Communicated by Fernando Pineda

Learning State Space Trajectories in Recurrent Neural Networks

Barak A. Pearlmutter
School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, USA

Many neural network learning procedures compute gradients of the errors on the output layer of units after they have settled to their final values. We describe a procedure for finding $\partial E/\partial w_{ij}$ where E is an error functional of the temporal trajectory of the states of a continuous recurrent network and w_{ij} are the weights of that network. Computing these quantities allows one to perform gradient descent in the weights to minimize E. Simulations in which networks are taught to move through limit cycles are shown. This type of recurrent network seems particularly suited for temporally continuous domains, such as signal processing, control, and speech.

1 Introduction

Pineda (1987) has shown how to train the fixpoints of a recurrent temporally continuous generalization of backpropagation networks (Rumelhart et al. 1986). Such networks are governed by the coupled differential equations

$$T_i \frac{dy_i}{dt} = -y_i + \sigma(x_i) + I_i \tag{1.1}$$

where

$$x_i = \sum_j w_{ji} y_j \tag{1.2}$$

is the total input to unit i, y_i is the state of unit i, T_i is the time constant of unit i, σ is an arbitrary differentiable function[1], w_{ij} are the weights, and the initial conditions $y_i(t_0)$ and driving functions $I_i(t)$ are the inputs to the system.

Consider minimizing $E(\mathbf{y})$, some functional of the trajectory taken by $\mathbf{y}$ between t_0 and t_1. For instance, $E = \int_{t_0}^{t_1} (y_0(t) - f(t))^2 dt$ measures the deviation of y_0 from the function f, and minimizing this E would teach the network to have y_0 imitate f. Below, we develop a technique for computing $\partial E(\mathbf{y})/\partial w_{ij}$ and $\partial E(\mathbf{y})/\partial T_i$, thus allowing us to do gradient descent in the weights and time constants so as to minimize E.

[1]Typically $\sigma(\xi) = (1 + e^{-\xi})^{-1}$, in which case $\sigma'(\xi) = \sigma(\xi)(1 - \sigma(\xi))$.

 © 1989 Massachusetts Institute of Technology

2 A Forward/Backward Technique

Let us define

$$e_i(t) = \frac{\delta E}{\delta y_i(t)}. \tag{2.1}$$

In the usual case E is of the form $\int_{t_0}^{t_1} f(\mathbf{y}(t), t)\, dt$ so $e_i(t) = \partial f(\mathbf{y}(t), t)/\partial y_i(t)$. Intuitively, $e_i(t)$ measures how much a small change to y_i at time t affects E if everything else is left unchanged.

If we define z_i by the differential equation

$$\frac{dz_i}{dt} = \frac{1}{T_i} z_i - e_i - \sum_j \frac{1}{T_j} w_{ij} \sigma'(x_j) z_j \tag{2.2}$$

with boundary conditions $z_i(t_1) = 0$ then

$$\frac{\partial E}{\partial w_{ij}} = \frac{1}{T_j} \int_{t_0}^{t_1} y_i \sigma'(x_j) z_j dt \tag{2.3}$$

and

$$\frac{\partial E}{\partial T_i} = -\frac{1}{T_i} \int_{t_0}^{t_1} z_i \frac{dy_i}{dt} dt. \tag{2.4}$$

These results are derived using a finite difference approximation in (Pearlmutter 1988), and can also be derived using the calculus of variations and Lagrange multipliers (William Skaggs, personal communication) or from the continuous form of dynamic programming (Bryson 1962).

3 Simulation Results

Using first order finite difference approximations, we integrated the system $\mathbf{y}$ forward from t_0 to t_1, set the boundary conditions $z_i(t_1) = 0$, and integrated the system $\mathbf{z}$ backwards from t_1 to t_0 while numerically integrating $z_j\, \sigma'(x_j)\, y_i$ and $z_i\, dy_i/dt$, thus computing $\partial E/\partial w_{ij}$ and $\partial E/\partial T_i$.

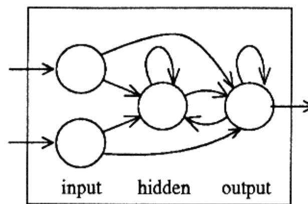

Figure 1: The XOR network.

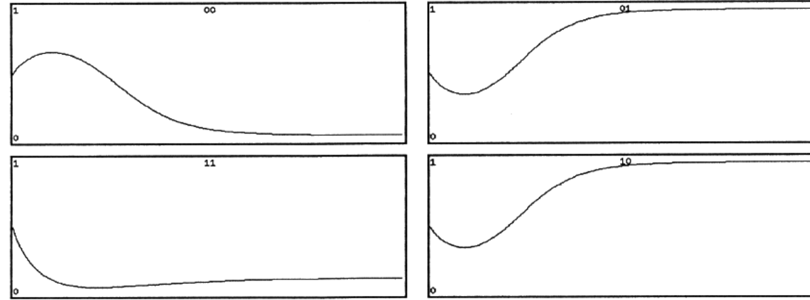

Figure 2: The states of the output unit in the four cases plotted from $t = 0$ to $t = 5$ after 200 epochs of learning. The error was computed only between $t = 2$ and $t = 3$.

Since computing dz_i/dt requires knowing $\sigma'(x_i)$, we stored it and replayed it backwards as well. We also stored and replayed y_i as it is used in expressions being numerically integrated.

We used the error functional

$$E = \frac{1}{2}\sum_i \int_{t_0}^{t_1} s_i(y_i - d_i)^2 dt \tag{3.1}$$

where $d_i(t)$ is the desired state of unit i at time t and $s_i(t)$ is the importance of unit i achieving that state at that time. Throughout, we used $\sigma(\xi) = (1 + e^{-\xi})^{-1}$. Time constants were initialized to 1, weights were initialized to uniform random values between 1 and -1, and the initial values $y_i(t_0)$ were set to $I_i(t_0) + \sigma(0)$. For these simulations we used $\Delta t = 0.1$.

All of these networks have an extra unit which has no incoming connections, an external input of 0.5, and outgoing connections to all other units. This unit provides a bias, which is equivalent to the negative of a threshold. This detail is suppressed below.

3.1 Exclusive Or. The network of figure 1 was trained to solve the XOR problem. Aside from the addition of time constants, the network topology was that used by Pineda in (Pineda 1987). We defined $E = \sum_k \frac{1}{2} \int_2^3 (y_o^{(k)} - d^{(k)})^2 dt$ where k ranges over the four cases, d is the correct output, and y_o is the state of the output unit. The inputs to the net $I_1^{(k)}$ and $I_2^{(k)}$ range over the four possible boolean combinations in the four different cases. With suitable choice of step size and momentum, training

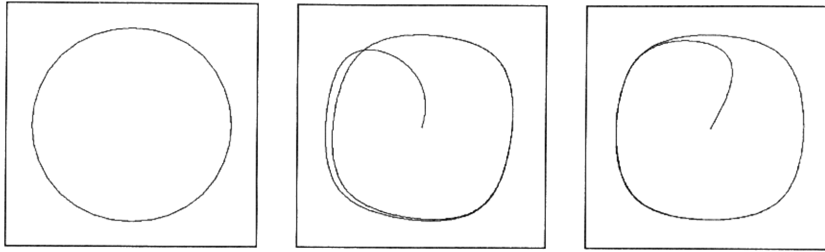

Figure 3: Desired states d_1 and d_2 plotted against each other (left); actual states y_1 and y_2 plotted against each other at epoch 1,500 (center) and 12,000 (right).

time was comparable to standard backpropagation, averaging about one hundred epochs.

It is interesting that even for this binary task, the network made use of dynamical behavior. After extensive training the network behaved as expected, saturating the output unit to the correct value. Earlier in training, however, we occasionally (about one out of every ten training sessions) observed the output unit at nearly the correct value between $t = 2$ and $t = 3$, but then saw it move in the wrong direction at $t = 3$ and end up stabilizing at a wildly incorrect value. Another dynamic effect, which was present in almost every run, is shown in figure 2. Here, the output unit heads in the wrong direction initially and then corrects itself before the error window. A very minor case of diving towards the correct value and then moving away is seen in the lower left-hand corner of figure 2.

3.2 A Circular Trajectory. We trained a network with no input units, four hidden units, and two output units, all fully connected, to follow the circular trajectory of figure 3. It was required to be at the leftmost point on the circle at $t = 5$ and to go around the circle twice, with each circuit taking 16 units of time. While unconstrained by the environment, the network moves from its initial position at $(0.5, 0.5)$ to the correct location at the leftmost point on the circular trajectory. Although the network was run for ten circuits of its cycle, these overlap so closely that the separate circuits are not visible.

Upon examining the network's internals, we found that it devoted three of its hidden units to maintaining and shaping a limit cycle, while the fourth hidden unit decayed away quickly. Before it decayed, it pulled the other units to the appropriate starting point of the limit cycle, and

after it decayed it ceased to affect the rest of the network. The network used different units for the limit behavior and the initial behavior, an appropriate modularization.

3.3 A Figure Eight. We were unable to train a network with four hidden units to follow the figure eight shape shown in figure 4, so we used a network with ten hidden units. Since the trajectory of the output units crosses itself, and the units are governed by first order differential equations, hidden units are necessary for this task regardless of the σ function. Training was more difficult than for the circular trajectory, and shaping the network's behavior by gradually extending the length of time of the simulation proved useful.

Before $t = 5$, while unconstrained by the environment, the network moves in a short loop from the initial position at $(0.5, 0.5)$ to where it should sit on the limit cycle at $t = 5$, namely $(0.5, 0.5)$. Although the network was run for ten circuits of its cycle to produce this graph, these overlap so closely that the separate circuits are not visible.

4 Embellishments

Adding time delays to the links simply adds analogous time delays to the differential equation for z. This approach can be used to learn modifiable time delays.

We can avoid the backwards pass by using a shooting method to update guesses for the correct values of $z_i(t_0)$ such that $z_i(t_1) = 0$ and integrating everything in the forward direction. Regretably, the computation required to compute the derivatives required by the shooting method seems excessive, and numeric stability is poor.

We can derive a "teacher forced" variant of our learning algorithm, presumably obtaining speedups similar to those reported by Williams and Zipser (1989).

It would be useful to have some characterization of the class of trajectories that a network can learn as a function of the number of hidden units. These networks have at least the representational power of Fourier decompositions, as one can use a pair of nodes to build an oscillator of arbitrary frequency by making use of the local linearity of the σ function (Furst 1988). We have also found simple bounds on d^2y/dt^2 based on the number of units, the largest weight, and the largest reciprocal time constant.

Experiments with perturbing the cyclic networks of section 3 shows that they have developed true limit cycles which attract neighboring states and pull them into the cycle. The oscillatory behavior of the two output units was not independent, but was coupled by the hidden units, which keep them phase locked even in the face of massive disruptions.

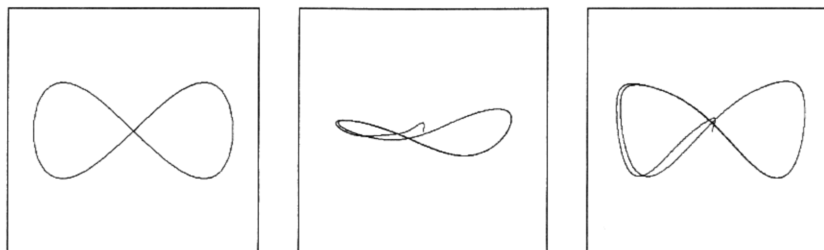

Figure 4: Desired states d_1 and d_2 plotted against each other (left); actual states y_1 and y_2 plotted against each other at epoch 3,182 (center) and 20,000 (right).

Further details on these and other related topics can be found in (Pearlmutter 1988).

5 Related Network Models

We use the same class of networks used by Pineda (1987), but he is concerned only with the limit behavior of these networks, and completely suppresses all other temporal behavior. His learning technique is applicable only when the network has a simple fixpoint; limit cycles or other non-point attractors violate a mathematical assumption upon which his technique is based.

We can derive Pineda's equations from ours. Let I_i be held constant, assume that the network settles to a fixpoint, let the initial conditions be this fixpoint, that is, $y_i(t_0) = y_i(\infty)$, and let E measure Pineda's error integrated over a short interval after t_0, with an appropriate normalization constant. As t_1 tends to infinity, (2.2) and (2.3) reduce to Pineda's equations, so in a sense our equations are a generalization of Pineda's; but these assumptions strain the analogy.

Jordan (1986) uses a conventional backpropagation network with the outputs clocked back to the inputs to generate temporal sequences. The treatment of time is the major difference between Jordan's networks and those in this work. The heart of Jordan's network is atemporal, taking inputs to outputs without reference to time, while an external mechanism is used to clock the network through a sequence of states in much the same way that hardware designers use a clock to drive a piece of combinatorial logic through a sequence of states. In our work, the network is not externally clocked; instead, it evolves continuously through time according to a set of coupled differential equations.

Most recently, Williams and Zipser (1989) have discovered an online learning procedure for networks of this sort. The tradeoffs between this procedure and that of Williams and Zipser is explored in some detail in (Pearlmutter 1988).

6 Acknowledgments

We thank Richard Szeliski for helpful comments and David Touretzky for unflagging support.

This research was sponsored in part by National Science Foundation grant EET-8716324, and by the Office of Naval Research under contract number N00014-86-K-0678. Barak Pearlmutter is a Fannie and John Hertz Foundation fellow.

References

Bryson, A.E., Jr. 1962. A steepest ascent method for solving optimum programming problems. *Journal of Applied Mechanics*, **29(2)**, 247.

Furst, M. 1988. Personal communication.

Jordan, M.I. 1986. Attractor dynamics and parallelism in a connectionist sequential machine. *In:* Proceedings of the 1986 Cognitive Science Conference, 531–546. Lawrence Erlbaum.

Pearlmutter, B. 1988. *Learning state space trajectories in recurrent neural networks.* Technical Report CMU-CS-88-191, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA.

Pineda, F. 1987. Generalization of backpropagation to recurrent neural networks. *Physical Review Letters*, **19(59)**, 2229–2232.

Rumelhart, D.E., G.E. Hinton, and R.J. Williams. 1986. Learning internal representations by error propagation. *In:* Parallel distributed processing: Explorations in the microstructure of cognition. Cambridge, MA: Bradford Books.

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

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

Received 17 October 1988; accepted 14 March 1989.