Dynamic Recurrent Neural Networks

Barak A. Pearlmutter

December 1990

CMU-CS-90-196

(supersedes CMU-CS-88-191)

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

We survey learning algorithms for recurrent neural networks with hidden units and attempt to put the various techniques into a common framework. We discuss fixpoint learning algorithms, namely recurrent backpropagation and deterministic Boltzmann Machines, and non-fixpoint algorithms, namely backpropagation through time, Elman's history cutoff nets, and Jordan's output feedback architecture. Forward propagation, an online technique that uses adjoint equations, is also discussed. In many cases, the unified presentation leads to generalizations of various sorts. Some simulations are presented, and at the end, issues of computational complexity are addressed.

This research was sponsored in part by The Defense Advanced Research Projects Agency, Information Science and Technology Office, under the title "Research on Parallel Computing", ARPA Order No. 7330, issued by DARPA/CMO under Contract MDA972-90-C-0035 and in part by the National Science Foundation under grant number EET-8716324 and in part by the Office of Naval Research under contract number N00014-86-K-0678. The author held a Fannie and John Alexander Hertz Fellowship.

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 the Hertz Foundation or the U.S. government.

Keywords: learning, sequences, temporal structure, recurrent neural networks, fixpoints

Contents

1 Introduction **1**
- 1.1 Why Recurrent Networks . . . 1
- 1.2 Why Hidden Units . . . 1
- 1.3 Continuous vs. Discrete Time . . . 2

2 Learning in Networks with Fixpoints **3**
- 2.1 Will a Fixpoint Exist? . . . 3
- 2.2 Problems with Fixpoints . . . 4
- 2.3 Recurrent Backpropagation . . . 5
  - 2.3.1 Simulation of an Associative Network . . . 6
- 2.4 Deterministic Boltzmann Machines . . . 7

3 Backpropagation Through Time **9**
- 3.1 Time Constants . . . 11
- 3.2 Time Delays . . . 12
- 3.3 Some Simulations . . . 13
  - 3.3.1 Exclusive Or . . . 13
  - 3.3.2 A Circular Trajectory . . . 15
  - 3.3.3 A Figure Eight . . . 15
  - 3.3.4 A Rotated Figure Eight . . . 16
- 3.4 Stability and Perturbation Experiments . . . 17
- 3.5 Leech Simulations . . . 18

4 Other Non-fixpoint Techniques **18**
- 4.1 "Elman Nets" . . . 18
- 4.2 The Moving Targets Method . . . 18
- 4.3 Forward Propagation . . . 19
  - 4.3.1 Extending Online Learning to Time Constants and Delays . . . 20
  - 4.3.2 Faster Online Techniques . . . 21
- 4.4 Feedforward Networks with State . . . 21

5 Teacher Forcing **21**
- 5.1 In Continuous Time . . . 22
- 5.2 "Jordan Nets" . . . 23
- 5.3 Continuous Time "Jordan Nets" . . . 23

6 Summary and Conclusion **24**
- 6.1 Complexity Comparison . . . 24
- 6.2 Future Work . . . 25
- 6.3 Conclusions . . . 25
- 6.4 Acknowledgments . . . 25

1 Introduction

1.1 Why Recurrent Networks

The subject of this document is training recurrent neural networks. The problem of training non-recurrent, layered architectures has been covered adequately elsewhere, and will not be discussed here.

The motivation for exploring recurrent architectures is their potential for dealing with two sorts of temporal behavior. First of all, recurrent networks are capable of settling to a solution, as in vision system which gradually solve a complex set of conflicting constraints to arrive at an interpretation. Although this is discussed to some extent below, we are primarily concerned with the problem of causing networks to exhibit particular desired detailed temporal behavior, as in the modeling of a central pattern generator of an insect.

It should be noted that many real-world problems which one might think would require recurrent architectures for their solution seem soluble with layered architectures; for this reason, we would urge engineers to try layered architectures first before resorting to the "big gun" of recurrence.

1.2 Why Hidden Units

We will restrict our attention to training procedures for networks with *hidden units,* units which have no particular desired behavior, are not directly involved in the input or output of the network. For the biologically inclined, they can be thought of as interneurons.

With the practical successes of backpropagation, it seems gratuitous to expound the virtues of hidden units and internal representations. Hidden units make is possible for networks to discover and exploit regularities of the task at hand, such as symmetries or replicated structure [15], and training procedures for exploiting hidden units, such as backpropagation, [18, 44] are behind much of the current excitement in the neural networks field. Also, training algorithms that do not operate with hidden units, such as the Widrow-Hoff LMS rule procedure [51], can be used to train recurrent networks without hidden units, so recurrent networks without hidden units reduce to non-recurrent networks without hidden units, and therefore do not need special learning algorithms.

Consider a neural network governed by the equations

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

where y_i is the state or activation level of unit i,

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

is the total input to unit i, w_{ij} is the strength of the connection from unit i to unit j, and σ is an arbitrary differentiable function. (Typically the function chosen is either the squashing function $\sigma(\xi) = (1 + e^{-\xi})^{-1}$, in which case $\sigma'(\xi) = \sigma(\xi)(1 - \sigma(\xi))$, or $\sigma(\xi) = \tan^{-1}(\xi)$, in

which case $\sigma'(\xi) = (1+\sigma(\xi))(1-\sigma(\xi))$. Even though the latter symmetric squashing function is usually preferable, as it has a number of computational advantages, the former was used in all the simulations presented below.) The initial conditions $y_i(t_0)$ and driving functions $I_i(t)$ are the inputs to the system.

This defines a rather general dynamic system. Even assuming that the external input terms $I_i(t)$ are held constant, it is possible for the system to exhibit a wide range of asymptotic behaviors. The simplest is that the system reaches a stable fixpoint; in the next section, we will discuss two different techniques for modifying the fixpoints of networks that exhibit them.

More complicated possible asymptotic behaviors include limit cycles and even chaos. Later, we will describe a number of training procedures that can be applied to training networks to exhibit desired limit cycles, or particular detailed temporal behavior. Although it has been theorized that chaotic dynamics play a significant computational role in the brain [11], there are no training procedures for chaotic attractors in networks with hidden units. However, Crutchfield et al. [8] and Lapedes and Farber [27] have had success with the identification of chaotic systems using models without temporally hidden units.

1.3 Continuous vs. Discrete Time

We will be concerned predominantly with continuous time networks, as in (1). However, all of the learning procedures we will discuss can be equally well applied to discrete time systems, which obey equations like

$$y_i(t+1) = \sigma(x_i(t)) + I_i(t). \tag{3}$$

Continuous time has advantages for expository purposes, in that the derivative of the state of a unit with respect to time is well defined, allowing calculus to be used instead of tedious explicit temporal indexing, making for simpler derivations and exposition.

When a continuous time system is simulated on a digital computer, it is usually converted into a set a simple first order difference equations, which is formally identical to a discrete time network. However, regarding the discrete time network running on the computer as a simulation of a continuous time network has a number of advantages. First, more sophisticated and faster simulation techniques than simple first order difference equations can be used, such as higher order forward-backward techniques. Second, even if simple first order equations are used, the size of the time step can be varied to suit changing circumstances; for instance, if the network is being used for a signal processing application and faster sensors and computers become available, the size of the time step could be decreased without retraining the network. Third, because continuous time units are stiff in time, they tend to retain information better through time. Another way of putting this is that their bias in the learning theory sense is towards temporally continuous tasks, which is certainly advantageous if the task being performed is also temporally continuous.

Another advantage of continuous time networks is somewhat more subtle. Even for tasks which themselves have no temporal content, such as constraint satisfaction, the best way for a recurrent network to perform the required computation is for each unit to represent nearly

the same thing at nearby points in time. Using continuous time units makes this the default behavior; in the absence other forces, units will tend to retain their state through time. In contrast, in discrete time networks, there is no a-priori reason for a unit's state at one point in time to have any special relationship to its state at the next point in time.

A pleasant benefit of units tending to maintain their state through time is that it helps make information about the past decay more slowly, speeding up learning about the relationship between temporally distant events.

2 Learning in Networks with Fixpoints

The fixpoint learning algorithms we will discuss assume that the networks involved converge to stable fixpoints.[1] Networks that converge to fixpoints are interesting because of the class of things they can compute, like constraint satisfaction and associative memory tasks. In such tasks, the problem is usually given to the network either by the initial conditions or by a constant external input, and the answer is given by the state of the network once it has reached its fixpoint. This is precisely analogous to the relaxation algorithms used to solve such things as steady state heat equations, except that the constraints need not have spatial structure or uniformity.

2.1 Will a Fixpoint Exist?

One problem with fixpoints is that recurrent networks do not always converge to them. However, there are a number of special cases that guarantee converge to a fixpoint.

- Some simple linear conditions on the weights, such as zero-diagonal symmetry ($w_{ij} = w_{ji}$, $w_{ii} = 0$) guarantee that the Lyopunov function

$$L = -\sum_{i,j} w_{ij} y_i y_j + \sum_i (y_i \log y_i + (1 - y_i) \log(1 - y_i)) \tag{4}$$

  decreases until a fixpoint is reached [7]. The weight symmetry condition arises naturally if weights are considered to be Bayesian constraints, as in Boltzmann Machines [17].

- Atiya [4] showed that a unique fixpoint is reached regardless of initial conditions if $\sum_{ij} w_{ij}^2 < \max(\sigma')$, but in practice much weaker bounds on the weights seem to suffice, as indicated by empirical studies of the dynamics of networks with random weights [41].

- Other empirical studies indicate that applying fixpoint learning algorithms stabilizes networks, causing them to exhibit asymptotic fixpoint behavior [2, 12]. There is as yet no theoretical explanation for this phenomenon.

[1] Technically, these algorithms only require that a fixpoint be reached, not that it be stable. However, it is unlikely (with probability zero) that a network will converge to an *unstable* fixpoint, and in practice the posibility of convergence to unstable fixpoints can be ignored.

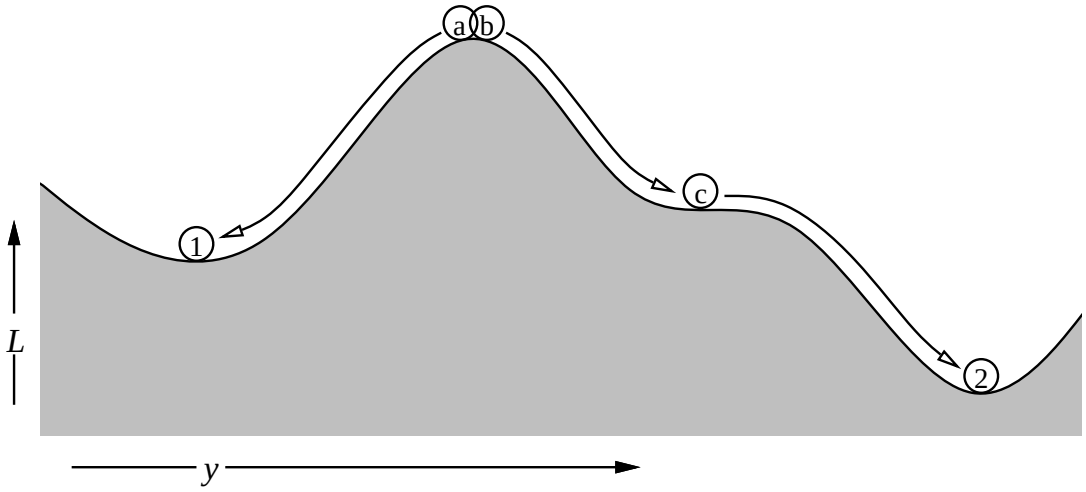

Figure 1: This energy landscape, represented by the curved surface, and the balls, representing states of the network, illustrates some potential problems with fixpoints. The initial conditions **a** and **b** can differ infinitesimally but map to different fixpoints, so the mapping of initial conditions to fixpoints is not continuous. Likewise, an infinitesimal change to the weights can change which fixpoint the system evolves to from a given starting point by moving the boundary between the watersheds of two attractors. Similarly, point **c** can be changed from a fixpoint to a non-fixpoint by an infinitesimal change to the weights.

One algorithm that is capable of learning fixpoints, but that does not require the network being trained to settle to a fixpoint in order to operate, is backpropagation through time. This has been used by Nowlan to train a constraint satisfaction network for the eight queens problem, where shaping was used to gradually train a discrete time network without hidden units to exhibit the desired attractors [32].

However, the other fixpoint algorithms we will consider take advantage of the special properties of a fixpoint to simplify the learning algorithm.

2.2 Problems with Fixpoints

Even when it can be guaranteed that a network settles to a fixpoint, fixpoint learning algorithms can still run into trouble. The learning procedures discussed here all compute the derivative of some error measure with respect to the internal parameters of the network. This gradient is then used by an optimization procedure, typically some variant of gradient descent, to minimize the error. Such optimization procedures assume that the mapping from the network's internal parameters to the consequent error is continuous, and can fail spectacularly when this assumption is violated.

Consider mapping the initial conditions $\tilde{y}(t_0)$ to the resultant fixpoints, $\tilde{y}(t_\infty) = \mathcal{F}(\tilde{y}(t_0))$. Although the dynamics of the network are all continuous, $\mathcal{F}$ need not be. For purposes of

visualization, consider a symmetric network, whose dynamics thus cause the state of the network to descend the energy function of equation (4). As shown schematically in figure 1, even an infinitesimal change to the initial conditions, or to the location of a ridge, or to the slope of an intermediate point along the trajectory, can change which fixpoint the system ends up in. In other words, $\mathcal{F}$ is not continuous. This means that as a learning algorithm changes the locations of the fixpoints by changing the weights, it is possible for it to cross such a discontinuity, making the error jump suddenly; and this remains true no matter how gradually the weights are changed.

2.3 Recurrent Backpropagation

Pineda [39] and Alemeida [3] discovered that the error backpropagation algorithm [34, 44, 49] is a special case of a more general error gradient computation procedure. The backpropagation equations are

$$y_i = \sigma(x_i) + I_i \tag{5}$$

$$z_i = \sigma'(x_i) \sum_j w_{ij} z_j + e_i \tag{6}$$

$$\frac{\partial E}{\partial w_{ij}} = y_i z_j \tag{7}$$

where z_i is the ordered partial derivative of E with respect to y_i, E is an error metric over $y(t_\infty)$, and $e_i = \partial E / \partial y_i(t_\infty)$ is the simple derivative of E with respect to the final state of a unit. In the original derivations of backpropagation, the weight matrix is assumed to be triangular with zero diagonal elements, which is another way of saying that the connections are acyclic. This ensures that a fixpoint is reached, and allows it to be computed very efficiently in a single pass through the units. But the backpropagation equations remain valid even with recurrent connections, assuming a fixpoint is reached.

If we assume that equation (1) reaches a fixpoint, $y(t_\infty)$, then equation (5) must be satisfied. And if (5) is satisfied, then if we can find z_i that satisfy (6), then (7) will give us the derivatives we seek, even in the presence of recurrent connections. (For a simple task, it has been reported [33] that reaching the precise fixpoint is not crucial to learning.)

One way to compute a fixpoint for (5) is to relax to a solution. By subtracting y_i from each side, we get

$$0 = -y_i + \sigma(x_i) + I_i$$

and at a fixpoint $dy_i/dt = 0$ so the equation

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

has the appropriate fixpoints. Now we note that if $-y_i + \sigma(x_i) + I_i$ is greater than zero than we could reduce its value by increasing y_i, so under these circumstances dy_i/dt should be positive, so k should be greater than zero. We can choose $k = 1$, giving (1) as a technique for relaxing to a fixpoint of (5).

Equation (6) is linear once y is determined, so its solution is unique. Any technique for solving a set of linear equations could be used. Since we are computing a fixpoint of (5) using the associated differential equation (1), it is tempting to do the same for (6) using

$$\frac{dz_i}{dt} = -z_i + \sigma'(x_i) \sum_j w_{ij} z_j + e_i. \tag{8}$$

These equations admit to direct analog implementation. In a real analog implementation, different time constants would probably be used for (1) and (8), and under the assumption that the time y and z spend settling is negligible compared to the time they spend at their fixpoints and that the rate of weight change η is slow compared to the speed of presentation of new training samples, the weights would likely be updated continuously by an equation like

$$\frac{dw_{ij}}{dt} = -\eta \frac{dE}{dw_{ij}} = -\eta y_i z_j \tag{9}$$

or, if a momentum term $0 < \alpha < 1$ is desired,

$$\frac{d^2 w_{ij}}{dt^2} + (1 - \alpha) \frac{dw_{ij}}{dt} + \eta y_i z_j = 0. \tag{10}$$

2.3.1 Simulation of an Associative Network

We simulated a recurrent backpropagation network learning a higher order associative task, that of associating three pieces of information: two four bit shift registers, A and B, and a direction bit, D. If D is off, then B is equal to A. If D is on, then B is equal to A rotated one bit to the right. The task is to reconstruct one of these three pieces of information, given the other two.

The architecture of the network is shown in figure 2. Three groups of visible units hold A, B, and D. An undifferentiated group of ten hidden units is fully and bidirectionally connected to all the visible units. There are no connections between visible units. An extra unit, called a bias unit, is used to implement thresholds. This unit has no incoming connections, and is forced to always have a value of 1 by a constant external input of 0.5. Connections go from it to each other unit, allowing units to have biases, which are equivalent to the negative of the threshold, without complicating the mathematics. Inputs are represented by an external input of +0.5 for an on bit, —0.5 for an off bit, and 0 for a bit to be completed by the network.

The network was trained by giving it external inputs that put randomly chosen patterns on two of the three visible groups, and training the third group to attain the correct value. The error metric was the squared deviation of each I/O unit from its desired state, except that units were not penalized for being "too correct."[2] All 96 patterns were successfully learned, except for the ones which were ambiguous, as shown in the state diagrams of figure 4. The weights after this training, which took about 300 epochs, are shown in figure 3. By inspection, many weights are large and decidedly asymmetric; but during training, no

[2] A unit with external input could be pushed beyond the [0,1] bounds.

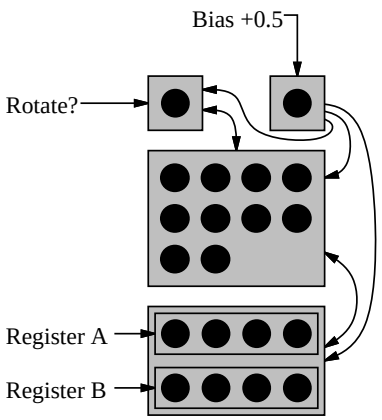

Figure 2: The architecture of a network to solve an associative version of the four bit rotation problem.

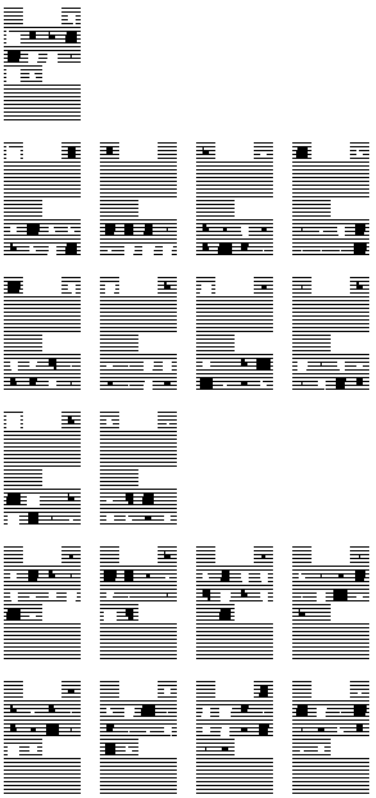

Figure 3: A Hinton diagram of weights learned by the network of figure 2.

instabilities were observed. The network consistently settled to a fixpoint within twenty simulated time units. When the network was tested on untrained completion problems, such as reconstructing D as well as half of A and B from partially, but unambiguously, specified A and B, performance was poor. However, redoing the training with weight symmetry enforced caused the network to learn not only the training data but also to do well on these untrained completions.

Pineda and Alemeida's recurrent backpropagation learning procedure has also been successfully applied to learning weights for a relaxation procedure for dense stereo disparity problems with transparent surfaces by Qian and Sejnowski [40]. By training on examples, they were able to learn appropriate weights instead of deriving them from simplified and unrealistic analytical model of the distribution of surfaces to be encountered, as is usual.

2.4 Deterministic Boltzmann Machines

The mean field form of the stochastic Boltzmann Machine learning rule [38] has recently been shown to descend an error functional [16]. Stochastic Boltzmann Machines themselves [1] are beyond the scope of this document; here we give only the probabilistic interpretation of MFT Boltzmann Machines, without derivation.

Figure 4: Network state for all the cases in the four bit rotation problem. This display shows the states of the units, arranged as in figure 2. Each row of six shows one value for register A. There are $2^4 = 16$ such rows. Within each row, the three diagrams on the left show the network's state when competing the direction bit, register B, and register A, unshifted. The right three are the same, except with a shift. Note that all completions are correct except in the two cases where the rotation bit can not be determined from the two shift registers, namely a pattern of 0000 or 1111.

In a a deterministic Boltzmann Machine, the transfer function of (1) is $\sigma(\xi) = (1 + e^{-\xi/T})^{-1}$, where T is the *temperature,* which starts at a high value and is gradually lowered to a *target temperature* each time the network is presented with a new input; without loss of generality, we assume this target temperature to be $T = 1$. The weights are assumed to be symmetric and zero-diagonal. Input is handled in a different way than in the other procedures we discuss: the external inputs I_i are set to zero, and a subset of the units, rather than obeying (1), have their values set externally. Such units are said to be *clamped.*

In learning, a set of *input units* (states over which we will index with α) are clamped to some values, the network is allowed to settle, and the quantities

$$p_{ij}^{-} = <y_i y_j> = \sum_{\alpha} P(\alpha) y_i^{(\alpha)} y_j^{(\alpha)} \tag{11}$$

are accumulated, where $<\cdot>$ denotes an average over the environmental distribution and superscripts denote clamping. The same procedure is then repeated, but with the output units (states of which we will index by β) clamped to their desired values too, yielding

$$p_{ij}^{+} = <y_i y_j> = \sum_{\alpha,\beta} P(\alpha) y_i^{(\alpha,\beta)} y_j^{(\alpha,\beta)}. \tag{12}$$

At this point, it is the case that

$$\frac{\partial G}{\partial w_{ij}} = p_{ij}^{+} - p_{ij}^{-} \tag{13}$$

where

$$G = \sum_{\alpha,\beta} P(\alpha) \log \frac{P(\beta|\alpha)}{P^{-}(\beta|\alpha)} \tag{14}$$

is a measure of the information theoretic difference between the clamped and unclamped distribution of the output units given the clamped input units. $P^{-}(\beta|\alpha)$ measures how probable the network says β is given α, and its definition is beyond the scope of this document.

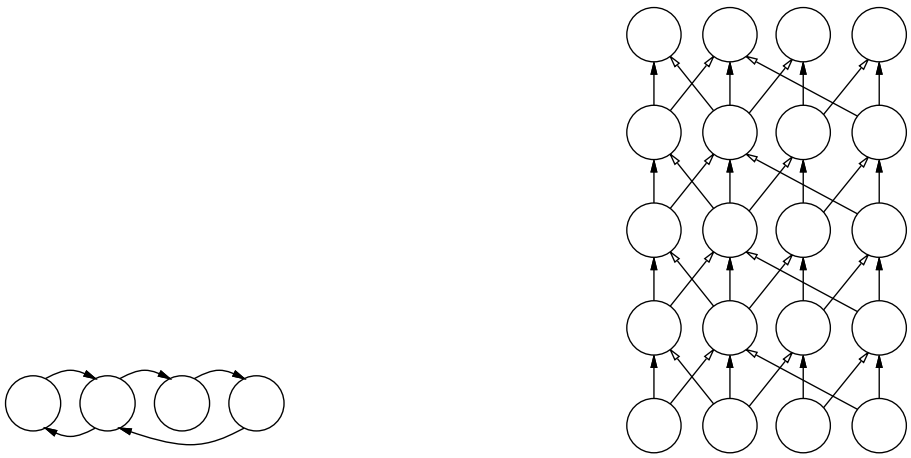

Figure 5: A recurrent network is shown on the left, and a representation of that network unfolded in time through four time steps is shown on the right.

This learning rule (13) is a version of Hebb's rule in which the sign of synaptic modification is alternated, positive during the "waking" phase and negative during the "hallucinating" phase.

Even before the learning rule was rigorously justified, deterministic Boltzmann Machines were applied with success to a number of tasks [37, 38]. Although weight symmetry is assumed in the definition of energy which is used in the definition of probability, and is thus fundamental to these mathematics, it seems that in practice weight asymmetry can be tolerated in large networks [12]. This makes MFT Boltzmann Machines the most biologically plausible of the various learning procedures we discuss, but it is difficult to see how it would be possible to extend them to learning more complex phenomena, like limit cycles or paths through state space. And thus, although they are probably the best technique in their domain of application, we now turn our attention to procedures suitable for learning more dynamic sorts of behaviors.

3 Backpropagation Through Time

The fixpoint learning procedures discussed above are unable to learn non-fixpoint attractors, or to produce desired temporal behavior over a bounded interval, or even to learn to reach their fixpoints quickly. Here, we turn to learning procedures suitable for such non-fixpoint situations.

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 derive a technique for computing $\partial E(\mathbf{y})/\partial w_{ij}$ efficiently, thus allowing us to do gradient descent in the weights so as to minimize E. Backpropagation through time has been used to train discrete time networks to perform a variety of tasks [44, 32]. Here, we will derive the continuous time version of backpropagation through time, as in [36], and use it in a couple toy domains.

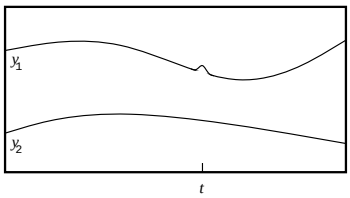

Figure 6: The infinitesimal changes to $\mathbf{y}$ considered in $e_1(t)$.

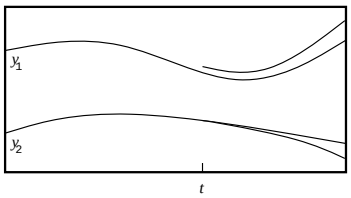

Figure 7: The infinitesimal changes to $\mathbf{y}$ considered in $z_1(t)$.

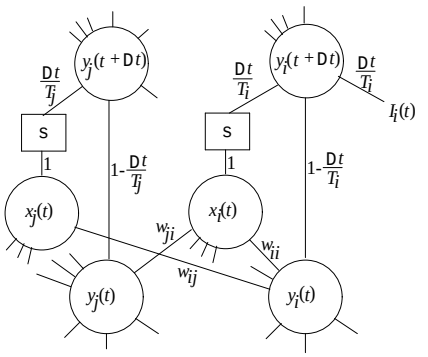

Figure 8: A lattice representation of (16).

In this derivation, we take the conceptually simple approach of unfolding the continuous time network into a discrete time network with a step of Δt, applying backpropagation to this discrete time network, and taking the limit as Δt approaches zero to get a continuous time learning rule.

The derivative in (1) can be approximated with

$$\frac{dy_i}{dt}(t) \approx \frac{y_i(t+\Delta t) - y_i(t)}{\Delta t}, \tag{15}$$

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

$$\tilde{y}_i(t+\Delta t) = (1-\Delta t)\tilde{y}_i(t) + \Delta t \sigma(\tilde{x}_i(t)) + \Delta t I_i(t). \tag{16}$$

Tildes are used throughout for temporally discretized versions of continuous functions.

Let us define

$$e_i(t) = \frac{\delta E}{\delta y_i(t)}. \tag{17}$$

In the usual case E is of the form

$$E = \int_{t_0}^{t_1} f(\mathbf{y}(t), t)\, dt \tag{18}$$

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.

As usual in backpropagation, let us define

$$\tilde{z}_i(t) = \frac{\partial^+ E}{\partial \tilde{y}_i(t)} \tag{19}$$

where the ∂^+ denotes an ordered derivative [50], with variables ordered here by time and not unit index. 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 7. Of course, z_i is the limit of $\tilde{z}_i$ as $\Delta t \to 0$. This z is the δ of the standard backpropagation "generalized δ rule."

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 $d\tilde{y}_i(t)$ has an effect on $\tilde{y}_i(t + \Delta t)$ of $d\tilde{y}_i(t)\ (1 - \Delta t)$, giving us a second term, namely $(1 - \Delta t)\tilde{z}(t + \Delta t)$.

Each weight w_{ij} makes $\tilde{y}_i(t)$ influence $\tilde{y}_j(t + \Delta t)$, $i \neq j$. Let us compute this influence in stages. Varying $\tilde{y}_i(t)$ by $d\tilde{y}_i(t)$ varies $\tilde{x}_j(t)$ by $d\tilde{y}_i(t)\ w_{ij}$, which varies $\sigma(\tilde{x}_j(t))$ by $d\tilde{y}_i(t)\ w_{ij}\ \sigma'(\tilde{x}_j(t))$, which varies $\tilde{y}_j(t + \Delta t)$ by $d\tilde{y}_i(t)\ w_{ij}\ \sigma'(\tilde{x}_j(t))\ \Delta t$. This gives us our third and final term, $\sum_j w_{ij}\ \sigma'(\tilde{x}_j(t))\ \Delta t\ \tilde{z}_j(t + \Delta t)$.

Combining these,

$$\tilde{z}_i(t) = \Delta t\, e_i(t) + (1 - \Delta t)\tilde{z}_i(t + \Delta t) + \sum_j w_{ij}\sigma'(\tilde{x}_j(t))\Delta t\tilde{z}_j(t + \Delta t). \tag{20}$$

If we put this in the form of (15) and take the limit as $\Delta t \to 0$ we obtain the differential equation

$$\frac{dz_i}{dt} = z_i - e_i - \sum_j w_{ij}\sigma'(x_j)z_j. \tag{21}$$

For boundary conditions note that by (17) and (19) $\tilde{z}_i(t_1) = \Delta t e_i(t_1)$, 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 $y_i(t)\sigma'(x_j(t))\Delta t z_j(t) dw_{ij}$. Since we wish to know the effect of making this infinitesimal change to w_{ij} throughout time, we integrate over the entire interval, yielding

$$\frac{\partial E}{\partial w_{ij}} = \int_{t_0}^{t_1} y_i\sigma'(x_j)z_j dt. \tag{22}$$

One can also derive (21), (22) and (26) using the calculus of variations and Lagrange multipliers (William Skaggs, personal communication), as in optimal control theory [22]. In fact, the idea of using gradient descent to optimize complex systems was explored by control theorists in the late 1950s. Although their mathematical techniques handled hidden units, they refrained from exploring systems with so many degrees of freedom, perhaps in fear of local minima.

It is also interesting to note the recurrent backpropagation learning rule (section 2.3) can be derived from these. Let I_i be held constant, assume that the network settles to a fixpoint, and let E be integrated for one time unit before t_1. As $t_1 \to \infty$, (21) and (22) reduce to the recurrent backpropagation equations (8) and (7), so in this sense backpropagation through time is a generalization of recurrent backpropagation.

3.1 Time Constants

If we add a time constant T_i to each unit i, modifying (1) to

$$T_i\frac{dy_i}{dt} = -y_i + \sigma(x_i) + I_i, \tag{23}$$

and carry these terms through the derivation of the last section, equations (21) and (22) become

$$\frac{dz_i}{dt} = \frac{1}{T_i} z_i - e_i - \sum_j \frac{1}{T_j} w_{ij} \sigma'(x_j) z_j. \tag{24}$$

and

$$\frac{\partial E}{\partial w_{ij}} = \frac{1}{T_j} \int_{t_0}^{t_1} y_i \sigma'(x_j) z_j dt. \tag{25}$$

In order to learn these time constants rather than just set them by hand, we need to compute $\partial E(\mathbf{y})/\partial T_i$. If we substitute $\rho_i = T_i^{-1}$ into (23), find $\partial E/\partial \rho_i$ with a derivation similar to that of (22), and substitute T_i back in we get

$$\frac{\partial E}{\partial T_i} = -\frac{1}{T_i} \int_{t_0}^{t_1} z_i \frac{dy_i}{dt} dt. \tag{26}$$

3.2 Time Delays

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

$$x_i(t) = \sum_j w_{ji} y_j(t - \tau_{ji}), \tag{27}$$

τ_{ji} being the time delay along the connection from unit j to unit i. Let us include the variable time constants of section 3.1 as well. Surprisingly, such time delays merely add analogous time delays to (24) and (25),

$$\frac{dz_i}{dt}(t) = \frac{1}{T_i} z_i(t) - e_i(t) - \sum_j w_{ij} \sigma'(x_j(t + \tau_{ij})) \frac{1}{T_j} z_j(t + \tau_{ij}), \tag{28}$$

$$\frac{\partial E}{\partial w_{ij}} = \frac{1}{T_j} \int_{t_0}^{t_1} y_i(t) \sigma'(x_j(t + \tau_{ij})) z_j(t + \tau_{ij}) dt, \tag{29}$$

while (26) remains unchanged. If we set $\tau_{ij} = \Delta t$, these modified equations alleviate concern over time skew when simulating networks of this sort, obviating any need for accurate numerical simulations of the differential equations and allowing simple difference equations to be used without fear of inaccurate error derivatives.

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

$$\frac{\partial E}{\partial \tau_{ij}} = \int_{t_0}^{t_1} z_j(t) \sigma'(x_j(t)) w_{ij} \frac{dy_i}{dt}(t - \tau_{ij}) dt. \tag{30}$$

We have not yet simulated networks with modifiable time delays, although there is work in progress at another institution to do so.

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 can certainly be accommodated by this 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 [25, 26, 48]. 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 specify 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 (27) as

$$x_i(t) = \sum_{j \,|\mathcal{L}(j)=i} w_j y_{\mathcal{R}(j)}(t - \tau_j). \tag{31}$$

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

3.3 Some Simulations

In the following simulations, we used networks without time delays, but with mutable time constants. As in the associative network of section 2.3.1, an extra input unit whose value was always held at 1 by a constant external input of 0.5, and which had outgoing connections to all other units, was used to implement biases.

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 $\partial E/\partial T_i$. Since computing dz_i/dt requires $\sigma'(x_i)$, 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

$$E = \frac{1}{2}\sum_i \int_{t_0}^{t_1} s_i(\,y_i - d_i)^2 dt \tag{32}$$

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 first order difference equations (16) and (20) with $\Delta t = 0.1$.

3.3.1 Exclusive Or

The network of figure 9 was trained to solve the xor problem. Aside from the addition of time constants, the network topology was that used by Pineda in [39]. 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

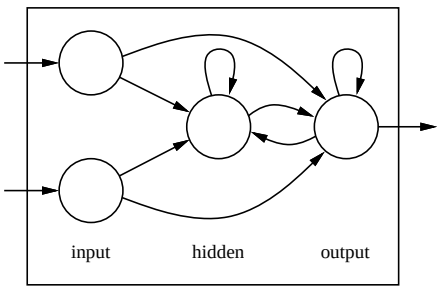

Figure 9: The XOR network.

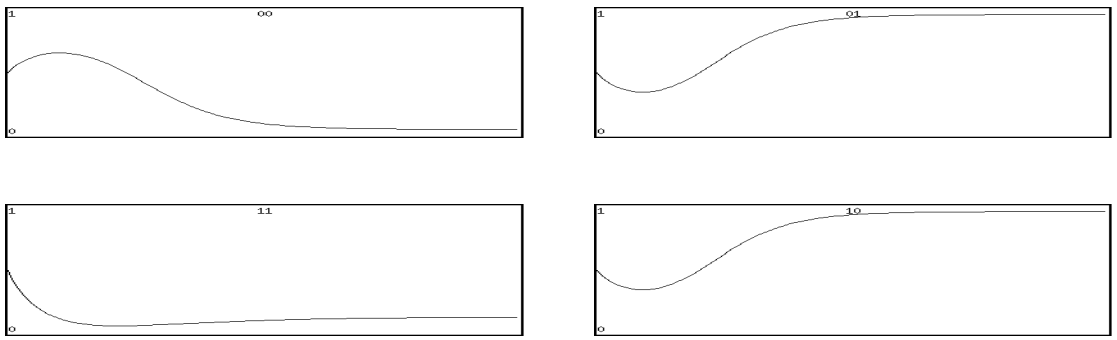

Figure 10: The states of the output unit in the four input cases plotted from $t = 0$ to $t = 5$ after 200 epochs of learning. The error was computed only between $t = 2$ and $t = 3$.

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. 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. An easier, and perhaps more justifiable, approach is to simply introduce a minimum time constant; this was done in later simulations.

What is interesting is that that even for this binary task, the network made use of dynamic 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 10. 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

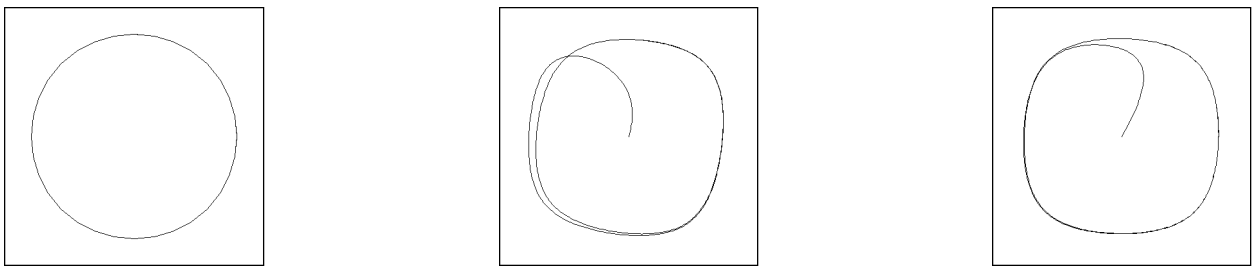

Figure 11: Desired states d_1 and d_2 plotted against each other (left); actual states y_1 and y_2 plotted against each other at epoch 1500 (center) and 12000 (right).

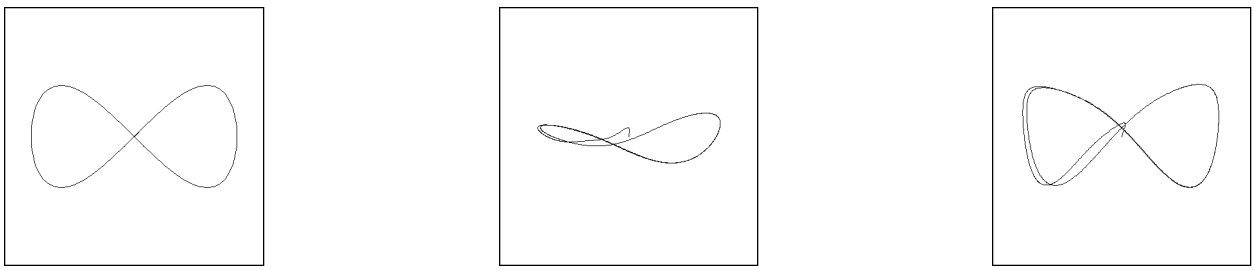

Figure 12: Desired states d_1 and d_2 plotted against each other (left); actual states y_1 and y_2 plotted against each other at epoch 3182 (center) and 20000 (right).

correct value and then moving away is seen in the lower left hand corner of figure 10.

3.3.2 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 11. 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.

3.3.3 A Figure Eight

We were unable to train a network with four hidden units to follow the figure eight shape shown in figure 12, 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

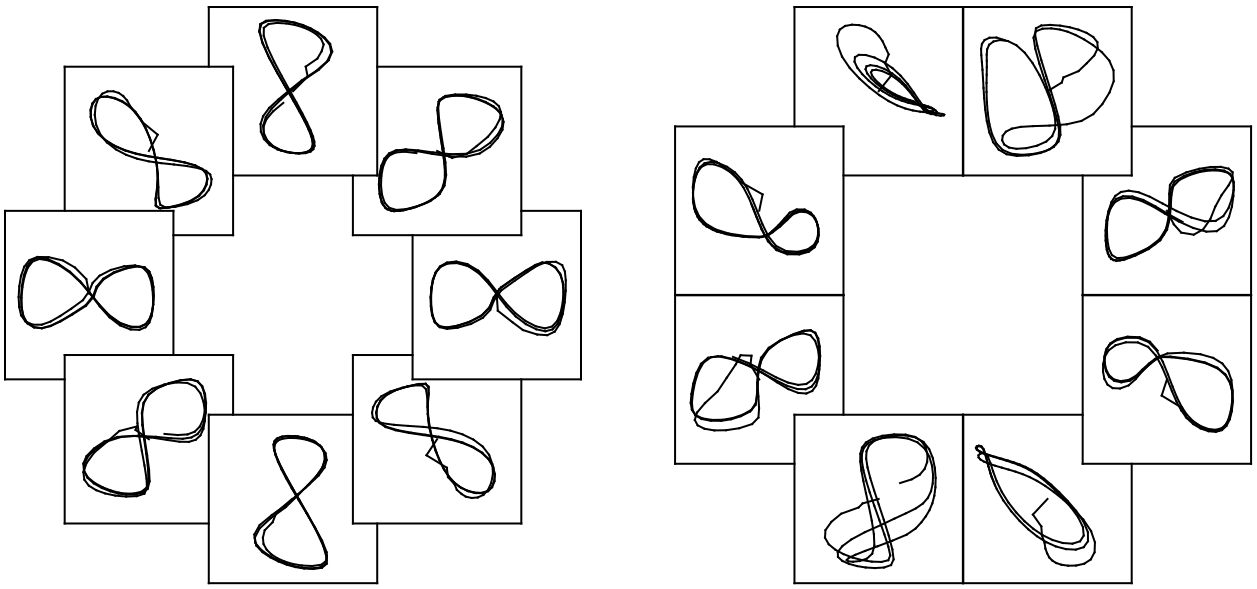

Figure 13: The output of the rotated eight network at all the trained angles (left) and some untrained angles (right).

difficult than for the circular trajectory, and shaping the network's behavior by gradually extending the length of time of the simulation proved useful.

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.

3.3.4 A Rotated Figure Eight

In this simulation a network was trained to generate a figure eight on its output units in precisely the same way as in the last section, except that the figure eight was to be rotated about its center by an angle θ which was input to the network through two input units which held the coordinates of a unit vector in the appropriate direction. Eight different values of θ, equally spaced about the circle, were used to generate the training data. In experiments with 20 hidden units, the network was unable to learn the task. Increasing the number of hidden units to 30 allowed the network to learn the task, as shown on the left in figure 13. But when the network is run with the eight input angles furthest the training angles, as shown on the right in figure 13, generalization is poor.

The task would be simple to solve using second order connections, as they would allow the problem to be decoupled. A few units could be devoted to each of the orthogonal oscillations, and the connections could form a rotation matrix. The poor generalization of the network shows that it is not solving the problem in such a straightforward fashion, and suggests that for tasks of this sort it might be better to use slightly higher order units.

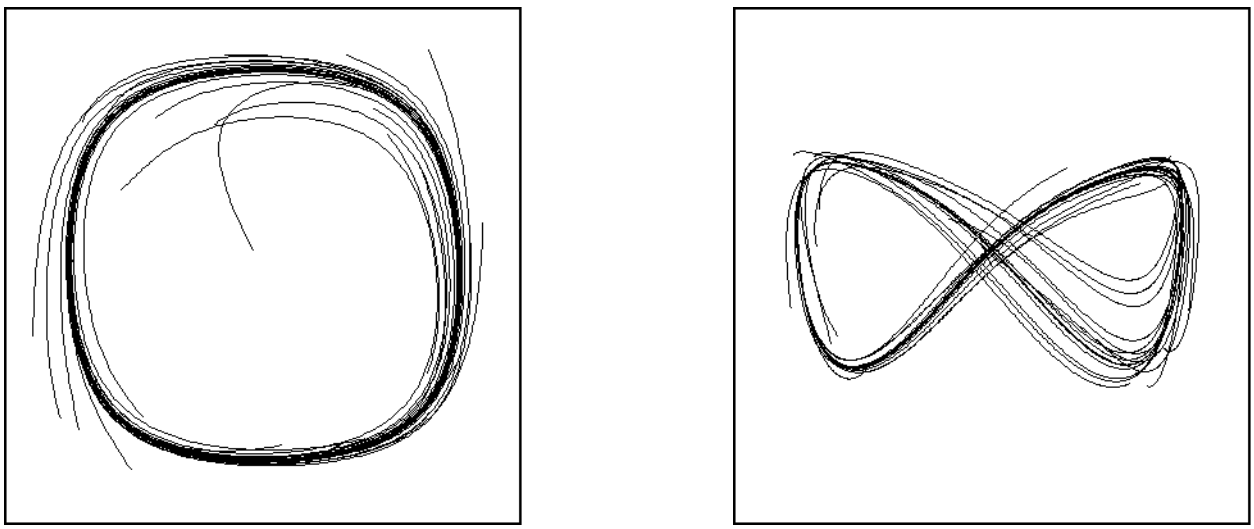

Figure 14: The output states y_1 and y_2 plotted against each other for a 1000 time unit run, with all the units in the network perturbed by a random amount about every 40 units of time. The perturbations in the circle network (left) were uniform in ± 0.1, and in the eight network (right) in ± 0.05.

3.4 Stability and Perturbation Experiments

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.

In an attempt to judge the stability of the limit cycles exhibited above, rather than going to the trouble of calculating Df, where $f(y(t)) = y(t + 16)$, we simply modified the simulator to introduce random perturbations and observed the effects of these perturbations upon the evolution of the system.[3] The two output units in the unrotated 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 create by analytically determining weights through decoupling the two output units and using linearized subnets to generate the desired oscillatory behavior.

The limit cycle on the right in figure 12 is symmetric, but when perturbations are introduced, as in the right of figure 14, 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

[3]Actually, we wouldn't care about the eigenvalues of Df per se, because we wouldn't care about perturbations in the direction of travel, as these effect only the phase. For this reason, we would want to project this out of the matrix before computing the eigenvalues. This effect is achieved automatically in our display in figure 14.

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 towards 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.

3.5 Leech Simulations

Lockery et al. used the techniques discussed above to fit a low level neurophysiological model of the leech local bending reflex to data on sensory and motor neuron activity [28, 29, 30, 31]. They modified the dynamic equations substantially in order to model their system at a low level, using activity levels to represent currents rather than voltages. Their trained model disagreed with human intuition concerning what the synaptic strengths, and in fact signs, would be, but qualitatively matched empirical measurements of these counterintuitive synaptic strengths.

4 Other Non-fixpoint Techniques

4.1 "Elman Nets"

Elman [9] investigated a version of discrete backpropagation through time in which the temporal history is cut off. Typically, only one or two timesteps are preserved, at the discretion of the architect. This cutoff makes backpropagation through time an online algorithm, as the backpropagation to be done to account for the error at each point in time is done immediately. However, it makes the computational expense per time step scale linearly with the number of timesteps of history being maintained. This accuracy of the computed derivative is smoothly traded off against storage and computation.

The real question with Elman networks is whether the contribution to the error from the history that has been cut off is significant. This question can only be answered relative to a particular task. For instance, Elman [9] finds some problems amenable to the history cutoff, but resorts to full fledged backpropagation through time for other tasks. Cleeremans et al. [6] find a regular language token prediction task which is difficult for Elman nets when the transition probabilities are equal, but find that breaking this symmetry allows these nets to learn the task.

4.2 The Moving Targets Method

Rohwer, among others, has proposed a moving targets learning algorithm [43]. In such algorithms, two phases alternate. In one phase, the hidden units' targets are improved, such that if the targets are attained better performance would be achieved. In the other

phase, the weights are modified such that each unit comes closer to attaining its target. The error can be regarded as having two terms, one term which penalizes the units being too far from their targets, and another which penalizes the targets for being too far from the values actually attained. This technique has the appeal of decoupling temporally distant actions during the learning of weights, and the disadvantage of requiring the targets to be stored and updated. In the limit, as learning rates are decreased, the moving targets method becomes equivalent to backpropagation though time.

The primary disadvantage of the technique is that each pattern to be learned must have associated with it the targets for the hidden units, and these targets must be learned just as the weights are. This makes the technique inapplicable for online learning, in which each pattern is seen only once.

4.3 Forward Propagation

An online, exact, and stable, but computationally expensive, procedure for training fully recurrent networks was discovered by Robinson and Fallside [42], and later rediscovered independently by others [13, 52]. We can develop this technique as follows. If E is of the form of (18), we can calculate $\partial E/\partial w_{ij}$ as follows. First apply the chain rule to E,

$$\frac{\partial E}{\partial w_{ij}} = \int \sum_k \frac{\partial f}{\partial y_k}(t)\frac{\partial y_k}{\partial w_{ij}}(t)dt = \int \sum_k e_k(t)\gamma_{ijk}(t)dt \tag{33}$$

where

$$\gamma_{ijk} = \partial y_k/\partial w_{ij}. \tag{34}$$

These can be calculated by taking the derivative of (1) with respect to w_{ij}, yielding the auxiliary equations

$$\frac{d\gamma_{ijk}}{dt} = \frac{\partial f_k}{\partial y_k}\gamma_{ijk} + ([j=k]y_j + \sum_l w_{lk}\gamma_{ijl})\frac{\partial f_k}{\partial net_k}. \tag{35}$$

Rather than integrating (33) during the simulation and then making a weight change, we can continuously update the weights according to the equation

$$\frac{dw_{ij}}{dt}(t) = -\eta \sum_k \frac{\partial g}{\partial y_k}(t)\gamma_{ijk}(t). \tag{36}$$

By integrating this expression between t_0 and t_1 under the assumption that the online weight changes do not affect the trajectory taken, we can see that it is equivalent to the discrete update equation it replaces.

Since the auxiliary quantities γ_{ijk} have only initial boundary conditions–zero at the start of time–all the computations can be carried out forward in time. Because of this, the technique is an *online* learning procedure, as the amount of time that the network will be run need not be known in advance, and no history need be stored. In addition, (35) is stable if (1) is, so the technique will not introduce numerical instabilities.

Regretably, this technique poses a substantial computational burden, with the computation of the auxiliary equations dominating. If we have n units and n^2 weights, then there are n primary state variables and only $O(n^2)$ calculations are required to update them. But there are n^3 auxiliary variables, and they require a total of $O(n^4)$ calculations to update! Furthermore, although the primary equations could potentially be implemented directly in analog hardware, the auxiliary equations use each weight n^2 times, making analog hardware implementation difficult.

4.3.1 Extending Online Learning to Time Constants and Delays

We can easily extend this online learning procedure to take account of time constants. If we begin with (23), substitute k for i, take the partial with respect to w_{ij}, and substitute in γ where possible, we have a the differential equation for γ

$$T_k \frac{\gamma kij}{dt} = -\gamma kij + \sigma'(x_k) \sum_l w_{lk} \gamma lij, \tag{37}$$

nearly the same as (35) except for a time constant.

We can derive analogous equations for the time constants themselves; define

$$q_j^i(t) = \frac{\partial y_i(t)}{\partial T_j}, \tag{38}$$

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

$$T_i \frac{dq_j^i}{dt} = -q_j^i - \frac{dy_i}{dt} + \sigma'(x_i) \sum_k w_{ki} q_j^k \tag{39}$$

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

$$\frac{d\,T_i}{dt} = -\eta \sum_j e_j q_i^j. \tag{40}$$

Similarly, let us derive equations for modifying the time delays of section 3.2. Define

$$r_{ij}^k(t) = \frac{\partial y_k(t)}{\partial \tau_{ij}} \tag{41}$$

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

$$T_k \frac{dr_{ij}^k}{dt} = -r_{ij}^k + \sigma'(x_k)(\underbrace{w_{ij} \frac{dy_i}{dt}(t - \tau_{ij})}_{\text{included if } j = k} - \sum_l w_{lk} r_{ij}^l (t - \tau_{lk})). \tag{42}$$

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

$$\frac{d\tau_{ij}}{dt} = -\eta \sum_k e_k r_{ij}^k. \tag{43}$$

4.3.2 Faster Online Techniques

One way to reduce the complexity of the algorithm is to simply leave out auxiliary variables that one has reason to believe will remain approximately zero, simply discarding the corresponding terms. This approach, in particular ignoring the coupling terms which relate the states of units in one module to weights in another, has been explored by Zipser [53].

Recently Toomarian and Barhen [46] used clever transformations of the adjoint equations to derive an exact, stable variant of this forward propagation algorithm which requires only $2n + n^2$ auxiliary variables, which can be updated in just $O(n^3)$ time. Their technique was announced just before this document went to press, and has not yet been simulated, but appears quite promising. If verified in practice, their technique would appear to dominate the online algorithm described above, and would become the technique of choice for online learning.

4.4 Feedforward Networks with State

It is noteworthy that that the same basic mathematical technique of forward propagation can be applied to networks with a restricted architecture, feedforward networks whose units have state [14, 24, 47]. This is the same as requiring the w_{ij} matrix to be triangular, but allowing non-zero diagonal terms. If we let the γ quantities be ordered derivatives, as in standard backpropagation, than this simplified architecture reduces the computational burden substantially. The elimination of almost all temporal interaction makes $\gamma_{ijk} = 0$ unless $i = k$, leaving only $O(n^2)$ auxiliary equations, each of which can be updated with $O(1)$ computation, for a total update burden of $O(n^2)$, which is the same as conventional backpropagation. This favorable computational complexity makes it of practical significance even for large feedforward recurrent networks. But these feedforward networks are outside the scope of this paper.

5 Teacher Forcing

Teacher forcing [52] consists of jamming the desired output values into units as the network runs; thus, the teacher forces the output units to have the correct states, even as the network runs, and hence the name. This technique is applied to discrete time, clocked networks, as only then does the concept of changing the state of an output unit each time step make sense.

The error is as usual, with the caveat that errors are to be measured before output units are forced, not after. Williams and Zipser report that their teacher forcing technique radically reduced training time for their recurrent networks, although others using teacher forcing on networks with a larger number of hidden units reported difficulties[35].

5.1 In Continuous Time

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 approach we shall take is to add some knobs that can be used to control the states of the output units, and use them to keep the output units locked at their desired states. The error function to be minimized will measure the amount of control that it was necessary to exert, with zero error coming only when the knobs need not be twisted at all.

Let

$$F_i = \frac{1}{T_i}(-y_i + \sigma(x_i) + I_i) \tag{44}$$

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

$$\frac{dy_i}{dt} = F_i + f_i. \tag{45}$$

Using Φ to denote the set of units to be forced, we will let d_i be the trajectory that we will force y_i to follow, for each $i \in \Phi$. So we set

$$f_i = \frac{dd_i}{dt} - F_i \tag{46}$$

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

$$E = \int_{t_0}^{t_1} L(f(t), t)dt, \tag{47}$$

where typically $L = \sum_{i \in \Phi} f_i^2$.

We can modify the derivation in section 3 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 (17) 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

$$e_i = \sum_{j \in \Phi} \frac{\partial L}{\partial f_j} - \frac{1}{T_j}\sigma'(x_j)w_{ij} \tag{48}$$

For $i \notin \Phi$ (21) and (26) are unchanged, and for $j \notin \Phi$ and any i (22) 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

$$\frac{\partial E}{\partial w_{ij}} = \frac{1}{T_j} \int_{t_0}^{t_1} y_i \sigma'(x_j) \frac{\partial L}{\partial f_i} dt. \tag{49}$$

Analogously, for $i \in \Phi$

$$\frac{\partial E}{\partial T_i} = -\frac{1}{T_i}\int_{t_0}^{t_1} \frac{\partial L}{\partial f_i}\frac{dy_i}{dt}dt. \tag{50}$$

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 (21), (22), and (26) 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.

5.2 "Jordan Nets"

Jordan [21] used a backpropagation network with the outputs clocked back to the inputs to generate temporal sequences. Although these networks were used long before teacher forcing, from our perspective Jordan nets are simply a restricted class of teacher forced recurrent networks, in particular, discrete time networks in which the only recurrent connections emanate from output units. By teacher forcing these output units, no real recurrent paths remain, so simple backpropagation through a single time step suffices for training.

The main disadvantage of such an architecture is that state to be retained by the network across time must to manifest in the desired outputs of the network, so new persistent internal representations of temporal structures can not be created. For instance, it would be impossible to train such networks to perform the figure eight task of section 3.3.3. In the usual control theory way, this difficulty can be partially alleviated by cycling back to the inputs not just the previous timestep's outputs, but also those from a small number of previous timesteps. The tradeoffs between using hidden units to encapsulate temporally hidden structure and using a temporal window of values which must contain the desired information is problem dependent, and depends in essence on how long a hidden variable can remain hidden without being manifested in the observable state variables.

5.3 Continuous Time "Jordan Nets"

It is easy to construct a continuous time Jordan network, in which the units' values are continuous in time, the output units constantly have corrected values jammed into them from external sources, and the only recurrent connections are from the outputs back to the inputs. Although this was done in the more general setting of fully recurrent networks, we note in passing that the most natural teacher forced continuous time Jordan network has no state held at individual units, and is equivalent to simply training a layered network to produce the first derivative of the "output signal" given the current value of the "output signal" as an input.

technique	*time*	*space*	*online?*	*stable?*	*local?*
storing y	$O(m)$	$O(sn+m)$	no	yes	yes
y backwards	$O(m)$	$O(m)$	no	no	yes
forward propagation 1	$O(n^2m)$	$O(nm)$	yes	yes	no
forward propagation 2	$O(nm)$	$O(n^2+m)$	yes	yes	no

Table 1: A summary of the complexity of some learning procedures for recurrent networks. In the "storing y" technique we store y as time is run forwards and replay it as we run time backwards computing z. In "y backwards" we do not store y, instead recomputing it as time is run backwards. "Forward propagation" 1 and 2 are the online techniques described in section 4.3. The times given are for computing the gradient with respect to one pattern.

6 Summary and Conclusion

6.1 Complexity Comparison

Consider a network with n units and m weights which is run for s time steps (variable grid methods [5] would reduce s by dynamically varying Δt) 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 $\mathbf{y}$ system takes $O(m+n) = O(m)$ time for each time step, as does simulating the $\mathbf{z}$ system. This means that using the technique described in section 3.3, 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 $\mathbf{y}$ during the forward simulation to replay while simulating $\mathbf{z}$ backwards takes $O(sn)$ space, so if we use this technique the entire computation takes $O(sn+m)$ space. If we simulate $\mathbf{y}$ backwards during the backwards simulation of $\mathbf{z}$, 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.

The online technique described in section 4.3 requires $O(n^2m)$ time each time step, and $O(nm)$ space. The other technique alluded to in that section requires less time and space, and retains all of its online advantages, so it would appear to dominate the original technique, assuming simulations bear out its stability.

These time complexity results are for sequential machines, and are summarized in table 1.

Note that in this section we are concerning ourselves with how much computation it takes to obtain the gradient information. This is generally just the inner loop of a more complex algorithm to adjust the weights, which uses the gradient information, such as a gradient descent algorithm, or gradient descent with momentum, or conjugate gradient, or whatever is used. Experience has shown that learning in these networks has tended to be "stiff" in the sense that the Hessian of the error with respect to the weights (the matrix of second derivatives) tends to have a wide eigenvalue spread. One technique that has apparently proven useful in this particular situation is that of Robert Jacobs [20] which was applied by Fang and Sejnowski to the problem described in section 3.3.3 with great success [10]. It was

also used in the leech simulations of Lockery et al. described in section 3.5, apparently with a substantial reduction in the number of epochs required for convergence.

6.2 Future Work

Applications to identification and control are being 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 might be naturally applied. Such domains may lead us to complex architectures like those discussed in section 3.2. 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. In fact, Simard, Rayzs and Victorri have developed a technique for learning the local maximum eigenvalue of the transfer function [45], optionally projecting out directions whose eigenvalues are not of interest. This technique has not yet been applied in a control domain.

On the other hand, we can turn the logic of section 3.4 around. Consider a difficult constraint satisfaction task of the sort that neural networks are sometimes applied to, such as the traveling salesman problem [19]. Two competing techniques for such problems are simulated annealing [23, 1] and mean field theory [37]. 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, and into the best ways to structure annealing schedules.

6.3 Conclusions

Recurrent networks are often avoided because of a fear of inordinate learning times and incomprehensible algorithms and mathematics. It should be clear from the above that such fears are unjustified. Certainly there is no reason to use a recurrent network when a layered architecture suffices; but on the other hand, if recurrence is needed, there are a plethora of learning algorithms available across the spectrum of quiescence vs. dynamics and across the spectrum of accuracy vs. complexity and across the spectrum of space vs. time and storage.

6.4 Acknowledgments

I would like to thank my advisor, David Touretzky.

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] Robert B. Allen and Joshua Alspector. Learning of stable states in stochastic asymmetric networks. Technical Report TM-ARH-015240, Bell Communications Research, Morristown, NJ, November 1989.

[3] L. B. Almeida. A learning rule for asynchronous perceptrons with feedback in a combinatorial environment. In *Proceedings, 1st First International Conference on Neural Networks*, volume 2, pages 609–618, San Diego, CA, June 1987. IEEE.

[4] Amir F. Atiya. Learning on a general network. In Dana Z. Anderson, editor, *Neural Information Processing Systems*, pages 22–30, New York, New York, 1987. American Institute of Physics.

[5] 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.

[6] Axel Cleeremans, David Servan-Schreiber, and James McClelland. Finite state automata and simple recurrent networks. *Neural Computation*, 1(3):372–381, 1989.

[7] M. A. Cohen and Steven Grossberg. Stability of global pattern formation and parallel memory storage by competitive neural networks. *IEEE Transactions on Systems, Man, and Cybernetics*, 13:815–826, 1983.

[8] J. P. Crutchfield and B. S. McNamara. Equations of motion from a data series. *Complex Systems*, 1:417–452, 1987.

[9] Jeffrey L. Elman. Finding structure in time. Technical Report CRL-8801, Center for Research in Language, UCSD, April 1988.

[10] Yan Fang and Terrence J. Sejnowski. Faster learning for dynamic recurrent backpropagation. *Neural Computation*, 2(3):270–273, 1990.

[11] W. Freeman and S. Scarda. How brains make chaos in order to make sense of the world. *Brain and Behavioral Science*, November 87.

[12] Conrad C. Galland and Geoffrey E. Hinton. Deterministic boltzmann learning in networks with asymmetric connectivity. Technical Report CRG-TR-89-6, University of Toronto Department of Computer Science, 1989.

[13] Michael Gherrity. A learning algorithm for analog, fully recurrent neural networks. In *International Joint Conference on Neural Networks*, volume 2, pages 643–644. IEEE, 1989.

[14] Marco Gori, Yoshua Bengio, and Renato De Mori. Bps: A learning algorithm for capturing the dynamic nature of speech. In *International Joint Conference on Neural Networks*, volume 2, pages 417–423. IEEE, 1989.

[15] Geoffrey E. Hinton. Learning distributed representations of concepts. In *Proceedings of the Eighth Annual Cognitive Science Conference.* Lawrence Erlbaum, 1986.

[16] Geoffrey E. Hinton. Deterministic Boltzmann learning performs steepest descent in weight-space. *Neural Computation*, 1(1):143–150, 1989.

[17] Geoffrey E. Hinton and Terrence J. Sejnowski. Optimal perceptual inference. In *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*, pages 448–453, Washington, DC, June 1983. IEEE Computer Society.

[18] Geoffrey E. Hinton, Terrence J. Sejnowski, and David H. Ackley. Boltzmann Machines: Constraint satisfaction networks that learn. Technical Report CMU-CS-84-119, Carnegie-Mellon University, May 1984.

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

[20] Robert Jacobs. Increased rates of convergence through learning rate adaptation. Technical Report COINS 87-117, University of Massachusetts, Amherst, MA 01003, 1987.

[21] Michael I. Jordan. Attractor dynamics and parallelism in a connectionist sequential machine. In *Proceedings of the 1986 Cognitive Science Conference*, pages 531–546. Lawrence Erlbaum, 1986.

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

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

[24] Gary Kuhn. A first look at phonetic discrimination using connectionist models with recurrent links. SCIMP working paper 82018, Institute for Defense Analysis, Princeton, New Jersey, April 1987.

[25] 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.

[26] Kevin J. Lang, Geoffrey E. Hinton, and Alex Waibel. A time-delay neural network architecture for isolated word recognition. *Neural Networks*, 3(1):23–43, 1990.

[27] Alan Lapedes and Robert Farber. Nonlinear signal processing using neural networks: Prediction and system modelling. Technical report, Theoretical Division, Los Alamos National Laboratory, 1987.

[28] Shawn R. Lockery, Yan Fang, and Terrence J. Sejnowski. A dynamic neural network model of sensorimotor transformations in the leech. *Neural Computation*, 2(3):274–282, 1990.

[29] Shawn R. Lockery and W. B. Kristan Jr. Distributed processing of sensory information in the leech i: Input-output relations of the local bending relex. *Journal of Neuroscience*, 1990.

[30] Shawn R. Lockery and W. B. Kristan Jr. Distributed processing of sensory information in the leech ii: Identification of interneurons contributing to the local bending reflex. *Journal of Neuroscience*, 1990.

[31] Shawn R. Lockery, G. Wittenberg, W. B. Kristan Jr., N. Qian, and T. J. Sejnowski. Neural network analysis of distributed representations of sensory information in the leech. In David Touretzky, editor, *Advances in Neural Information Processing Systems II*, pages 28–35. Morgan Kauffman, 1990.

[32] Steven J. Nowlan. Gain variation in recurrent error propagation networks. *Complex Systems*, 2(3):305–320, June 1988.

[33] Mary B. Ottaway, Patrice Y. Simard, and Dana H. Ballard. Fixed point analysis for recurrent neural networks. In David Touretzky, editor, *Advances in Neural Information Processing Systems I*. Morgan Kauffman, 1989.

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

[35] Barak Pearlmutter. Learning state space trajectories in recurrent neural networks. Technical Report CMU-CS-88-191, Department of Computer Science, Carnegie Mellon University, Pittsburgh, PA, 1988.

[36] Barak Pearlmutter. Learning state space trajectories in recurrent neural networks. *Neural Computation*, 1(2):263–269, 1989.

[37] C. Peterson and James R. Anderson. A mean field theory learning algorithm for neural networks. Technical Report EI-259-87, MCC, August 1987.

[38] C. Peterson and J.R. Anderson. A mean field theory learning algorithm for neural nets. *Complex Systems*, 1, 1987.

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

[40] Ning Qian and Terrence J. Sejnowski. Learning to solve random-dot stereograms of dense and transparent surfaces with recurrent backpropagation. In *Proceedings of the 1988 Connectionist Models Summer School*, pages 435–443, San Mateo, CA, 1989. Morgan Kaufman.

[41] Steve Renals and Richard Rohwer. A study of network dynamics. *Journal of Statistical Physics*, 58:825–848, June 1990.

[42] A. J. Robinson and F. Fallside. Static and dynamic error propagation networks with application to speech coding. In Dana Z. Anderson, editor, *Neural Information Processing Systems*, pages 632–641, New York, New York, 1987. American Institute of Physics.

[43] Richard Rohwer. The "moving targets" training algorithm. In D. S. Touretzky, editor, *Advances in Neural Information Processing Systems 2*, pages 558–565, San Mateo, CA, 1990. Morgan Kaufmann.

[44] 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*, volume I. Bradford Books, Cambridge, MA, 1986.

[45] Patrice Y. Simard, Jean Pierre Rayzs, and Bernard Victorri. Shaping the state space landscape in recurrent networks. In D. S. Touretzky, editor, *Advances in Neural Information Processing Systems 3*. Morgan Kaufmann, 1991. To Appear.

[46] N. Toomarian and J. Barhen. Adjoint-operators and non-adiabatic learning algorithms in neural networks. In D. S. Touretzky, editor, *Advances in Neural Information Processing Systems 3*. Morgan Kaufmann, 1991. To Appear.

[47] Tadasu Uchiyama, Katsunori Shimohara, and Yukio Tokunaga. A modified leaky integrator network for temporal pattern recognition. In *International Joint Conference on Neural Networks*, volume 1, pages 469–475. IEEE, 1989.

[48] Alex Waibel, T. Hanazawa, G Hinton, K. Shikano, and K. Lang. Phoneme recognition using time-delay networks. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 37(3):328–339, 1989.

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

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

[51] B. Widrow and M. Hoff. Adaptive switching circuits. In *Western Electronic Show and Convention, Convention Record*, volume 4, pages 96–104. Institute of Radio Engineers, 1960.

[52] 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.

[53] David Zipser. Subgrouping reduces complexity and speeds up learning in recurrent networks. In D. S. Touretzky, editor, *Advances in Neural Information Processing Systems 2*, pages 638–641, San Mateo, CA, 1990. Morgan Kaufmann.