitrary recurrent network as it runs, and it is specifically designed to execute efficiently on serial machines.

1 Introduction

Artificial neural networks having feedback connections can implement a wide variety of dynamical systems. The problem of training such a network is the problem of finding a particular dynamical system from among a parameterized family of such systems which best fits the desired specification. This paper proposes a specific learning algorithm for *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. Another example is sequence production, 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.

A number of recurrent network learning algorithms have been investigated, many based on computation of the gradient of an error measure in weight space. Some of these use methods for computing this gradient exactly and some involve approximations designed to simplify the computation. Two specific approaches that use exact gradient computation methods are *back-propagation through time* (BPTT) and *real-time recurrent learning* (RTRL), to be described in somewhat more detail below. Both of these methods have been independently rediscovered by a number of investigators in the neural network field, and their origins can, in fact, be traced to a much earlier literature on optimal control of nonlinear dynamical systems. One reference for

*This research was partially supported by Grant IRI-8921275 from the National Science Foundation.

BPTT is (Rumelhart, Hinton, & Williams, 1986), and a reference for RTRL is (Williams & Zipser, 1989a). A related algorithm is the *recurrent backpropagation algorithm* of Pineda (1988), which can be viewed essentially as a computationally attractive special case of BPTT appropriate for situations when both the actual and desired trajectories consist of settling to a constant state. An extensive discussion of a number of gradient-based learning algorithms, including BPTT, RTRL, recurrent backpropagation, and other related algorithms can be found in (Williams and Zipser, 1990). Among the algorithms described in detail there is the particular one to be highlighted here.

The algorithm to be described here has five important properties. First, it is an on-line algorithm, designed to be used to train a network while it runs; no "manual" state resets or segmentation of the training stream into epochs is required. Second, it is a general-purpose algorithm, intended to be used with networks having arbitrary recurrent connectivity; no special architectures are assumed. Third, it is designed to train networks to perform arbitrary time-varying behaviors; it is not restricted to settling or periodic behaviors. Fourth, it is designed for time-efficient implementation on serial machines, in the sense that it minimizes the amount of computation required per time tick as the network runs. Finally, it has been experimentally verified to work at least as well as other comparable algorithms in solving a number of recurrent network learning problems; that is, it finds such solutions at least as often as these other algorithms and it generally requires no more time steps of network operation to do so. It is important to point out, however, that one property not claimed for this algorithm is biological plausibility. The real aim of the work reported here is to provide to those wishing to experiment with adaptive recurrent networks an algorithm combining some of the most attractive features of algorithms currently in use. In particular, this new algorithm is designed to enjoy the computational efficiency of BPTT while retaining the on-line character of RTRL. As will be seen below, it is not a radically new algorithm at all but is really just a more efficient variant of a method already being used successfully in some recurrent network research circles.

2 Formal Assumptions and Definitions

2.1 Network Architecture and Dynamics

For concreteness, we assume a network of semilinear units; it is straightforward to derive corresponding algorithms for a wide variety of alternative unit transfer functions. Also, we restrict attention here to a discrete-time formulation of the network dynamics.

Let the network have n units, with m external input lines. Let $\mathbf{y}(t)$ denote the n-tuple of outputs of the units in the network at time t, and let $\mathbf{x}^{\text{net}}(t)$ 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 k^{th} 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

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

For example, in a computer implementation using zero-based array indexing, it is convenient to index units and input lines by integers in the range $[0, m+n)$, with indices in $[0, m)$ corresponding to input lines and indices in $[m, m+n)$ corresponding to units in the network. 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 this use of 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 necessary for this exposition, we would have 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 i^{th} unit from either the j^{th} unit, if $j \in U$, or the j^{th} 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 i^{th} 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.

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 k^{th} 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

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

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. It is convenient in the algorithm descriptions given below to allow complete generality in the choice of squashing functions used (except for requiring them to be differentiable), but it is typical to let them all be the logistic function

$$f_k(s_k(t)) = \frac{1}{1 + e^{-s_k(t)}}. \tag{4}$$

In this case, the algorithms to be discussed below will make use of the fact that

$$f_k'(s_k(t)) = y_k(t)[1 - y_k(t)]. \tag{5}$$

The system of equations (**??**) and (**??**), where k ranges over U, constitute the entire discrete-time dynamics of the network, where the x_k values are defined by equation (**??**). 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.

2.2 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 at specified times. 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 k^{th} unit should match at time t. Then define a time-varying n-tuple $\mathbf{e}$ by

$$e_k(t) = \begin{cases} d_k(t) - y_k(t) & \text{if } k \in T(t) \\ 0 & \text{otherwise.} \end{cases} \tag{6}$$

Note that this formulation allows for the possibility that target values are specified for different units at different times. Now let

$$E(t) = 1/2 \sum_{k \in U} [e_k(t)]^2 \tag{7}$$

denote the overall network error at time t. A natural objective of learning is to minimize the total error

$$E^{\text{total}}(t', t) = \sum_{\tau=t'+1}^{t} E(\tau) \tag{8}$$

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

$$\nabla_{\mathbf{W}} E^{\text{total}}(t', t) = \sum_{\tau=t'+1}^{t} \nabla_{\mathbf{W}} E(\tau). \tag{9}$$

The point of computing this error gradient is to use it to adjust the weights. One natural way to make these weight changes is along a constant negative multiple of this gradient, so that

$$\Delta w_{ij} = -\alpha \frac{\partial E^{\text{total}}(t', t)}{\partial w_{ij}}, \tag{10}$$

where α is a positive learning rate parameter. For concreteness, all learning algorithms considered in this paper will be of this form. Because the main focus here is really on the strategy for computing the gradient itself, nothing to be described here is intended to preclude making some alternative use of this gradient information if desired. For example, all algorithms discussed here could incorporate momentum.

Similarly, for concreteness, the learning algorithms to be discussed here will be assumed to make their weight changes at the time when the gradient itself is computed, but this is not intended to rule out the possibility of delaying the actual weight changes until some later time. For example, it might make sense to keep a separate set of weight change accumulators whose values would only occasionally be added to the weights themselves.

3 Some Existing Approaches

3.1 Real-Time Recurrent Learning

The RTRL algorithm involves maintaining and updating the set of values

$$p_{ij}^k(t) = \frac{\partial y_k(t)}{\partial w_{ij}} \tag{11}$$

for each $k \in U$, $i \in U$, $j \in U \cup I$. These values are all initialized to 0 and updated on every time step by means of the equations

$$p_{ij}^k(t+1) = f_k'(s_k(t+1)) \left[\sum_{l \in U} w_{kl} p_{ij}^l(t) + \delta_{ik} x_j(t) \right], \tag{12}$$

where δ_{ik} denotes the Kronecker delta. On any time step when there is error, weights can be updated according to the equations

$$\Delta w_{ij} = -\alpha \frac{\partial E(t)}{\partial w_{ij}} = \alpha \sum_{k \in U} e_k(t) p_{ij}^k(t). \tag{13}$$

In a network having n units and $O(n^2)$ weights, this algorithm requires the storage of $O(n^3)$ real numbers and the performance of $O(n^4)$ arithmetic operations per time step, which is quite severe. Zipser (1989) has proposed a strategy called *subgrouping* which is designed to reduce this computational complexity by ignoring certain structural dependencies in the network and computing the gradient as if these were zero. The attractive feature of any form of RTRL is its natural on-line character.

3.2 Epochwise Backpropagation Through Time

The backpropagation-through-time approach can be derived by unfolding the temporal operation of a network into a multilayer feedforward network that grows by one layer on each time step. If the training stream is segmented into epochs, then one can derive the specific version that we will call *epochwise backpropagation through time*. This algorithm is organized as follows. With t_0 denoting the start time of the epoch and t_1 denoting its end time, the objective is compute the gradient of $E^{\text{total}}(t_0, t_1)$. This is done by first letting the network run through the interval $[t_0, t_1]$ and saving the entire history of inputs to the network, network state, and target vectors over this interval. Then a single backward pass over this history buffer is performed to compute the set of values $\delta_k(\tau) = -\partial E^{\text{total}}(t_0, t_1)/\partial s_k(\tau)$, for all $k \in U$ and $\tau \in (t_0, t_1]$, by means of the equations

$$\delta_k(\tau) = \begin{cases} f_k'(s_k(\tau)) e_k(\tau) & \text{if } \tau = t_1 \\ f_k'(s_k(\tau)) \left[e_k(\tau) + \sum_{l \in U} w_{lk} \delta_l(\tau + 1) \right] & \text{if } t_0 < \tau < t_1 \end{cases} \tag{14}$$

This can be viewed as representing the familiar backpropagation computation applied to a feedforward network in which target values are specified for units in many layers, not just the last. The process begins at the last time step and proceeds to earlier time steps through the repeated use of these equations. When describing this algorithm it is helpful to speak of *injecting error* at time τ to mean the computational step of adding $e_k(\tau)$ for each k to the appropriate sum when computing the bracketed expression in the equation for $\delta_k(\tau)$. We also consider the computation of $\delta_k(t_1)$ to involve a corresponding injection of error; in this case, it can be viewed as adding $e_k(t_1)$ to 0.

Once the backpropagation computation has been performed back to time $t_0 + 1$, the weight changes may be made along the negative gradient of overall error by means of the equations

$$\Delta w_{ij} = -\alpha \frac{\partial E^{\text{total}}(t_0, t_1)}{\partial w_{ij}} = \alpha \sum_{\tau = t_0 + 1}^{t_1} \delta_i(\tau) x_j(\tau - 1). \tag{15}$$

When a network having n units and $O(n^2)$ weights is run over a single epoch of h time steps, it is easy to see that this algorithm requires the storage of $O(nh)$ real numbers and performs $O(n^2h)$ arithmetic operations during the backward pass, together with another $O(n^2h)$ operations to compute the weight updates.

3.3 Truncated Backpropagation Through Time

If the training stream is not segmented into independent epochs, one can still consider using BPTT. The idea is to compute the negative gradient of $E(t)$ and make the appropriate weight changes at each time t while the network continues to run. To do this requires saving the entire history of network input and network state since the starting time t_0. Then, for each fixed t, a set of values $\delta_k(\tau) = -\partial E(t_0, t_1)/\partial s_k(\tau)$ for all $k \in U$ and $\tau \in (t_0, t]$ are computed by means of the equations

$$\delta_k(\tau) = \begin{cases} f'_k(s_k(\tau))e_k(\tau) & \text{if } \tau = t \\ f'_k(s_k(\tau)) \sum_{l \in U} w_{lk}\delta_l(\tau + 1) & \text{if } t_0 < \tau < t \end{cases} \tag{16}$$

The process begins at the most recent time step by injecting error there, but, unlike epochwise BPTT, error is not injected for any earlier time steps. This is why earlier target values need not be saved for this algorithm.

Once the backpropagation computation has been performed back to time $t_0 + 1$, the weight changes may be made along the negative gradient of $E(t)$ by means of the equations

$$\Delta w_{ij} = -\alpha \frac{\partial E(t_0, t_1)}{\partial w_{ij}} = \alpha \sum_{\tau=t_0+1}^{t} \delta_i(\tau) x_j(\tau - 1). \tag{17}$$

It is interesting to note that this algorithm, which can be called *real-time backpropagation through time* is just another way of organizing the computation of the quantity also computed by RTRL, namely the gradient of $E(t)$ in weight space. Clearly, RTRL is preferable since this algorithm involves computation time and storage that grow linearly with time as the network runs. In fact, because of these obvious shortcomings, no one would seriously consider using this particular algorithm for on-line learning. Instead, one can use a bounded-history approximation to it in which relevant information is saved for a fixed number h of time steps and any information older than that is forgotten. In general, this should be regarded as a heuristic technique for simplifying the computation, although, as discussed below, it can sometimes serve as an adequate approximation to the true gradient and may also be more appropriate in those situations where weights are adjusted as the network runs. Let us call this algorithm *truncated backpropagation through time*. With h representing the number of prior time steps saved, this algorithm will be denoted BPTT(h).

For BPTT(h), one computes, at each time t, the values $\delta_k(\tau)$ for all $k \in U$, but only for $\tau \in (t - h, t]$, using the same equations (**??**) as before. After these have been computed, weight changes are made using

$$\Delta w_{ij} = \alpha \sum_{\tau=t-h+1}^{t} \delta_i(\tau) x_j(\tau - 1), \tag{18}$$

which is, of course, just equation (**??**) with all terms for which $\tau \leq t - h$ taken to be 0.

In a network of n units and $O(n^2)$ weights, this algorithm clearly requires $O(nh)$ storage and, at each time step, requires $O(n^2h)$ arithmetic operations for the backward pass through the history buffer and another $O(n^2h)$ operations to compute the weight updates.[1] When compared with RTRL, BPTT(h) with any reasonably small h is clearly a much more efficient on-line algorithm in terms of the computational effort required per time step.

An extreme example of this truncation strategy is found in the algorithm explored by Cleeremans, Servan-Schreiber, and McClelland (1989), which combines the use of BPTT(1) in the recurrent portion of a particular network architecture with feedforward backpropagation in the remainder of the network.

4 The Improved Algorithm

Now consider a problem in which either BPTT(h) or epochwise BPTT might be used, say a problem involving training over a single epoch which is much longer than h. Epochwise BPTT will, of course, require more storage, but it is clear that it will still only require an average of $O(n^2)$ arithmetic operations per time step, compared to $O(n^2h)$ for BPTT(h). Thus, in terms of execution time, BPTT(h) is slower than epochwise BPTT by esentially a factor of h. Since h must generally be chosen sufficiently large to permit a particular task to be learned, it is not an option to reduce h to speed up the execution time.[2]

One may thus ask whether there are any effective general-purpose on-line algorithms having the same execution time as epochwise BPTT. As it turns out, there is a class of on-line algorithms whose average computational effort expended per time step asymptotically approach that of epochwise BPTT. This class of algorithms is obtained by combining aspects of epochwise BPTT with the truncated BPTT approach.

Note that in BPTT(h) a backward pass through the most recent h time steps is performed anew each time the network is run through an additional time step. To generalize this, one may consider letting the network run through h' additional time steps before performing the next BPTT computation, where $h' \leq h$. In this case, if t represents a time at which BPTT is to be performed, the algorithm computes an approximation to $\nabla_{\mathbf{W}} E^{\text{total}}(t-h', t)$ by taking into account only that part of the history over the interval $[t-h, t]$. Specifically, this involves performing a backward pass at time t to compute the set of values $\delta_k(\tau)$ for all $k \in U$ and $\tau \in (t-h, t]$, just as with BPTT(h), but this time using the equations

$$\delta_k(\tau) = \begin{cases} f_k'(s_k(\tau))e_k(\tau) & \text{if } \tau = t \\ f_k'(s_k(\tau))\left[e_k(\tau) + \sum_{l \in U} w_{lk}\delta_l(\tau+1)\right] & \text{if } t-h' < \tau < t \\ f_k'(s_k(\tau))\sum_{l \in U} w_{lk}\delta_l(\tau+1) & \text{if } t-h < \tau \leq t-h' \end{cases} \tag{19}$$

[1] When the weights are adjusted on every time step, it may be appropriate to store all the past weights as well, using these on the backward pass rather than the current weights. This variant obviously requires $O(n^2h)$ storage. In practice, we have not found much difference in performance between this version and the simpler one described here.

[2] In particular, h must generally be chosen to be roughly as large as typically encountered delays between input and corresponding output in the training stream.

After this backward pass has been performed, weight updates can be computed using

$$\Delta w_{ij} = \alpha \sum_{\tau=t-h+1}^{t} \delta_i(\tau) x_j(\tau - 1), \tag{20}$$

just as with BPTT(h).

The key feature of this algorithm is that the next backward pass is not performed until time step $t + h'$; in the intervening time the history of network input, network state, and target values are saved in the history buffer, but no processing is performed on this data. Let us denote this algorithm BPTT($h; h'$). Clearly BPTT(h) is the same as BPTT($h; 1$), and BPTT($h; h$) is the epochwise BPTT algorithm. Figure **??** depicts the processing performed by the BPTT($h; h'$) algorithm.

The storage requirements of this algorithm are essentially the same as those of BPTT(h), except that it requires the additional storage of $h' - 1$ sets of prior target values. However, 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'. It thus requires $O(nh)$ space and requires performing an average of $O(n^2 h/h')$ operations per time step in a network having n units and $O(n^2)$ weights. Furthermore, it is clear that making h/h' small makes the algorithm more efficient. At the same time, to obtain a reasonably close approximation to the true gradient (or at least to what would be obtained through the use of truncated BPTT), it is only necessary to make the difference $h - h'$ sufficiently large. This is because no error is injected for the earliest $h - h'$ time steps in the buffer. Thus a practical and highly efficient on-line algorithm for recurrent networks is 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.

5 Experimental Performance

An important question to be addressed in studies of recurrent network learning algorithms, whatever other constraints to which they must conform, is how much total computational effort must be expended to achieve the desired performance. While BPTT($h; h'$) has been shown to require less average computational effort per time step than comparable general-purpose on-line recurrent network learning algorithms, it is equally important to determine how it compares with these other algorithms in terms 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.

A number of experiments have been performed to compare the performance of the improved version of truncated BPTT with its unmodified counterpart. In particular, studies were performed in which BPTT($2h; h$) was compared with BPTT(h) on specific tasks. The results of these experiments have been that the success rate of BPTT($2h; h$) is essentially identical to that of BPTT(h), with the number of time ticks required to find a solution comparable for both algorithms. Thus, in these experiments, the actual running time was significantly reduced when BPTT($2h; h$) was used, with the amount of speedup essentially equal to that predicted by an analysis of the computational requirements per time step. Among the tasks studied were several described in (Williams &

Figure 1: A schematic representation of the storage and processing required for the BPTT($h; h'$) algorithm. Shown here is the history buffer at time step t, assumed to be one of the times when the backpropagation pass is to occur. At this time the history buffer contains the values of the network input and activity for time step t as well as for the h prior time steps. It also contains target values for the most recent h' time steps, including time step t. Dashed arrows indicate injection of error, which occurs only at the h' uppermost levels in the buffer. Once the backpropagation pass has occurred and the weights have been adjusted, the process is next performed at time step $t + h'$. The more familiar BPTT(h) algorithm is just the special case when $h' = 1$.

Zipser, 1989a) and elaborated upon in (Williams & Zipser, 1989b). One noteworthy example is the "Turing machine" task, involving networks having 12-15 units. In earlier experiments, reported in (Williams & Zipser, 1990), it had been found that BPTT(9) gave a factor of 28 speedup in running time over RTRL on this same task, with success rate at least as high. Using BPTT(16;8) gave an additional factor of 2 speedup,[3] making BPTT(16;8) well over 50 times faster than RTRL on this task.

6 Discussion

Note that as the algorithm has been described here, it actually involves a very uneven rate of use of computational resources per time step. On many time steps, no significant computation is performed, while at certain times much higher peak computation than the average is required. For simulation studies, this is of no consequence; the only noticeable effect of this will be that simulated time runs unevenly. However, one might also consider applying recurrent network learning algorithms to real-time signal processing or control problems, in which it is important to spread the computation evenly over the individual time steps. In fact, this algorithm can be implemented in such a fashion. The idea is to interleave the backpropagation computation with the forward computation. For example, if $h = 2h'$, one should backpropagate through 2 time steps during each single (external) time step. In general, this requires a buffer which can hold the appropriate data for a total of $h + h'$ time steps.

We should point out that we are emphasizing the speed advantage of BPTT($h; h'$) over other general-purpose on-line recurrent network algorithms in *serial* machines. It can also be used as an efficient algorithm in parallel machines, but since BPTT(h) itself can be implemented efficiently in an appropriate specially designed pipelined architecture, the use of the BPTT($h; h'$) algorithm may offer no particular advantage.

It is also appropriate to make some additional observations concerning the degree of approximation involved when the backpropagation computation is truncated to h prior time steps (whether using BPTT(h) or BPTT($h; h'$)). If weights were held constant over the history of operation of the network, the true gradient would be that computed by backpropagating all the way back to the time of network initialization. Even in this case, however, it may well be that this computation undergoes exponential decay over (backward) time, in which case the difference between truncating or not can be negligible.[4] In situations when there is no such exponential decay, though, truncation may give a very poor approximation.

However, if weights are adjusted as the network operates, as they necessarily must be for any on-line algorithm, use of a gradient computation method based on the assumption that the weights are fixed over all past time involves a different kind of approximation which can actually be

[3] Careful analysis of the computational requirements of BPTT(9) and of 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.

[4] In general, whether such exponential decay occurs depends on the nature of the forward dynamics of the network; for example, one would expect it to occur when the network is operating in a basin of attraction of a stable fixed point.

mitigated by ignoring dependencies into the distant past, as occurs when using truncated BPTT. Such information involving the distant past 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. Unlike the truncation strategy, however, this does not reduce the computational complexity of the algorithm.

Another potential benefit of the truncation strategy is that it can help provide a useful inductive bias by forcing the learning system to consider only reasonably short-term correlations between input and desired output; of course, this is appropriate only when these short-term correlations are actually the important ones, as is often the case.

We believe that some combination of these effects is responsible for our experimentally observed result that not only is it detrimental to have too short a history buffer length, it can also be detrimental to have too long a history buffer. Studies reported in (Williams & Zipser, 1990), in which RTRL was compared with BPTT(h), are also consistent with this result. These other studies found that BPTT(9) not only ran faster than RTRL on the "Turing machine" task, as described above, but it also had a higher success rate. Since RTRL implicitly makes use of information over the entire history of operation of the network, this can be taken as evidence of the benefit of ignoring all but the recent past.

Finally, we should point out that while we have focused on the use of dicrete time here, the algorithm described by Pearlmutter (1989), which can be viewed as a continuous-time version of epochwise BPTT, can serve as the basis for a correspondingly efficient continuous-time version of the on-line algorithm presented here.

7 References

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

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

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

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

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.

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.

Williams, R. J., & Zipser, D. (1990). *Gradient-based learning algorithms for recurrent connectionist networks*, (Technical Report NU-CCS-90-9). Boston: Northeastern University, College of Computer Science.

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

View publication stats