A Learning Algorithm for Continually Running Fully Recurrent Neural Networks

Ronald J. Williams
College of Computer Science
Northeastern University
Boston, Massachusetts 02115

and
David Zipser
Institute for Cognitive Science
University of California, San Diego
La Jolla, California 92093

Appears in *Neural Computation, 1*, pp. 270-280, 1989.

Abstract

The exact form of a gradient-following learning algorithm for completely recurrent networks running in continually sampled time is derived and used as the basis for practical algorithms for temporal supervised learning tasks. These algorithms have: (1) the advantage that they do not require a precisely defined training interval, operating while the network runs; and (2) the disadvantage that they require nonlocal communication in the network being trained and are computationally expensive. These algorithms are shown to allow networks having recurrent connections to learn complex tasks requiring the retention of information over time periods having either fixed or indefinite length.

1 Introduction

A major problem in connectionist theory is to develop learning algorithms that can tap the full computational power of neural networks. Much progress has been made with feedforward networks, and attention has recently turned to developing algorithms for networks with recurrent connections, which have important capabilities not found in feedforward networks, including attractor dynamics and the ability to store information for later use. Of particular interest is their ability to deal with time-varying input or output through their own natural temporal operation.

A variety of approaches to learning in networks with recurrent connections have been proposed. Algorithms for the special case of networks that settle to stable states, often regarded as associative

memory networks, have been proposed by Hopfield (1982), Lapedes and Farber (1986), Almeida (1987), Pineda (1988), and Rohwer and Forrest (1987).

Other researchers have focused on learning algorithms for more general networks that use recurrent connections to deal with time-varying input and/or output in nontrivial ways. A general framework for such problems was laid out by Rumelhart, Hinton, and Williams (1986), who unfolded the recurrent network into a multilayer feedforward network that grows by one layer on each time step. We will call this approach *backpropagation through time*. One of its primary strengths is its generality, but a corresponding weakness is its growing memory requirement when given an arbitrarily long training sequence.

Other approaches to training recurrent nets to handle time-varying input or output have been suggested or investigated by Jordan (1986), Bachrach (1988), Mozer (1988), Elman (1988), Servan-Schreiber, Cleeremans, and McClelland (1988), Robinson and Fallside (1987), Stornetta, Hogg, and Huberman (1987), Gallant and King (1988), and Pearlmutter (1988). Many of these approaches use restricted architectures or are based on more computationally limited approximations to the full backpropagation-through-time computation.

The approach we propose here enjoys the generality of the backpropagation-through-time approach while not suffering from its growing memory requirement in arbitrarily long training sequences. It coincides with an approach suggested in the system identification literature (McBride & Narendra, 1965) for tuning the parameters of general dynamical systems. The work of Bachrach (1988) and Mozer (1988) represents special cases of the algorithm presented here, and Robinson and Fallside (1987) have given an alternative description of the full algorithm as well. However, to the best of our knowledge, none of these investigators has published an account of the behavior of this algorithm in unrestricted architectures.

2 The Learning Algorithm and Variations

2.1 The Basic Algorithm

Let the network have n units, with m external input lines. Let $\mathbf{y}(t)$ denote the n-tuple of outputs of the units in the network at time t, and let $\mathbf{x}(t)$ denote the m-tuple of external input signals to the network at time t. We concatenate $\mathbf{y}(t)$ and $\mathbf{x}(t)$ to form the $(m+n)$-tuple $\mathbf{z}(t)$, with U denoting the set of indices k such that z_k is the output of a unit in the network and I the set of indices k for which z_k is an external input. The indices on $\mathbf{y}$ and $\mathbf{x}$ are chosen to correspond to those of $\mathbf{z}$, so that

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

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 $ntimes(m+n)$ matrix. To allow each unit to have a bias weight we simply include among the m input lines one input whose value is always 1.

In what follows we use a discrete time formulation and we assume that the network consists entirely of semilinear units; it is straightforward to extend the approach to continuous time and

other forms of differentiable unit computation. We let

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

denote the net input to the kth unit at time t, for $k \in U$, with its output at the next time step being

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

where f_k is the unit's squashing function.

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

We now derive an algorithm for training this network in what we will call a *temporal 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 kth unit should match at time t. Then define a time-varying n-tuple $\mathbf{e}$ by

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

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

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

denote the overall network error at time t. For the moment, assume that the network is run starting at time t_0 up to some final time t_1. We take as the objective the minimization of the total error

$$J_{\text{total}}(t_0, t_1) = \sum_{t=t_0+1}^{t_1} J(t) \tag{6}$$

over this trajectory. We do this by a gradient descent procedure, adjusting $\mathbf{W}$ along the negative of $\nabla \mathbf{W} J_{\text{total}}(t_0, t+1)$.

Since the total error is just the sum of the errors at the individual time steps, one way to compute this gradient is by accumulating the values of $\nabla \mathbf{W} J(t)$ for each time step along the trajectory. The overall weight change for any particular weight w_{ij} in the network can thus be written as

$$\Delta w_{ij} = \sum_{t=t_0+1}^{t_1} \Delta w_{ij}(t), \tag{7}$$

where

$$\Delta w_{ij}(t) = -\alpha \frac{\partial J(t)}{\partial w_{ij}} \tag{8}$$

and α is some fixed positive learning rate.

Now

$$-\frac{\partial J(t)}{\partial w_{ij}} = \sum_{k \in U} e_k(t) \frac{\partial y_k(t)}{\partial w_{ij}}, \tag{9}$$

where $\partial y_k(t)/\partial w_{ij}$ is easily computed by differentiating the network dynamics (equations 2 and 3), yielding

$$\frac{\partial y_k(t+1)}{\partial w_{ij}} = f_k{}'(s_k(t)) \left[\sum_{l \in U} w_{kl} \frac{\partial y_l(t)}{\partial w_{ij}} + \delta_{ik} z_j(t) \right], \tag{10}$$

where δ_{ik} denotes the Kronecker delta. Because we assume that the initial state of the network has no functional dependence on the weights, we also have

$$\frac{\partial y_k(t_0)}{\partial w_{ij}} = 0. \tag{11}$$

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

We thus create a dynamical system with variables p_{ij}^k for all $k \in U$, $i \in U$, and $j \in U \cup I$, and dynamics given by

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

with initial conditions

$$p_{ij}^k(t_0) = 0, \tag{13}$$

and it follows that

$$p_{ij}^k(t) = \frac{\partial y_k(t)}{\partial w_{ij}} \tag{14}$$

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

The precise algorithm then consists of computing, at each time step t from t_0 to t_1, the quantities $p_{ij}^k(t)$, using equations 12 and 13, and then using the discrepancies $e_k(t)$ between the desired and actual outputs to compute the weight changes

$$\Delta w_{ij}(t) = \alpha \sum_{k \in U} e_k(t) p_{ij}^k(t). \tag{15}$$

The overall correction to be applied to each weight w_{ij} in the net is then simply the sum of these individual $\Delta w_{ij}(t)$ values for each time step t along the trajectory.

In the case when each unit in the network uses the logistic squashing function we use

$$f_k{}'(s_k(t)) = y_k(t+1)[1 - y_k(t+1)] \tag{16}$$

in equation 12.

2.2 Real-Time Recurrent Learning

The above algorithm was derived on the assumption that the weights remained fixed throughout the trajectory. In order to allow real-time training of behaviors of indefinite duration, however, it is useful to relax this assumption and actually make the weight changes while the network is running. This has the important advantage that no epoch boundaries need to be defined for training the network, leading to both a conceptual and an implementational simplification of the procedure. For this algorithm, we simply increment each weight w_{ij} by the amount $\Delta w_{ij}(t)$ given

by equation 15 at time step t, without accumulating the values elsewhere and making the weight changes at some later time.

A potential disadvantage of this real-time procedure is that it no longer follows the precise negative gradient of the total error along a trajectory. However, this is exactly analogous to the commonly used method of training a feedforward net by making weight changes after each pattern presentation rather than accumulating them elsewhere and then making the net change after the end of each complete cycle of pattern presentation. While the resulting algorithm is no longer guaranteed to follow the gradient of total error, the practical differences are often slight, with the two versions becoming more nearly identical as the learning rate is made smaller. The most severe potential consequence of this departure from true gradient-following behavior for real-time procedure for training the dynamics is that the observed trajectory may itself depend on the variation in the weights caused by the learning algorithm, which can be viewed as providing another source of negative feedback in the system. To avoid this, one wants the time scale of the weight changes to be much slower than the time scale of the network operation, meaning that the learning rate must be sufficiently small.

2.3 Teacher-Forced Real-Time Recurrent Learning

An interesting technique that is frequently used in dynamical supervised learning tasks (Jordan, 1986; Pineda, 1988) is to replace the actual output $y_k(t)$ of a unit by the teacher signal $d_k(t)$ in subsequent computation of the behavior of the network, whenever such a value exists. We call this technique *teacher forcing*. The dynamics of a teacher-forced network during training are given by equations 2 and 3, as before, but where $\mathbf{z}(t)$ is now defined by

$$z_k(t) = \begin{cases} x_k(t) & \text{if } k \in I \\ d_k(t) & \text{if } k \in T(t) \\ y_k(t) & \text{if } k \in U - T(t). \end{cases} \tag{17}$$

rather than by equation 1.

To derive a learning algorithm for this situation, we once again differentiate the dynamical equations with respect to w_{ij}. This time, however, we find that

$$\frac{\partial y_k(t+1)}{\partial w_{ij}} = f_k{}'(s_k(t)) \left[\sum_{l \in U - T(t)} w_{kl} \frac{\partial y_l(t)}{\partial w_{ij}} + \delta_{ik} z_j(t) \right], \tag{18}$$

since $\partial d_l(t)/\partial w_{ij} = 0$ for all $l \in T(t)$ and for all t. For the teacher-forced version we thus alter our learning algorithm so that the dynamics of the p_{ij}^k values are given by

$$p_{ij}^k(t+1) = f_k{}'(s_k(t)) \left[\sum_{l \in U - T(t)} w_{kl} p_{ij}^l(t) + \delta_{ik} z_j(t) \right], \tag{19}$$

rather than equation 12, with the same initial conditions as before. Note that equation 19 is the same as equation 12 if we treat the values of $p_{ij}^l(t)$ as zero for all $l \in T(t)$ when computing $p_{ij}^k(t+1)$.

The teacher-forced version of the algorithm is thus essentially the same as the earlier one, with two simple alterations: (1) where specified, desired values are used in place of actual values to compute future activity in the network; and (2) the corresponding p_{ij}^k values are set to zero after they have been used to compute the Δw_{ij} values.

2.4 Computational Features of the Real-Time Recurrent Learning Algorithms

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

3 Simulation Experiments

We have tested these algorithms on several tasks, most of which can be characterized as requiring the network to learn to configure itself so that it stores important information computed from the input stream at earlier times to help determine the output at later times. In other words, the network is required to learn to represent useful internal state to accomplish these tasks. For all the tasks described here, the experiments were run with the networks initially configured with full interconnections among the units, with every input line connected to every unit, and with all weights having small randomly chosen values. The units to be trained were selected arbitrarily. More details on these simulations can be found in Williams and Zipser (1988).

3.1 Pipelined XOR

For this task, two nonbias input lines are used, each carrying a randomly selected bit on each time step. One unit in the network is trained to match a teacher signal at time t consisting of the XOR of the input values given to the network at time $t - tau$, where the computation delay tau is chosen in various experiments to be 2, 3, or 4 time steps. With 3 units and a delay of 2 time steps, the network learns to configure itself to be a standard 2-hidden-unit multilayer network for computing this function. For longer delays, more units are required, and the network generally configures itself to have more layers in order to match the required delay. Teacher forcing was not used for this task.

3.2 Simple Sequence Recognition

For this task, there are two units and m nonbias input lines, where $m >= 2$. Two of the input lines, called the a and b lines, serve a special purpose, with all others serving as distractors. At each time step exactly one input line carries a 1, with all others carrying a 0. The object is for a selected unit in the network to output a 1 immediately following the first occurrence of activity on the b line following activity on the a line, regardless of the intervening time span. At all other times, this unit should output a 0. Once such a b occurs, its corresponding a is considered to be "used up," so that the next time the unit should output a 1 is when a new a has been followed by its first "matching" b. Unlike the previous task, this cannot be performed by any feedforward network whose input comes from tapped delay lines on the input stream. A solution consisting essentially of a flip-flop and an AND gate is readily found by the unforced version of the algorithm.

3.3 Delayed Nonmatch to Sample

In this task, the network must remember a cued input pattern and then compare it to subsequent input patterns, outputting a 0 if they match and a 1 if they don't. We have investigated a simple version of this task using a network with two input lines. One line represents the pattern and is set to 0 or 1 at random on each cycle. The other line is the cue that, when set to 1, indicates that the corresponding bit on the pattern line must be remembered and used for matching until the next occurrence of the cue. The cue bit is set randomly as well. This task has some elements in common with both of the previous tasks in that it involves an internal computation of the XOR of appropriate bits (requiring a computation delay) as well as having the requirement that the network retain indefinitely the value of the cued pattern. One of the interesting features of the solutions found by the unforced version of the algorithm is the nature of the internal representation of the cued pattern. Sometimes a single unit is recruited to act as an appropriate flip-flop, with the other units performing the required logic; at other times a dynamic distributed representation is developed in which no static pattern indicates the stored bit.

3.4 Learning to Be a Turing Machine

The most elaborate of the tasks we have studied is that of learning to mimic the finite state controller of a Turing machine deciding whether a tape marked with an arbitrary length string of left and right parentheses consists entirely of sets of balanced parentheses. The network observes the actions of the finite state controller but is not allowed to observe its states. Networks with 15 units always learned the task. The minimum-size network to learn the task had 12 units.

3.5 Learning to Oscillate

Three simple network oscillation tasks that we have studied are (1) training a single unit to produce 010101...; (2) training a 2-unit net so that one of the units produces 00110011...; and (3) training a 2-unit net so that one of the units produces approximately sinusoidal oscillation of period on the order of 25 time steps in spite of the nonlinearity of the units involved.

We have used both versions of the algorithm on these oscillation tasks, with and without teacher forcing, and we have found that only the version with teacher forcing is capable of solving

these problems in general. The reason for this appears to be that in order to produce oscillation in a net that initially manifests settling behavior (because of the initial small weight values), the weights must be adjusted across a bifurcation boundary, but the gradient itself cannot yield the necessary information because it is zero or very close to zero. However, if one is free to adjust both weights and initial conditions, at least in some cases this problem disappears. Something like this appears to be at the heart of the success of the use of teacher forcing: By using desired values in the net, one is helping to control the initial conditions for the subsequent dynamics. Pineda (1988) has observed a similar need for teacher forcing when attempting to add new stable points in an associative memory rather than just moving existing ones around.

4 Discussion

Our primary goal here has been to derive a learning algorithm to train completely recurrent, continually updated networks to learn temporal tasks. Our emphasis has been on using uniform starting configurations that contain no a priori information about the temporal nature of the task. In most cases we have used statistically derived training sets that have not been extensively optimized to promote learning. The results of the simulation experiments described here demonstrate that the algorithm has sufficient generality and power to work under these conditions.

The algorithm we have described here is nonlocal in the sense that, for learning, each weight must have access to both the complete recurrent weight matrix $\mathbf{W}$ and the whole error vector $\mathbf{e}$. This makes it unlikely that this algorithm, in its current form, can serve as the basis for learning in actual neurophysiological networks. The algorithm is, however, inherently quite parallel so that computation speed would benefit greatly from parallel hardware.

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

Acknowledgements

We wish to thank Jonathan Bachrach for sharing with us his insights into the issue of training recurrent networks. It was his work that first brought to our attention the possibility of on-line computation of the error gradient, and we hereby acknowledge his important contribution to our own development of these ideas.

This research was supported by Grant IRI-8703566 from the National Science Foundation to the first author and Contract N00014-87-K-0671 from the Office of Naval Research, Grant 86-0062 from the Air Force Office of Scientific Research, and grants from the System Development Foundation to the second author.

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.*

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

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

Gallant, S. I. & King, D. (1988). Experiments with sequential associative memories. *Proceedings of the Tenth Annual Conference of the Cognitive Science Society.*

Hopfield, J. J. (1982). Neural networks as physical systems with emergent collective computational abilities. *Proceedings of the National Academy of Sciences, 79*, 2554-2558.

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

Lapedes, A. & Farber, R. (1986). A self-optimizing, nonsymmetrical neural net for content addressable memory and pattern recognition, *Physica D, 22*, 247-259.

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

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

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

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

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

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

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

Servan-Schreiber, D., Cleeremans, A., & McClelland (1988). *Encoding sequential structure in*

simple recurrent networks (Tech. Rep. CMU-CS-88-183). Pittsburgh: Carnegie Mellon University, Department of Computer Science.

Stornetta, W. S., Hogg, T., & Huberman, B. A. (1987). A dynamical approach to temporal pattern processing. *Proceedings of the IEEE Conference on Neural Information Processing Systems.*