1988

Learning state space trajectories in recurrent neural networks

Barak Pearlmutter
Carnegie Mellon University

Published In
.

Learning State Space Trajectories in Recurrent Neural Networks

Barak A. Pearlmutter
December 31, 1988
CMU-CS-88-191

Abstract

We describe a number of procedures 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, so these procedures form the kernels of connectionist learning algorithms. Simulations in which networks are taught to move through limit cycles are shown. We also describe a number of elaborations of the basic idea, such as mutable time delays and teacher forcing, and conclude with a complexity analysis. This type of network seems particularly suited for temporally continuous domains, such as signal processing, control, and speech.

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. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of NSF, ONR, the Fannie and John Hertz Foundation or the U.S. Government.

1 Introduction

Note: this is an expanded version of an earlier paper of the same title [9].

Pineda [11] has shown how to train the fixpoints of a recurrent temporally continuous generalization of backpropagation networks [8,12,14]. Such networks are governed by the coupled differential equations

$$T_i \frac{dy_i}{dt} = -y_i + \sigma(x_i) + I_i \tag{1}$$

where

$$x_i = \sum_j w_{ji} y_j \tag{2}$$

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. The computation of $\partial E/\partial w_{ij}$ seems to require a phase in which the network is run backwards in time, and tricks for avoiding this are also described.

2 A Forward/Backward Technique

We can approximate the derivative in (1) with

$$\frac{dy_i}{dt}(t) \approx \frac{y_i(t+\Delta t) - y_i(t)}{\Delta t}, \tag{3}$$

which yields a first order difference approximation to (1),

$$\tilde{y}_i(t+\Delta t) = \left(1 - \frac{\Delta t}{T_i}\right) \tilde{y}_i(t) + \frac{\Delta t}{T_i} \sigma(\tilde{x}_i(t)) + \frac{\Delta t}{T_i} I_i(t). \tag{4}$$

We use tildes to indicate temporally discretized versions of continuous functions. The notation $\tilde{y}_i(t)$ is being used as shorthand for the particular variable representing the descrete version of $y_i(t_0 + n\Delta t)$, where n is an integer and $t = t_0 + n\Delta t$.

Let us define

$$e_i(t) = \frac{\delta E}{\delta y_i(t)}. \tag{5}$$

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.

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

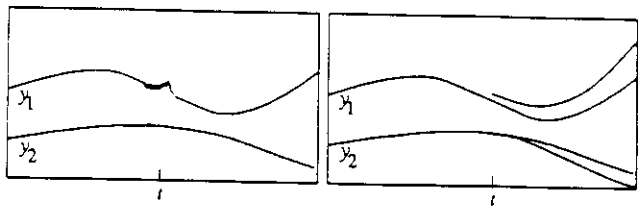

Figure 1: The infinitesimal changes to $\mathbf{y}$ considered in $e_1(t)$ (left) and $z_1(t)$ (right).

Let us define

$$\tilde{z}_i(t) = \frac{\partial^+ E}{\partial \tilde{y}_i(t)} \tag{6}$$

where the ∂^+ denotes an ordered derivative [15], with variables ordered by time. Intuitively, $\tilde{z}_i(t)$ measures how much a small change to $\tilde{y}_i$ at time t affects E when this change is propagated forward through time and influences the remainder of the trajectory, as in figure 1. Of course, z_i is the limit of $\tilde{z}_i$ as $\Delta t \to 0$.

We can use the chain rule for ordered derivatives to calculate $\tilde{z}_i(t)$ in terms of the $\tilde{z}_j(t+\Delta t)$. According to the chain rule, we add all the separate influences that varying $\tilde{y}_i(t)$ has on E. It has a direct contribution of $\Delta t e_i(t)$, which comprises the first term of our equation for $\tilde{z}_i(t)$. Varying $\tilde{y}_i(t)$ by ϵ has an effect on $\tilde{y}_i(t+\Delta t)$ of $\epsilon(1 - \Delta t/T_i)$, giving us a second term, namely $(1 - \Delta t/T_i)\tilde{z}(t+\Delta t)$.

Each weight w_{ij} allows $\tilde{y}_i(t)$ to influence $\tilde{y}_j(t+\Delta t)$. Let us compute this influence in stages; varying $\tilde{y}_i(t)$ by ϵ varies $\tilde{x}_j(t)$ by ϵw_{ij}, which varies $\sigma(\tilde{x}_j(t))$ by $\epsilon w_{ij}\sigma'(\tilde{x}_j(t))$, which varies $\tilde{y}_j(t+\Delta t)$ by $\epsilon w_{ij}\sigma'(\tilde{x}_j(t))\Delta t/T_j$. This gives us our third and final term, $\sum_j w_{ij}\sigma'(\tilde{x}_j(t))\Delta t \tilde{z}_j(t+\Delta t)/T_j$.

Combining these,

$$\begin{aligned} \tilde{z}_i(t) &= \Delta t\, e_i(t) + \left(1 - \frac{\Delta t}{T_i}\right) \tilde{z}_i(t+\Delta t) \\ &\quad + \sum_j w_{ij}\sigma'(\tilde{x}_j(t)) \frac{\Delta t}{T_j} \tilde{z}_j(t+\Delta t). \end{aligned} \tag{7}$$

If we put this in the form of (3) and take the limit as $\Delta t \to 0$ we obtain the differential equation

$$\frac{dz_i}{dt} = \frac{1}{T_i} z_i - e_i - \sum_j \frac{1}{T_j} w_{ij}\sigma'(x_j) z_j. \tag{8}$$

For boundary conditions note that by (5) and (6) $\tilde{z}_i(t_1) = \Delta t e_i(t_1)$, so in the limit as $\Delta t \to 0$ we have $z_i(t_1) = 0$.

Consider making an infinitesimal change dw_{ij} to w_{ij} for a period Δt starting at t. This will cause a corresponding infinitesimal change in E of

$$y_i(t)\sigma'(x_j(t)) \frac{\Delta t}{T_j} z_j(t) dw_{ij}.$$

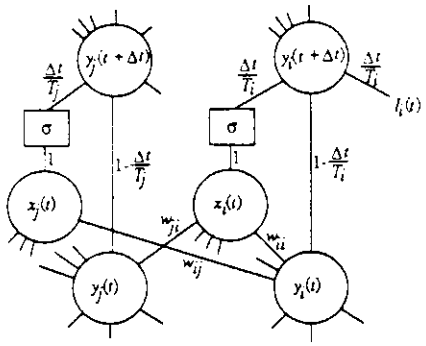

Figure 2: A lattice representation of (4).

Since we wish to know the effect of making this infinitesimal change to w_{ij} throughout time, we integrate over the entire interval yielding

$$\frac{\partial E}{\partial w_{ij}} = \frac{1}{T_j}\int_{t_0}^{t_1} y_i \sigma'(x_j) z_j dt. \tag{9}$$

If we substitute $\rho_i = T_i^{-1}$ into (4), find $\partial E/\partial \rho_i$ by proceeding analogously, and substitute T_i back in we get

$$\frac{\partial E}{\partial T_i} = -\frac{1}{T_i}\int_{t_0}^{t_1} z_i \frac{dy_i}{dt} dt. \tag{10}$$

One can also derive (8), (9) and (10) using the calculus of variations and Lagrange multipliers (William Skaggs, personal communication), or from the continuous form of dynamic programming [5].

3 Simulation Results

Using first order finite difference approximations, we integrated the system y forward from t_0 to t_1, set the boundary conditions $z_i(t_1) = 0$, and integrated the system 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 $\partial E/\partial T_i$. 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

$$E = \frac{1}{2}\sum_i \int_{t_0}^{t_1} s_i (y_i - d_i)^2 dt \tag{11}$$

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 uniformly distributed random values between 1 and -1, and the initial values $y_i(t_0)$ were set to $I_i(t_0) + \sigma(0)$. The simulator used the approximations (4) and (7) with $\Delta t = 0.1$.

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 3 was trained to solve the xor problem. Aside from the addition of time constants, the network topology was that used by Pineda in [11]. 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 time was comparable to standard backpropagation, averaging about one hundred epochs.

For this task it is to the network's benefit for units to attain their final values as quickly as possible, so there was a tendency to lower the time constants towards 0. In an effort to avoid small time constants, which degrade the numerical accuracy of the simulation, we introduced a term to decay the time constants towards 1. This decay factor was not used in the other simulations described below, and was not really necessary in this task if a suitably small Δt was used in the simulation.

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 4. 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 4.

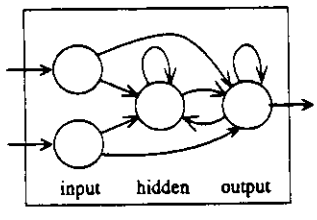

Figure 3: The XOR network.

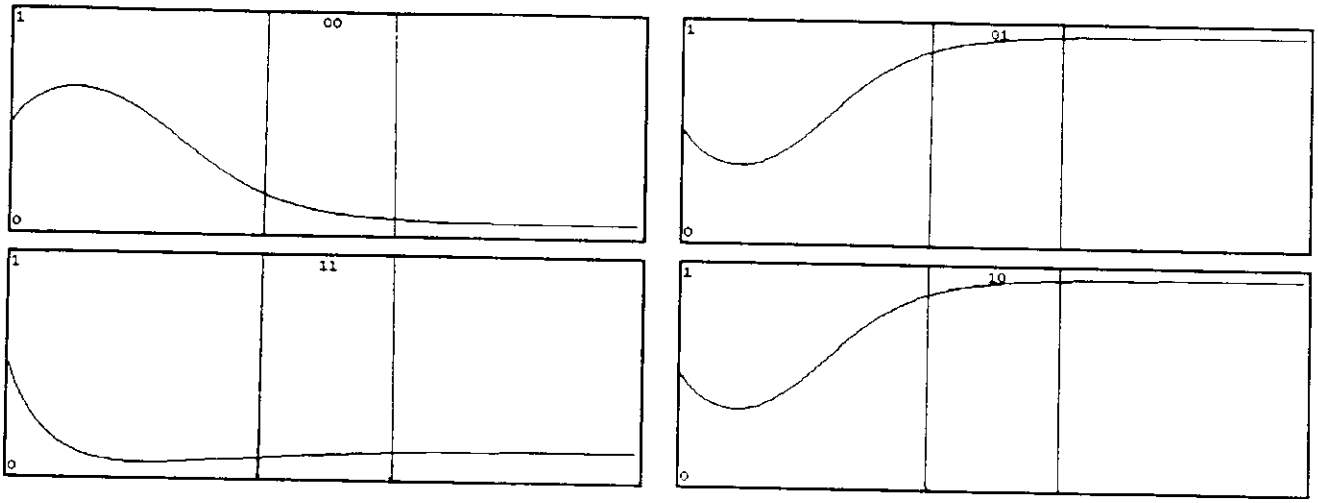

Figure 4: The states of the output unit in the four input cases plotted from $t = 0$ to $t = 5$ after 200 epochs of learning. The error was computed only between $t = 2$ and $t = 3$.

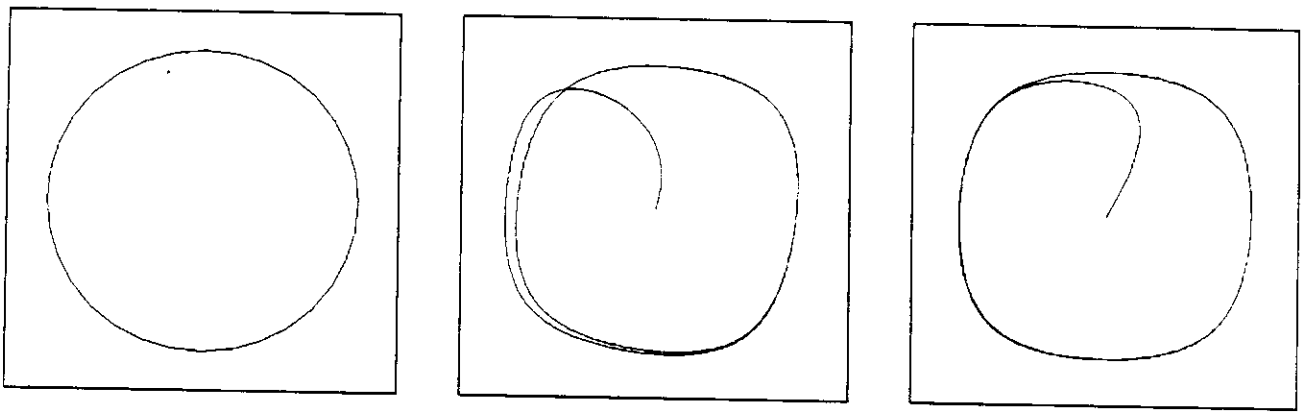

Figure 5: Desired states d_1 and d_2 plotted against each other (left); actual states y_1 and y_2 plotted against each other at epoch 1500 (center) and 12000 (right).

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 5. 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. The environment does not include desired outputs between $t = 0$ and $t = 5$, and during this period 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 6, 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.

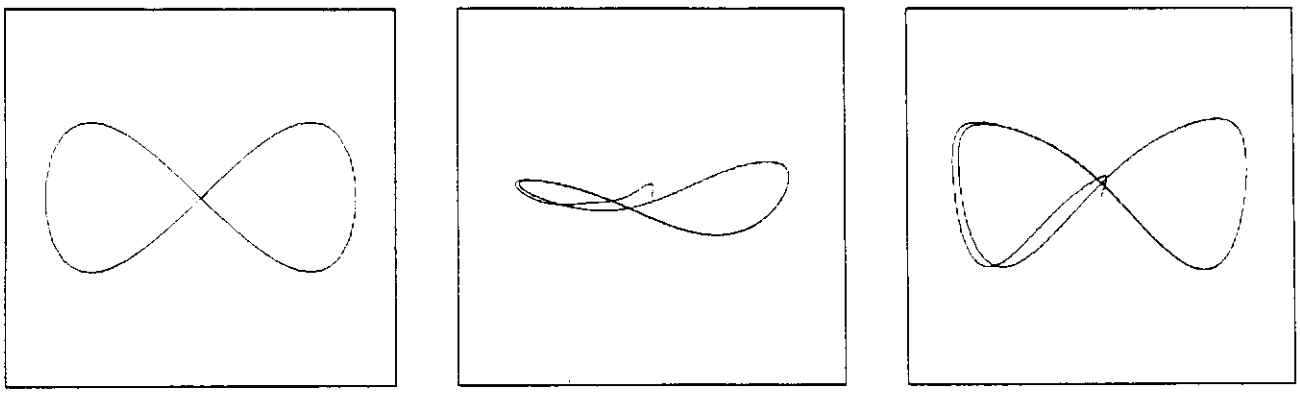

Figure 6: Desired states d_1 and d_2 plotted against each other (left); actual states y_1 and y_2 plotted against each other at epoch 3182 (center) and 20000 (right).

From $t = 0$ to $t = 5$ the network moves in a short loop from its initial position at $(0.5, 0.5)$ to where it ought to be at $t = 5$, namely $(0.5, 0.5)$. Following this, it goes through the figure eight shaped cycle every 16 units of time. 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.

3.4 Perturbation Experiments

In an attempt to judge the stability of the limit cycles exhibited above, we modified the simulator to introduce random perturbations and observed the effects of these perturbations upon the cycles. It is interesting to note that the two output units in the figure eight task appear to be phase locked, as their phase relationship remains invariant even in the face of major perturbations. This phase locking is unlike the solution that a human would wire up by hand.

The limit cycle on the right in figure 6 is symmetric, but when perturbations are introduced, as in the right of figure 7, symmetry is broken. The portion of the limit cycle moving from the upper left hand corner towards the lower right hand corner has diverging lines, but we do not believe that they indicate high eigenvalues and instability. The lines converge rapidly in the upward stroke on the right hand side of the figure, and analogous unstable behavior is not present in the symmetric downward stroke from the upper right hand corner towards the lower left. Analysis shows that the instability is caused by the initialization circuitry being inappropriately activated; since the initialization circuitry is adapted for controlling just the initial behavior of the network, when the net must delay at $(0.5, 0.5)$ for a time before beginning the cycle by moving lowards the lower left corner, this circuitry is explicitly not symmetric. The diverging lines seem to be caused by this circuitry being activated and exerting a strong influence on the output units while the circuitry itself deactivates.

4 Embellishments

4.1 Time Delays

Consider a network of this sort in which signals take finite time to travel over each link, so that (2) is modified to

$$x_i(t) = \sum_j w_{ji} y_j(t - \tau_{ji}), \tag{12}$$

τ_{ji} being the time delay along the connection from unit j to unit i. Surprisingly, such time delays merely add analogous time delays to (8) and (9),

$$\frac{dz_i}{dt}(t) = \frac{1}{T_i} z_i(t) - e_i(t) - \sum_j w_{ij} \sigma'(x_j(t+\tau_{ij})) \frac{1}{T_j} z_j(t+\tau_{ij}), \tag{13}$$

$$\frac{\partial E}{\partial w_{ij}} = \frac{1}{T_j} \int_{t_0}^{t_1} y_i(t) \sigma'(x_j(t+\tau_{ij})) z_j(t+\tau_{ij}) dt, \tag{14}$$

while (10) remains unchanged. If we set $\tau_{ij} = \Delta t$, these modified equations alleviate concern over time skew when simulating networks of this sort, obviating the need for predictor/corrector methods.

Instead of regarding the time delays as a fixed part of the architecture, we can imagine modifiable time delays. Given modifiable time delays, we would like to be able to learn appropriate values for them, which can be accomplished using gradient descent by

$$\frac{\partial E}{\partial \tau_{ij}} = \int_{t_0}^{t_1} z_j(t) \sigma'(x_j(t)) w_{ij} \frac{dy_i}{dt}(t - \tau_{ij}) dt. \tag{15}$$

We have not yet simulated networks with modifyable time delays.

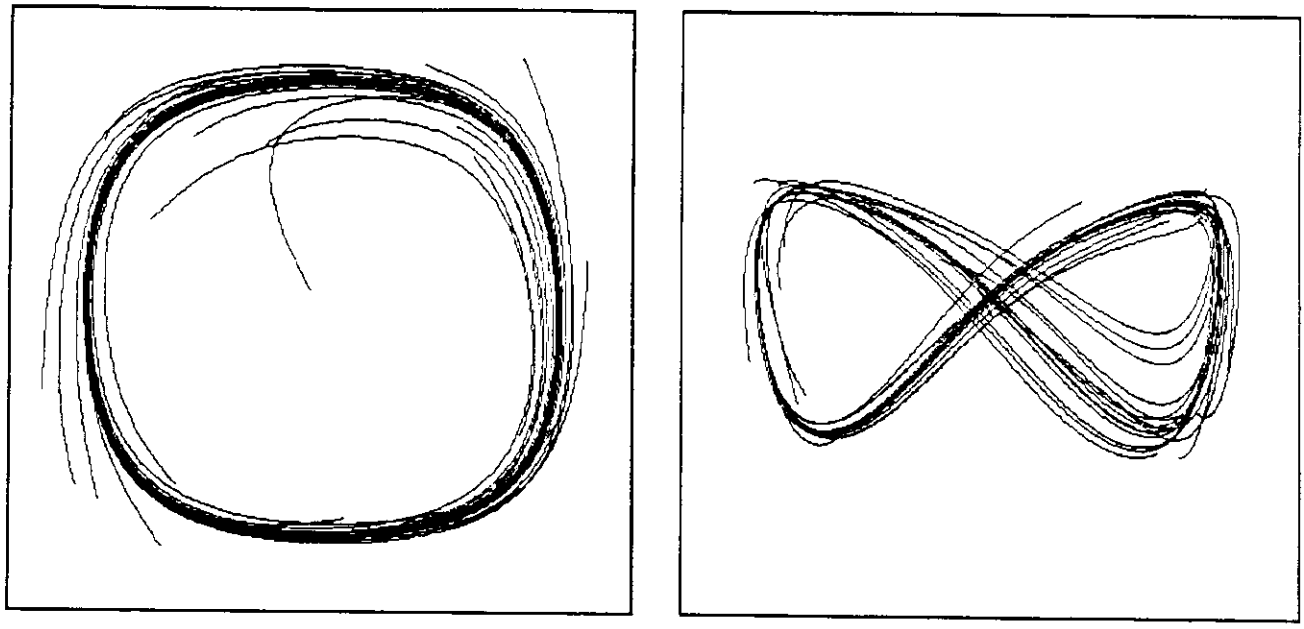

Figure 7: The output states y_1 and y_2 plotted against each other for a 1000 time unit run, with all the units in the network perturbed by a random amount about every 40 units of time. The perturbations in the circle network (left) were of magnitude less than 0.1, and in the eight network (right) of magnitude less than 0.05.

An interesting class of architectures would have the state of one unit modulate the time delay along some arbitrary link in the network or the time constant of some other unit. Such architectures seem appropriate for tasks in which time warping is an issue, such as speech recognition, and such architectures can certainly be accommodated by our approach.

In the presence of time delays, it is reasonable to have more than one connection between a single pair of units, with different time delays along the different connections. Such "time delay neural networks" have proven useful in the domain of speech recognition [7,13]. Having more than one connection from one unit to another requires us to modify our notation somewhat; weights and time delays are modified to take a single index, and we introduce some external apparatus to specity the source and destination of each connection. Thus w_i is the weight on a connection between unit $\mathcal{L}(i)$ and unit $\mathcal{R}(i)$, and τ_i is the time delay along that connection. Using this notation we write (12) as

$$x_i(t) = \sum_{j \mid \mathcal{L}(j)=i} w_j y_{\mathcal{R}(j)}(t - \tau_j).$$

Our equations would be more general if written in this notation, but readability would suffer, and the translation is quite mechanical.

4.2 Avoiding the Backwards Pass

As mentioned in section 3, the obvious way to simulate these networks is to start at t_0, simulate $\mathbf{y}$ forward to t_1 while storing it, set $\mathbf{z}(t_1) = 0$ and simulate $\mathbf{z}$ backwards from t_1 to t_0 while replaying $\mathbf{y}$. While simulating backwards, we numerically integrate according to equations (9) and (10), thereby computing the partials of E. However, this requires simulating backwards in time, which is not pleasing, and it requires remembering the trajectory of $\mathbf{y}$, which takes storage linear in $t_1 - t_0$. One way to avoid storing the trajectory of $\mathbf{y}$ is to simulate it backwards as we simulate $\mathbf{z}$ backwards, but note that simulating $\mathbf{y}$ backwards is typically numerically unstable.

Here, we consider the alternative of guessing $\mathbf{z}(t_0)$ such that $\mathbf{z}(t_1) = 0$ and doing all of our simulations forward through time. This is not attractive on serial machines with plentiful memory, but might be more attractive on parallel machines with limited storage. These complexity issues are discussed in section 5.2.

Let us find a way to compute $\partial z_i(t_1)/\partial z_j(t_0)$. We define

$$\zeta_{ij}(t) = \frac{\partial z_i(t)}{\partial z_j(t_0)} \tag{16}$$

and take the partial of (8) with respect to $z_j(t_0)$, substituting in ζ_{ij} where appropriate. This results in a differential

equation for ζ_{ij},

$$\frac{d\zeta_{ij}}{dt} = \frac{1}{T_i}\zeta_{ij} - \sum_k \frac{1}{T_k} w_{ik}\sigma'(x_k)\zeta_{kj}, \tag{17}$$

and for boundary conditions we note that

$$\zeta_{ij}(t_0) = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{otherwise.} \end{cases} \tag{18}$$

Given guesses for the correct value of $\mathbf{z}(t_0)$, we will simulate $\mathbf{y}$, $\mathbf{z}$ and ζ forward from t_0 to t_1 and then update the guesses in order to minimize B where

$$B = \frac{1}{2}\sum_i z_i(t_1)^2 \tag{19}$$

with a shooting method by making use of the fact that

$$\frac{\partial B}{\partial z_j(t_0)} = \sum_i z_i(t_1)\zeta_{ij}(t_1). \tag{20}$$

For notational convenience, let $b_i = \partial B/\partial z_i(t_0)$. We can use a Newton-Raphson method with the appropriate modification for the fact that B has a minimum of zero, resulting in the simple update rule

$$z_i(t_0) \leftarrow z_i(t_0) - 2\frac{B}{||\mathbf{b}||^2} b_i. \tag{21}$$

During our simulation we accumulate the appropriate integrals, so if our guesses for $z_i(t_0)$ were nearly correct we will have computed nearly correct values for $\partial E/\partial w_{ij}$ and $\partial E/\partial T_i$. If the w_{ij} change slowly the correct values for $z_i(t_0)$ will change slowly, so tolerable accuracy can be obtained by using the $\partial E/\partial w_{ij}$ computed from the slightly incorrect values for $z_i(t_0)$ while simultaneously updating the $z_i(t_0)$ for future use, eliminating the need for an inner loop which iterates to find the correct values for the $z_i(t_0)$. This argument assumes that the quadratic convergence of the Newton-Raphson method dominates the linear divergence of the changes to the w_{ij}, which can be guaranteed by choosing suitably low learning parameters. Regrettably, it also assumes that the forward simulation of $\mathbf{z}$ is numerically stable enough for our purposes, which is typically not the case for long trajectories.

4.3 An Online Variation

We can use the technique of Williams and Zipser [16] to create an online version of our algorithm. Let us define

$$p_{ij}^k(t) = \frac{\partial y_k(t)}{\partial w_{ij}} \tag{22}$$

and note that

$$\frac{\partial E}{\partial w_{ij}} = \int_{t_0}^{t_1} \sum_k e_k p_{ij}^k. \tag{23}$$

If we begin with (1), substitute k for i, take the partial with respect to w_{ij}, and substitute in p where possible, we have a differential equation for p,

$$T_k \frac{p_{ij}^k}{dt} = -p_{ij}^k + \sigma'(x_k)\sum_l w_{lk} p_{ij}^l, \tag{24}$$

which is stable in the forward direction. To construct an online algorithm we simulate the systems $\mathbf{y}$ and $\mathbf{p}$ forward through time and continuously update the weights to do gradient descent using (23), spreading the weight update across time using the continuous update rule

$$\frac{dw_{ij}}{dt} = -\epsilon \sum_k e_k p_{ij}^k. \tag{25}$$

We can derive analogous equations for the time constants; define

$$q_j^i(t) = \frac{\partial y_i(t)}{\partial T_j}, \tag{26}$$

take the partial of (1) with respect to T_j, and substitute in q. This yields

$$T_i \frac{dq_j^i}{dt} = -q_j^i - \frac{dy_i}{dt} + \sigma'(x_i)\sum_k w_{ki} q_j^k \tag{27}$$

which can be used to update the time constants using the continuous update rule

$$\frac{dT_i}{dt} = -\epsilon \sum_j e_j q_i^j. \tag{28}$$

Similarly, let us derive equations for modifying the time delays of section 4.1. Define

$$r_{ij}^k(t) = \frac{\partial y_k(t)}{\partial \tau_{ij}} \tag{29}$$

and take the partial of (1) with respect to τ_{ij}, arriving at a differential equations for r,

$$T_k \frac{dr_{ij}^k}{dt} = -r_{ij}^k + \sigma'(x_k)(\underbrace{w_{ij}\frac{dy_i}{dt}(t-\tau_{ij})}_{\text{included if } j=k} - \sum_l w_{lk} r_{ij}^l(t-\tau_{ik})). \tag{30}$$

The time delays can be updated online using the continuous update equation

$$\frac{d\tau_{ij}}{dt} = -\epsilon \sum_k e_k r_{ij}^k. \tag{31}$$

4.4 Teacher Forcing

Williams and Zipser report that their teacher forcing technique radically improves learning time in recurrent networks [16]. Teacher forcing involves using the training signal to modify the states of units to desired values as the network is run. Williams and Zipser's application of teacher forcing to their networks is deeply dependent on discrete time steps, so applying teacher forcing to temporally continuous networks requires a different approach.

The essential idea is that we will add some knobs that can be used to control the states of the output units, we will use them to keep the output units locked at some desired states, and we will minimize an error functional which measures the amount of control we have exerted.

Let

$$F_i = \frac{1}{T_i}(-y_i + \sigma(x_i) + I_i) \tag{32}$$

so that (1) is just $dy_i/dt = F_i$, and add a new forcing term $f_i(t)$ to (1),

$$\frac{dy_i}{dt} = F_i + f_i. \tag{33}$$

Let the set of forced units be Φ. For each $i \in \Phi$ let d_i be the trajectory that we will force y_i to follow, so we set

$$f_i = \frac{dd_i}{dt} - F_i \tag{34}$$

and $y_i(t_0) = d_i(t_0)$ for $i \in \Phi$ and $f_i = 0$ for $i \notin \Phi$, with the consequence that $y_i = d_i$ for $i \in \Phi$. Now let the error functional be of the form

$$E = \int_{t_0}^{t_1} L(f_1, \ldots, f_n, t)dt, \tag{35}$$

where typically $L = \sum_i f_i^2$.

We can modify the derivation in section 2 for this "teacher forced" system. For $i \in \Phi$ a change to $\tilde{y}_i$ will be canceled immediately, so taking the limit as $\Delta t \to 0$ yields $z_i = 0$. Because of this, it doesn't matter what e_i is for $i \in \Phi$.

We can apply (5) to calculate e_i for $i \notin \Phi$. The chain rule is used to calculate how a change in y_i effects E through the f_i, yielding

$$e_i = \sum_{j \in \Phi} \frac{\delta E}{\delta f_j} \frac{\partial f_j}{\partial y_i}$$

or

$$e_i = \sum_{j \in \Phi} \frac{\partial L}{\partial f_j} - \frac{1}{T_j}\sigma'(x_j)w_{ij} \tag{36}$$

For $i \notin \Phi$ (8) and (10) are unchanged, and for $j \notin \Phi$ and any i (9) also remains unchanged. The only equations still required are $\partial E/\partial w_{ij}$ for $j \in \Phi$ and $\partial E/\partial T_i$ for $i \in \Phi$. To derive the first, consider the instantaneous effect of a small change to w_{ij}, giving

$$\frac{\partial E}{\partial w_{ij}} = \frac{1}{T_j}\int_{t_0}^{t_1} y_i\sigma'(x_j)\frac{\partial L}{\partial f_i}dt. \tag{37}$$

Analogously, for $i \in \Phi$

$$\frac{\partial E}{\partial T_i} = -\frac{1}{T_i}\int_{t_0}^{t_1} \frac{\partial L}{\partial f_i}\frac{dy_i}{dt}dt. \tag{38}$$

We are left with a system with a number of special cases depending on whether units are in Φ or not. Interestingly, an equivalent system results if we leave (8), (9), and (10) unchanged except for setting $z_i = \partial L/\partial f_i$ for $i \in \Phi$ and setting all the $e_i = 0$. It is an open question as to whether there is some other way of defining z_i and e_i that results in this simplification.

5 Analysis

5.1 Computational Power

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. We are investigating this area, and have some preliminary results. 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, so one can take the first n terms of a function's Fourier decomposition and analytically find a set of weights for a network with $2n + 1$ nodes that generates this approximation to the function (Merrick Furst, personal communication).

We can also derive some fairly straightforward bounds on the possible ranges of the states and their derivatives. We use n for the number of units in the network, and the notation $\max|I|$ is used to delimit the maximum absolute value attainable by any I_i.

$$\max|y| \leq \max|\sigma| + \max|I| \tag{39}$$

$$\max\left|\frac{dy}{dt}\right| \leq 2 \max\left|\frac{1}{T}\right| \max|y| \tag{40}$$

$$\max\left|\frac{d^2y}{dt^2}\right| \leq \max\left|\frac{1}{T}\right| \left(\max\left|\frac{dy}{dt}\right| (1 + n \max|\sigma'| \max|w|) + \max\left|\frac{dI}{dt}\right|\right) \tag{41}$$

This bounds the rate at which the network's state can change, and the rate at which its velocity vector can change, thus limiting the class of trajectories that may be learned. But it does not limit the complexity (number of squiggles) of a trajectory, provided it is sufficiently slow moving. A stronger notion of trajectory complexity would be desirable.

5.2 Complexity

Consider a network with n units and m weights which is run for s time steps[2] where $s = (t_1 - t_0)/\Delta t$. Additionally, assume that the computation of each $e_i(t)$ is $O(1)$ and that the network is not partitioned.

Under these conditions, simulating the **y** system takes $O(m+n) = O(m)$ time for each time step, as does simulating the **z** system. This means that using the technique described in section 3, the entire simulation takes $O(m)$ time per time step, the best that could be hoped for. Storing the activations and weights takes $O(n+m) = O(m)$ space, and storing **y** during the forward simulation to replay while simulating **z** backwards takes $O(sn)$ space, so if we use this technique the entire computation takes $O(sn+m)$ space. If we simulate **y** backwards during the backwards simulation of **z**, the simulation requires $O(n+m)$ space, again the best that could be hoped for. This later technique, however, is susceptible to numeric stability problems.

Maintaining the ζ_{ij} terms of section 4.2 takes $O(nm)$ time each time step, and $O(n^2)$ space. These are the dominant factors in the calculation of the partials of B. The technique of Williams and Zipser described in section 4.3 requires $O(n^2m)$ time each time step, and $O(nm)$ space.

These time complexity results are for sequential machines, and are summarized in table 1. All these algorithms are embarrassingly parallel and eminently suitable for implementation on both vector processors and highly parallel machines.

5.3 Stability

We can analytically determine the stability of the network by measuring the eigenvalues λ_i of Df where f is the function that maps the state of the network at one point in time to its state at a later time. For instance, for a network exhibiting a limit cycle one would typically use the function that maps the network's state at some time in the cycle to its state at the corresponding time in the next cycle. It is tempting to introduce a term to be minimized which rewards the network for being stable, for instance $\sum_i \lambda_i^4$ where λ_i is an eigenvalue of Df. Regrettably, computing Df is costly, so we are investigating ways to add terms to

[2]Variable grid methods [2] can reduce s by dynamically varying Δt.

technique	store y	back y	shooting	W&Z
time	$O(m)$	$O(m)$	$O(nm)$	$O(n^2m)$
space	$O(sn+m)$	$O(m)$	$O(n^2+m)$	$O(nm)$
online?	no	no	semi	yes
stable?	yes	no	no	yes
local?	yes	yes	no	no

Table 1: A summary of the complexity of some learning procedures for recurrent networks. In the "store y" technique we store y as time is run forwards and replay it as we run time backwards computing z. In "back y" we do not store y, instead recomputing it as time is run backwards. W&Z is the technique of Williams and Zipser.

E which measure weaker but more economically computed criteria of stability than $\max_i |\lambda_i| < 1$, such as

$$\left(\frac{\mathrm{Tr}(Df)}{1+\mathrm{Det}(Df)}\right)^2.$$

We conjecture that the apparent noise tolerance shown in the simulations of section 3.4 is caused by the learning algorithm running in the presence of noise introduced by the conversion from differential equations to difference equations and perhaps floating point roundoff errors. This leads to the thought of enhancing the stability of the solutions that the learning algorithm derives by deliberately injecting noise into the system during training, thus punishing the algorithm for even short stretches of instability.

6 Future Work

Our next experiments will involve using inputs to specify a member of a class of continuous tasks, and testing generalization to novel inputs. We will make a network with two inputs and two outputs, where the inputs are used to specify the radius and cycle time of a circle to be traced out on the two output units. After that, we would like to experiment with more complex error functionals, involving dy_i/dt and correspondences between states at different points in time. We also plan on simulating networks with adjustable time delays, something which we have not experimented with at all to date.

In the longer term, there are obvious applications to identification and control, some of which will be explored in the author's thesis research. Signal processing and speech generation and recognition (using generative techniques) are also domains to which this type of network can be naturally applied. Such domains may lead us to complex

architectures like those discussed in section 4.1. For control domains, it seems important to have ways to force the learning towards solutions that are stable in the control sense of the term, so we are attempting to develop the ideas hinted at in section 5.3 into workable additions to the learning algorithm.

On the other hand, we can turn the logic of section 5.3 around. Consider a difficult contraint satisfaction task of the sort that neural networks are sometimes applied to, such as the traveling salesman problem [3]. Two competing techniques for such problems are simulated annealing [6,1] and mean field theory [10]. By providing a network with a noise source which can be modulated (by second order connections, say) we could see if the learning algorithm constructs a network that makes use of the noise to generate networks that do simulated annealing, or if pure gradient descent techniques are evolved. If a hybrid network evolves, its structure may give us insight into the relative advantages of these two different optimization techniques.

7 Relation to Other Work

We use the same class of networks used by Pineda [11], 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, i.e., $y_i(t_0) = y_i(\infty)$, 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, (8) and (9) reduce to Pineda's equations, so in a sense our equations are a generalization of Pineda's; but these assumptions strain the analogy.

Jordan [4] 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 though a sequence of states. In our work, the network is not externally clocked; instead, it evolves continuously though time according to a set of coupled differential equations.

Williams and Zipser [16] have discovered an online learning procedure for networks of this sort; a derivation of their technique is given in section 4.3 above.

8 Acknowledgments

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

References

[1] David H. Ackley, Geoffrey E. Hinton, and Terry J. Sejnowski. A learning algorithm for Boltzmann Machines. *Cognitive Science*, 9,:147–169, 1985.

[2] J. G. Blom, J. M. Sanz-Serna, and Jan G. Verwer. *On Simple Moving Grid Methods for One-Dimensional Evolutionary Partial Differential Equations.* Stichting Mathematisch Centrum, Amsterdam, The Netherlands, 1986.

[3] J. J. Hopfield and D. W. Tank. 'Neural' computation of decisions in optimization problems. *Biological Cybernetics*, 52:141–152, 1985.

[4] Michael I. Jordan. Attractor dynamics and parallelism in a connectionist sequential machine. In *Proceedings of the 1986 Cognitive Science Conference*, Lawrence Erlbaum, 1986.

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

[6] S. Kirkpatrick, C. D. Gelatt, Jr., and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680. 1983.

[7] Kevin Lang and Geoffrey Hinton. *The Development of the Time-Delay Neural Network Architecture for Speech Recognition.* Technical Report CMU-CS-88-152, Department of Computer Science, Carnegie Mellon University, November 1988.

[8] David B. Parker. *Learning-Logic.* Technical Report TR-47, MIT Center for Research in Computational Economics and Management Science, Cambridge, MA, 1985.

[9] Barak Pearlmutter. Learning state space trajectories in recurrent neural networks. In *Proceedings of the 1988 Connectionist Models Summer School*, Morgan Kaufman, San Mateo, CA, 1988.

[10] C. Peterson and James R. Anderson. *A Mean Field Theory Learning Algorithm for Neural Networks.* Technical Report EI-259-87, MCC, August 1987.

[11] Fernando Pineda. Generalization of back-propagation to recurrent neural networks. *Physical Review Letters*, 19(59):2229–2232, 1987.

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

[13] Alex Waibel, Toshiyuki Hanazawa, Geoffrey Hinton, Kiyohiro Shikano, and Kevin Lang. *Phoneme Recognition Using Time-Delay Neural Networks*. Technical Report TR-1-0006, ATR, 1987.

[14] Paul J. Werbos. *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences*. PhD thesis, Harvard University, 1974.

[15] Paul J. Werbos. Generalization of backpropagation with application to a recurrent gas market model. *Neural Networks*, 1:339–356, 1988.

[16] Ronald J. Williams and David Zipser. *A Learning Algorithm for Continually Running Fully Recurrent Neural Networks*. Technical Report ICS Report 8805, UCSD, La Jolla, CA 92093, November 1988.