# Dynamical-Systems Behavior in Recurrent and Non-Recurrent Connectionist Nets

An Honors Thesis Presented
by

Jason M. Eisner

to
The Department of Psychology
in Partial Fulfillment of the Requirements
for the Degree of Bachelor of Arts
with Honors in the Subject of Psycholgy

Harvard-Radcliffe Colleges
Cambridge, Massachusetts
April 2, 1990

**Abstract**

A broad approach is developed for training dynamical behaviors in connectionist networks. General recurrent networks are powerful computational devices, necessary for difficult tasks like constraint satisfaction and temporal processing. These tasks are discussed here in some detail. From both theoretical and empirical considerations, it is concluded that such tasks are best addressed by recurrent networks that operate continuously in time—and further, that effective learning rules for these continuous-time networks must be able to prescribe their *dynamical* properties. A general class of such learning rules is derived and tested on simple problems. Where existing learning algorithms for recurrent and non-recurrent networks only attempt to train a network's *position* in activation space, the models presented here can also explicitly and successfully prescribe the nature of its *movement through* activation space.

I am indebted to Jay Rueckl, my advisor, both for his suggestions and for his support. Jay Rueckl and Greg Galperin provided computational facilities that proved indispensable. I would also like to thank my family and the many friends whose encouragement saw this project through its final stages.

# Contents

# 1 Introduction

In the 1960's, Minsky and Papert pointed to hidden units as a potential remedy for some of connectionism's problems. Recurrent connections have lately been attracting the same kind of interest. Much as hidden units extend the computational power of perceptrons, recurrent connections extend the computational power of feedforward networks.

The work reported here is ultimately concerned with both recurrent and non-recurrent networks. However, it focuses on network properties that are most evident (and most useful) in the presence of recurrence. These are *dynamical* properties of networks—properties describing how networks' states change or remain stable over time.

The paper has three major aims, as follows. First, to highlight the features of recurrence that make it useful. Second, to demonstrate that certain network architectures exhibit especially rich kinds of behavior. Finally, to develop a training algorithm that can produce the desired behaviors in networks that use these architectures.

## 1.1 The uses of recurrent networks

Since it is useful to focus on actual problems, the early sections of this paper will pay special attention to two domains in which recurrent networks have proved especially useful. These are the *constraint satisfaction* domain and the *temporal* domain. In a constraint satisfaction problem, the network is supposed to discover any regularities that hold among various static inputs. The temporal domain includes all those tasks where a network's inputs and/or outputs are to change over time in a principled way.

### 1.1.1 Recurrent networks and constraint satisfaction

The general constraint satisfaction task is simple. Various *patterns* (vectors of numbers) are shown to a network. The network is supposed to discover regularities in the set of patterns it sees. When it is shown only part of a pattern, it should correctly fill in the missing elements.

The purest form of constraint satisfaction makes no distinction between input and output nodes of the network. There is simply a set of *visible nodes*, which hold the patterns. A partial pattern can be "clamped" onto some of

the visible nodes—this means that the nodes' activations are held constant at the component values of the pattern—and the remaining, "free" visible nodes are supposed to assume a set of activations that could consistently complete the partial pattern.

It should be clear why recurrence is necessary for a network to perform this sort of operation. The net must be able to run both backwards and forwards, depending on which visible nodes are clamped: sometimes node $i$ might have to influence node $j$ and sometimes vice-versa. If the network does contain mutually influential nodes like $i$ and $j$, then its graph must contain cycles. In short, the network must be recurrent.

Note that constraint satisfaction techniques would often be helpful if applied to other problems. Any network whose input patterns are often incomplete, or partly erroneous, might do well to filter them through such a procedure at the outset. Ideally, this filtering would not even be a separate phase of the network's operation, but would develop naturally as the network's internal representations learned to feed back and influence its input.

### 1.1.2 Recurrent networks and temporal problems

Recurrent networks are also well-suited to temporal problems, because recurrent connections help a network preserve information from one moment to the next. They allow the net to affect its own subsequent behavior.

For some temporal problems, the network does not have to preserve particularly complex information. A simple record of past input may suffice. This was roughly the approach of Sejnowski and Rosenberg's NETtalk (1987). NETtalk had no recurrent connections, but its "moving window" provided continuity in the input stream. A certain amount of its past input could continue to affect it.

This approach is not very general. An input buffer of fixed size can only hold a limited number of past events—but some tasks require memory for input events that happened arbitrarily far in the past. For example, one may have to remember the subject of an arbitrarily long sentence. The "cumulative XOR" task illustrates the difficulty clearly. In this task, a network is presented with a continuous input stream of 0's and 1's, and is expected at every moment to output the cumulative XOR of all the input to date. The problem can be easily solved by a network that maintains just one bit of state information. To solve it by remembering actual input, however, would

require a buffer of infinite length, because every bit is significant.

Even when a problem can be solved by preserving past input, it may be more useful to preserve some other kind of state information instead. For most problems, a network does not have to remember the exact pattern of raw input data that it has seen, but only certain relevant features of those data. Indeed, the features of past input that the network must remember are typically the same features it was required to extract when it originally saw that input. A buffer for raw data is clearly superfluous here.

Finally—and most important—input buffers fail to capture temporal invariance in a natural way. Suppose a network is processing a stream of data that contains, among other things, copies of two special input sequences designated START and STOP. Say the network needs to recognize the following condition:

I, the net, have received a START sequence
since the last STOP sequence.

If the network relies on an appropriately long buffer to remember its past input, then it must learn how to detect START and STOP sequences at each position in the buffer. This requires a great deal of learning and a complete set of training examples. It would be much more natural for the input to simply affect some internal variable as it arrives.

In general, it seems most sensible to let a network use any internal representations that help it do its job. Certainly there is a strong case for letting networks have internal states that reflect their past input and processing. And short of augmenting a network's units with some sort of storage capacity, recurrent connections seem to be the only way that a network can achieve such states.

### 1.1.3 The computational power of recurrent networks

Recurrent networks are a proper superset of non-recurrent networks, and constitute a more powerful class of computational devices.[1] Hence a final reason to study them is simply to find out what they can do. Recurrent networks may be capable of many kinds of behavior other than constraint satisfaction and temporal tasks.

[1] Indeed, digital electronics was founded upon the flip-flop memory, a recurrent electrical circuit.

As is always true in connectionism, we are especially interested in the class of behaviors that our networks can *learn*. Unless there is a natural way to teach recurrent networks particular tasks, their power is not especially useful. Later sections of this paper will derive actual algorithms for supervised learning in recurrent nets.

## 1.2 Recurrent networks in practice

The algorithms of this paper fit into an existing body of research involving recurrent networks. The best-known models to date, which are reviewed below, can be easily divided into the two groups discussed earlier. Some perform constraint satisfaction; others perform temporal tasks.

### 1.2.1 Existing models of constraint satisfaction

A classic example of a constraint satisfaction device is the *interactive activation* (IA) model of letter perception (Rumelhart & McClelland, 1986). Part of the interest of this model stems from its ability to predict experimental results on letter perception in humans, but it is also an excellent example of how recurrence operates in the constraint satisfaction domain. The IA model has three levels of units: visual feature detectors, letters, and words. The feature detectors simply respond to different kinds of line segments; they excite the letters that are known to contain those segments. The words excite, and are equally excited by, the letters they contain. Finally, all words inhibit each other, being inconsistent hypotheses, and so do all letters.[2]

Stimulating the feature detectors causes a set of possible letters to be activated, some more strongly than others. The object of the model is to decide which of these possible letters are really present in the word shown. It does this through the way the units interact. Words try to inhibit each other, and the word with the greatest activation will tend to be most successful in inhibiting the others. Letters compete for top activation in the same way. It follows that if a word or letter starts out with slightly more activation than its competitors, it will be likely to end up with substantially more activation.

[2] Why should letters be inconsistent with each other? The actual IA model has four copies of all of its letter units: one set for the first position in a four-letter word, one set for the second position, and so on. (Each set has its own feature detectors.) Inhibitory connections only occur among letters in the same set.

However, the word and letter units interact so as to produce a solution that is consistent on both levels simultaneously. Initially likely letters, if they do not combine to make any word, can be suppressed by other letter units that are supported by word units.

In short, the IA model finds a set of activations over the letter and word units that is internally consistent and also consistent with the active feature detectors. (Two units are consistent if they have an excitatory connection and similar activations, or an inhibitory connection and dissimilar activations.) This is similar to the "pure" constraint satisfaction idea described in 1.1.1.[3] One can clamp the feature detectors to get plausible activations over the letter and word units, or clamp a few of the letter units to get plausible activations over the word units and remaining letter units, and so on.

The IA model is a model of perception, not of perceptual learning, and has so no explicit learning rule. This is a serious shortcoming. The other constraint satisfaction architecture discussed here does have a learning rule, albeit a slow one. This is the Boltzmann machine of Hinton and Sejnowski (1986).

The Boltzmann machine, like the earlier model by Hopfield (1982) from which it derives, has an unusual architecture. Each node can only output 0 or 1, and its output flits back and forth between these two values. The flitting is stochastic; its probabilities are arranged in such a way that the node's total output per unit time is, on average, a logistic function of its input. The logistic function is made steeper over time so that the network eventually settles into a fixed pattern of 0's and 1's. This is called lowering the system's temperature.

Any pattern the network settles into will tend to be highly consistent, in the above sense of the term: that is, any two units with a positive connection will tend to have the same value.[4] Furthermore, the network can be trained to settle into particular patterns. The weights should be adjusted gradually in such a way as to make the desired patterns slightly more consistent, and the network's actual patterns slightly less consistent. If the desired patterns match the actual patterns, the weight changes will cancel each other out.

[3] Adding connections from the letter units back to the feature detectors would make it 100% pure.

[4] The network's likelihood of settling into a particular pattern is an exponential function of the pattern's consistency (appropriately measured), where the exponential function depends on temperature and is exactly as steep as the logistic function.

This is a very attractive model: each weight affects the consistency of a pattern in a local manner, and hence can be adjusted using only local information. However, it suffers from an extremely slow learning rule. The odd network architecture also presents some practical difficulties. Units must be assumed to produce only fixed binary outputs, since if the logistic function is left gentle enough for the units to output a mixture of 0's and 1's, the state of the system as a whole will be unstable.

Perhaps the most unfortunate aspect of the Boltzmann architecture is that it cannot be extended to the temporal domain. Section 1.1.1 noted that constraint satisfaction can be a very useful feature within other networks. However, the Boltzmann architecture simply is not suited to temporal problems. In order to correctly respond to new input, a Boltzmann machine has to raise its temperature and then gradually lower it again. Once its temperature is high, stochastic forces may cause its state to change completely. Successive states of the network are therefore not guaranteed to have any relation to each other.

The ideal constraint satisfaction model would do the job of a Boltzmann machine without relying on randomness. Section 2.2.5 demonstrates the difficulty of adapting Boltzmann methods to non-stochastic nets. The trouble is that once the network is made deterministic, it will only be able to settle into one pattern (for a given starting state)—and will not necessarily choose the most *consistent* one. The network's choice, therefore, cannot be controlled simply by controlling pattern consistencies.

Without randomness, the particular pattern a network settles into is the result of complex time-governed interactions among its units. It is much harder to control those interactions than to control pattern consistencies. Nonetheless, section 3 will describe a method for doing just that.

### 1.2.2 Existing temporal models

The other important models using recurrent networks are models of temporal processes. Earlier, section 1.1 defined temporal tasks as "tasks where a network's inputs and/or outputs are to change over time in a principled way." In other words, for a device that accomplishes the task, past inputs and outputs must have predictive power with respect to future inputs and outputs.

The best-known model of temporal output is the serial order model of

Michael Jordan (1986). Jordan's model is based directly on the idea that past outputs should predict future behavior. His network is a back-propagation net with two kinds of input: a constant "plan" vector specifying what sequence of outputs the network is to generate, and a "state" vector that reminds the network what output it has just generated.[5] Under any given plan, the past outputs completely determine the future behavior of the system. If we identify state with output, as Jordan does, the function of the network itself is simply to compute the system's next state from its previous states.

Jordan's model is the inverse of a model like NETtalk, where limited past input was available to the system: here, limited past output is available. The technique successfully describes how a network can be trained to produce a simple sequence of "actions," on its own, without step-by-step guidance from the input. The resulting network also has an interesting tendency to make successive actions overlap in time when possible, by increasing their duration.

Jeff Elman (1988) has devised a variation of Jordan's model that can deal with the temporal structure of input. His architecture, instead of feeding the network its past outputs, feeds it the past values of its hidden units. This is a simple but useful idea. Whereas Jordan's networks can only consider states prescribed by the environment—the output vectors—Elman's can create their own state variables.

Elman has demonstrated that a network with this architecture can discover non-trivial ordinal structure in its input. With relatively short training regimens, it can be made to predict at each time step the input value it is about to receive. And although Elman does not mention it explicitly, the architecture should be capable of producing Jordan-like output sequences as well.

Both Jordan and Elman rely on back propagation to train their networks. They use the standard form, which does not properly apply to recurrent architectures, because the recurrent connections in their models have fixed weights. For the purposes of the learning algorithm, the recurrent connections might as well not exist. The learning algorithm can treat the model as a feedforward network whose input just happens to reflect its previous state.[6]

[5] To arrange that outputs before the most recent one can help determine the next output, Jordan uses a state vector whose value is an exponentially weighted average of all past outputs. The state vector is updated using recurrent weights.

[6] Or so Jordan and Elman imply. In truth, their learning algorithm is approximate. The

These two models use only restricted recurrence. The feedback connections are carefully chosen and not permitted to change. While the networks are successful at solving certain problems, they do not allow the full range of behavior possible in unrestricted recurrent nets.

Williams and Zipser (1988) overcome these restrictions by deriving a full-fledged extension of back propagation.[7] Their gradient-descent algorithm can be applied to networks that contain any recurrent connections whatsoever. However, it pays a price for this generality: it is nonlocal and computationally intensive.

In the Williams and Zipser paradigm, a continuously running network receives input at each simulated time step, and has a target output at each time step. The network's error at a single time step is given by the usual sum-of-squares expression; but the learning rule always adjusts weights so as to reduce the network's *total* error, summed over all time steps. In other words, the network learns to compute the correct outputs any way that it can. If the target outputs are determined by the current inputs alone, for example, the network will learn an ordinary mapping. If they are determined by the past inputs, it will develop some sort of state variables. If they are determined solely by the passage of time, it will learn to produce a Jordan-style output sequence.

One might describe this algorithm as powerful, but greedy. Its time requirement is $O(n^4)$, where n is the number of nodes. The algorithm set forth in section 3 will turn out to work on similar principles. It is rather more powerful, and, unfortunately, equally greedy.

## 1.3 Summary

We have now seen how several previous researchers have thought about recurrent networks.

---

fixedness of the recurrent weights does not really make them irrelevant to the learning algorithm. To see why, consider the hidden units in Elman's network. Standard back propagation tries to arrange for them to have activations that are helpful to the output units; but it misses the chance to make them helpful to the hidden units on the next cycle.

In other words, error ought to be propagated backwards along the recurrent connections, even if the weights on those connections are not modifiable. This requires a recurrent learning rule of the Williams and Zipser sort. For Jordan's and Elman's models, however, ignoring this limited recurrence does not seem to have hurt the learning procedure much.

[7] Rumelhart, Hinton, and Williams (1986) had already given an approximate extension.

In the constraint satisfaction world, the IA and Boltzmann machine models take nearly opposite approaches. Units in IA have continuous activations and influence each other steadily, so that when one unit's activation overtakes another's, it is a qualitatively significant event. By contrast, there is no such thing as a steady influence in Boltzmann machines. Each unit simply modulates the probabilistic behavior of other units. The network's behavior does not change at all over time, except insofar as the temperature does.

The temporal processing research takes a different view. In the models of Jordan and Elman, the point of recurrent connections is to permit feedback. A network that is to operate in time needs knowledge of its past history. Recurrent connections, then, transmit information. In the case of discrete-time networks, they transmit a new packet of information on each time step.

Finally, Williams and Zipser take no particular position on the proper role of recurrent connections, except to note that they increase a network's computational power. For them, the main point of *any* connection is to make it easier for a network to produce exactly the right outputs at the right times.

The next section will discuss the constraint satisfaction and temporal domains in more detail, and will begin to make the case for a new way to think about recurrent networks: in terms of their dynamics. The whole point of connections is to allow units to influence each other. If a network operates over time, its connections determine its dynamical properties—its patterns of movement through activation space. The next section concludes that these dynamical properties are important. A network's ability to solve a problem often depends on the way its activations change over time, or remain the same.

# 2 Some Theoretical Observations

A primary aim of this paper is to develop an actual computational model. We begin by considering what *sorts* of models might be successful. The following remarks, then, are exploratory. They offer some techniques with which one might try to design a useful recurrent net, and provide some of the motivation for the model described later.

As we have seen, some recurrent network architectures are good at responding over time. Others are good at discovering regularities in their input. An ideal approach would be able to address both problems; we discuss

them in turn.

## 2.1 Thoughts on temporal pattern processing

### 2.1.1 The usefulness of gradual-response nets

In the temporal domain, the most useful recurrent architectures may be those that exhibit a *gradual response*. In discrete-time networks, a unit's activation at time $t+1$ simply replaces its activation at time $t$. A gradual-response network, by contrast, runs in continuous time. Units change their activations continuously; a unit's activation at $t+1$ follows from the accumulation of infinitesimal changes over the interval $(t, t+1]$.[8]

If we implement such a net on digital hardware, we are obviously forced to use discrete time steps. However, we can make these time steps as close to infinitesimals as we like. The system can be described without the assumption of discrete time; it performs a computation that is sensitive to the total time elapsed, not to the number of time steps.

Such a network is well-suited to temporal problems because its state changes continuously. As with any computational system, the environmental input may vary with time, perhaps discontinuously. But the net will respond gradually even to abrupt changes in input. This property is what allows it to preserve state.[9]

[8] To put this a little more formally, the activation of a unit, $act$, is to be differentiable with respect to time, so that $act'(t)$ exists and

$$act(t+1) = act(t) + \int_{\tau=t}^{t+1} act'(\tau) d\tau.$$

[9] By contrast, a feed-forward network ordinarily does not take state into account at all. The network responds independently to input patterns at times $t$, $t+1$, and so forth; its units' activations at time $t+1$ have nothing to do with their activations at $t$.

We can certainly imagine discrete-time networks whose state at $t+1$ is a function of both their input at $t+1$ and their state at $t$. This is in fact the approach of Elman (1988). Such a model has certain disadvantages, however. It assumes that the system's input (and its desired output) can be described as changing only at regular clock ticks. If more frequent ticks become necessary to describe the environment, the net might have to be completely retrained, because its operation depends in a fundamental way on the step size, i.e., the temporal graininess with which the environment is sampled. Furthermore, since activations do not change continuously, state properties of a discrete-time network

Note that state *preservation* is not strictly necessary for all temporal processing. All that is really required is that the system's current state can somehow *influence* its later behavior; its current state need not persist over time. But lasting states are important for two reasons. First, when states remain relatively constant over short periods of time, the network is not affected by slight temporal distortions in its input, and as Jordan (1986) observes, will produce outputs that are "spread in time" (i.e., non-instantaneous). Second, many temporal problems do happen to require the preservation of state over longer intervals. In this context, the tendency to preserve state is sometimes referred to as *memory*. It allows a system to take advantage of the relative stability of the physical world, to keep track of the subject of a sentence, and so on.

A gradual-response net is guaranteed to preserve state over at least the very short term, simply because its activations change continuously with time. A unit's activation may also stay constant for longer. In the standard case where the activation of a unit increases in proportion to the total input it receives, a unit will be relatively stable if it receives little input from other units and has little tendency to change on its own (e.g., by decaying).

To demonstrate that recurrent nets are well-suited to maintain their states over long periods of time, we can consider the extreme case in which the net's *only* goal is to keep its activations constant. In practice, such a net would not be very interesting. We ordinarily want activations to change in response to input and/or the passage of time. However, certain parts of the network might learn to act as memories, and stay constant except in the presence of certain external inputs. We want to be sure that the behavior is natural for our network to achieve.

It is easy to ensure that, by default, units receive little input. We can simply initialize weights to small values before any learning takes place. Moreover, if the weights are initially large, a network that wants to be stabler can easily learn to make them smaller. If $I$ is the input vector and $A$ is the vector of activations in the network, the weights serve to map $\{I, A\}$ directly onto $dA/dt$, with no hidden units. The network only has to learn the $\mathbf{0}$ mapping that takes each $I, A$ to the zero vector. Any rule that adjusts network weights in a manner consistent with the delta rule will learn this mapping easily; in

have no intrinsic tendency to persist over time—a property whose possible importance will be discussed in a moment.

particular, any gradient-descent rule will suffice.[10]

Section 2.1.3 discusses other ways to encourage a recurrent network to preserve state.

### 2.1.2 The dynamical systems approach

Recurrent networks can do more than just sit in the same state, of course. Input may affect them; and even when input is held constant, the activations in a network may continue to change. Indeed, the behavior of the network may depend in complex ways on both the input and the current activations of the network.

This notion is captured nicely by a construct used in mathematical physics, the *dynamical system*. A dynamical system consists of two parts: a continuous *state space*, which represents the set of possible states of the system, and a continuous function $v$ over the state space, which specifies an *instantaneous velocity vector* at each point in the state space. The idea is that the system moves continuously through state space; its speed and direction in state space are completely determined by its current position and specified by $v$. Any continuous path that the system may take through state space is called a *trajectory*.

For example, consider a pendulum. The state of the pendulum is specified along two dimensions. One dimension is the interval $[-\pi, +\pi)$, which represents the possible angles that the pendulum may make with the vertical. The other dimension gives the pendulum's rate of rotation. If we know the pendulum's current angle and rate of rotation, we can compute how quickly

[10] This discussion is not unique to a gradual-response architecture. In a model like Elman's, where the total input to a node replaces its activation instead of modifying it, the network could still preserve state. However, it would need to implement a different mapping to do so. Let $C$ be the vector of activations on the context units (which, in Elman's model, record the value of $A$ from the last time step). The weights in the network map $\{I, C\}$ onto $A$. For the network to retain its state, the weights must map each $\{I, C\}$ to $C$.

In other words, all weights should be 0 except for any weight from a context unit to its associated hidden unit, which must be 1. We could initialize the weights in this manner (somewhat awkwardly). We could also learn them if necessary. However, the mapping is somewhat harder to learn than the direct $\mathbf{0}$ mapping that our architecture requires. Its output is not a constant; and depending on the topology of the network, it may be necessary to train multiple layers of weights between $C$ and $A$. Static-state behavior is slightly more "natural" than this for a gradual-response network.

each is changing. In other words, if we know its current state, we know how it is currently moving in state space.

Any gradual-response network with fixed topology and weights, differentiable output functions, and constant input, specifies a dynamical system in input space $\times$ activation space.[11] If we know the current constant input vector to the network and the current activations of all the units, then we know how the input is changing (not at all) and how the activations are changing (e.g., in proportion to their units' net inputs).

The network may be initialized at any point in this state space and "released," i.e, allowed to run. Its own architecture determines what trajectory it follows after that. Changing the input means sliding the system to a different point in its state space and releasing it again.

This dynamical systems perspective is potentially useful, because it allows us to borrow some terminology, and equips us to think about certain phenomena that are commonly observed in dynamical systems. For instance, an *equilibrium point* of the system is any point in state space whose associated vector is 0—that is, any point from which the system will not move. Most such points happen to be *point attractors*. To say that $p$ is an attractor means that if the system is released at a point $q$ sufficiently near $p$, it will trace a trajectory that reaches $p$ (and stays there). The complete set of such points $q$ is called the *attractor basin* of $p$.

In general, an equilibrium set is any subset $S$ of state space having the property that if the system is released anywhere in $S$, it remains in $S$. $S$ need not be a point; sometimes, for example, one sees *periodic trajectories*, where the system repeatedly traces a closed loop in state space. Nearby trajectories may converge to this loop, in which case it is called a *cyclic attractor* or *limit cycle*, and has a basin like any other attractor.

Note that the state space of a dynamical system is divided up among lower-dimensional attractors and their basins, other equilibrium points, and points on divergent trajectories.

[11] *Input space* is the Euclidean space of possible input vectors. *Activation space* is the Euclidean space of possible activation vectors. In some models, like those described later, one provides input to the system by setting the activations of certain "input units." In this case there is no need for a separate input space.

### 2.1.3 Dynamics of small clusters

Within a recurrent network, it is possible to build small clusters of highly interconnected units that exhibit interesting and stable dynamic behaviors in their own activation spaces. These clusters need only be 2 to 5 units in size. Because they are capable of both time-dependent behavior and state preservation, they offer some exciting possibilities for future work in temporal pattern processing.

This section argues for the potential usefulness of cluster-based architectures. The argument is made not only for its own sake, but also to underscore the importance of being able to teach dynamical behavior to networks.

Actual cluster dynamics were explored through a computer simulation. The clusters studied were tiny, fully recurrent networks, made up of ordinary connectionist units with logistic output functions and decay. As it turns out, such minature dynamical systems are capable of features such as these:

- *Single point attractors with different basin dynamics.* A cluster with a single point attractor $p$ is useless as a memory, since it will always end up at the attractor. However, by choosing the recurrent weights carefully, we can achieve interesting dynamics in the basin of $p$. The basin dynamics determine how the cluster responds in the short term if an input stimulus moves it away from the attractor.

  For instance, one kind of cluster will converge very quickly when near $p$, but very slowly when far away. In other words, if it is displaced from $p$ by a strong input, it will stay away from it for a certain period of time before returning. If it is at the attractor, this indicates that it has not received such an input for at least this much time.

  Another possible behavior, for a cluster that has been jiggled off $p$ in the right direction, is to loop around and return to $p$. For example, a slight input stimulus might cause a cluster to snap out to high activation and right back again. In other words, the cluster generates a standardized pulse when prodded.

- *Multiple point attractors.* A cluster with two or more attractors is useful as a memory. Appropriate input can move the cluster from one attractor to another. A simple example is a pair of mutually inhibiting units, which has two stable states (on-off and off-on) and acts like an

ordinary flip-flop. These states are more useful as stable memories than the persistent states described in 2.1.1: without giving up the ability to be affected by strong inputs, they resist being affected by weak ones.

With multiple attractors, each basin can still adopt the same sorts of dynamics that we considered for single attractors earlier. It is even possible to build a cluster with two attractors, $p$ and $q$, in such a way that the basin of $p$ is non-convex and curves partway around the basin of $q$. This allows the *same* source of input to move the cluster's state from $p$ to $q$, and if the stimulus is repeated, from $q$ back into the basin of $p$. In other words, although the input source always pushes the cluster in the same direction in its activation space, input serves to toggle the cluster between $p$ and $q$. The initial state of the cluster determines where it ends up after input.[12]

- *Quantum units.* By carefully creating a cluster with several multiple attractors, and by paying attention to the output of only one node in the cluster, one can implement a potentially useful device called a *quantum unit*. A quantum unit behaves for the most part like an ordinary connectionist unit, but its activation space is discrete. It has only a small finite number of characteristic activations. Any induced activation will tend to snap to the nearest of these. The unit always outputs its activation (no logistic or threshold function is necessary).

[12]This is a somewhat counterintuitive result. To see that such toggling behavior is possible, consider how we would produce it by augmenting an ordinary two-state memory cluster. Suppose we already have a cluster C with two attractive states, which we call -1 and +1. Ordinarily, a positive input puts C in state +1; a negative input puts it in state -1. We want to arrange things so that a positive input always toggles the state of C.

The solution is simple, and similar to the solution for the classic XOR problem. We introduce an extra semilinear unit, D, that is weakly excited both by the input and by C, and that can inhibit C just enough to turn it off.

If C is at -1, a positive input pulse toggles it to +1. D will only cross threshold if a positive input pulse arrives when C is already at +1. In that case, once the input pulse ends, D's high output is able to move C back to -1. This system of C and D, then, produces the behavior promised in the text.

The attractor at $(C = -1, D = 0)$ does indeed have a non-convex basin. Choosing appropriate numbers, the basin contains the point $(C = +3, D = 1)$, which is the state the system assumes if C is at +1 and a positive input pulse arrives. The basin also contains the attractor itself. But it certainly excludes the point halfway in between, $(C = +1, D = 0.5)$, because that point happens to be the other attractor!

Such units could be useful for several reasons. First, they respond only to strong inputs. Second, although they can produce a range of outputs, those outputs are quantized in the style of threshold units. Third, they retain their state over time (and can accumulate state changes over time). Finally, their relative insensitivity to changing input would make them quick to settle when used in a recurrent network.

- *Limit cycles.* Finally, it is possible to produce a limit cycle within a small cluster. Surprisingly, this behavior requires only three units. One unit remains constant (acts as a bias) while the other two trace a circle in the plane. This limit cycle is a stable attractor whose basin is the whole space: if the cluster's trajectory is jolted off the cycle, it will spiral inward or outwards and return to the trajectory. (By increasing or decreasing all the weights, we can make it return faster or slower.) Such a cluster can easily be made to generate regular pulses in another unit. This feature might be useful as an internal "clock" for a larger network.

All these behaviors may be easily verified with a short computer program. It is easy to imagine network topologies that would make good use of such clusters. For example, we might construct a continuous-time feedforward net using both ordinary units and quantum units. Such a network would retain state information from one pattern presentation to the next, changing state variables only under strong input.

More generally, any cluster implements some deterministic finite automaton (DFA) that has only a few states. The input lines to the cluster supply the DFA with transition elements; the passage of time may also serve as a transition element. In a hierarchical architecture where most clusters take their input from other clusters, the higher levels may be able to recognize complex temporal patterns in the environmental input. This would be a worthwhile area to explore.

Unfortunately, without a learning rule, these are simply observations about useful topologies. It is unclear how we would train a cluster to adopt a particular behavior—or how we would decide what behavior the cluster *ought* to adopt in order to be useful. The work in section 3, which develops a learning rule for dynamical systems, was performed with an eye toward solving either or both of these learning problems.

## 2.2 Thoughts on constraint satisfaction

### 2.2.1 The need for hidden units

The ambition of any young constraint satisfaction network is to discover the regularities in its input environment. If the activations of visible units $V_1, V_2, \ldots V_n$ represent environment variables, the object is to find weights that capture the relations among these activations. In particular, given values for a subset $S$ of the $V_i$, the mature network should be able to predict values for visible units not in $S$.

Some constraint satisfaction networks have no hidden nodes, and simply try to find direct relationships among the visible units. The network can take advantage of these with appropriate direct weights between correlated or anticorrelated units.

Typically, such a network is expected to compute a consistent solution for all visible nodes at once. This parallelism is often helpful. An XOR problem mentioned by Hinton and Sejnowski (1986) provides a good illustration. In this problem's environment, there are four variables, $A$, $B$, and $C$, and $D$. Four patterns over these variables are regularly found in the environment: $0000, 0110, 1010$, and $1101$. Inspection shows that $C$ is the exclusive "or" of $A$ and $B$, while $D$ is their logical "and."

If a network without hidden units is given the values of $A$ and $B$, no set of direct weights will enable it to find the value of $C$. (Neither $A$ nor $B$ is at all correlated with $C$.) Luckily, the value of $D$ *can* be found using direct connections from $A$ and $B$. Since the network is looking for values for all the nodes at once, it will determine the value of $D$—and then $A$, $B$, and $D$ together can determine $C$ with only direct connections.

The point of this example is that $D$, an environment variable, performs a necessary function in the computation of $C$. The network only manages to determine $C$ because it is simultaneously interested in determining $D$, and has already learned from the environment how $D$ is related to the other variables.

This is the same trick that Rumelhart and McClelland (1981) used in their interactive activation model. Their model was designed to recognize words from their visual features, but it did not attempt to correlate features in isolation with particular words. Rather, features helped to predict the presence of letters, which in their turn were correlated with the words. The

model would certainly have failed if feature units had just been connected directly to word units.

Unfortunately, intermediate representations as helpful as letters may not always be available from the environment. To capture the regularities of the environment, the network may have to compute properties that are not recorded by any visible unit. This means training hidden units. In the XOR example mentioned earlier, the network might be able to do without $D$ if it could develop a hidden unit to assume the same function.

In general, hidden units are necessary to do constraint satisfaction when the constraints are complex, just as they are necessary to implement complex mappings in feedforward nets. Hidden units are the only way to detect important features of the environment that are not directly available in the environment.

### 2.2.2 A first attempt at a satisfaction model

The remainder of section 2.2 attempts, unsuccessfully, to develop a straightforward model for constraint satisfaction in a recurrent net. The approach fails, but for reasons that are worth understanding: in particular, because it pays insufficient attention to network dynamics. The model described in section 3 tries to address its problems.

The model takes the same tack as Hinton and Sejnowski (1986), who wrote:

> We would like to find a set of weights so that when the network is running freely, the patterns of activity that occur over the visible units are the same as they would be if the environment was clamping them. (p. 292)

Rather than using a Boltzmann-style device with stochastic binary units, however, we will use a gradual-response network of the sort described in 2.1.1. At any given time, some or all of the visible units may be designated as *clamped*, which means that their activations are not permitted to change. The dynamics of the network are given by the equations

$$a_i(t+\eta) = \begin{cases} a_i(t) & \text{if } i \text{ is clamped;} \\ a_i(t) + \eta\, net_i(t) - \eta\, decay(a_i(t)) & \text{if } i \text{ is free} \end{cases} \qquad (1)$$

where

$$net_i(t) = \sum_j f(a_j) w_{ij} \tag{2}$$

(Here $a_i$ is the activation of unit $i$; $net_i$ is its instantaneous input, defined by equation (2); and $w_{ij}$ is the weight to $j$ from $i$. All three may range from -infinity to +infinity. There are also network parameters that figure in the above equations. $f$ is the continuous output function for all units; *decay* is the continuous and invertible function that specifies the instantaneous decay rate of a given activation; and $\eta$ is the size of the time step. Good choices are to make $f$ the logistic function and let $decay(a)$ be proportional to $a$.)

To train the unit on a pattern, we clamp the visible nodes with that pattern and wait for the network to settle. We want our weights to have the effect that on every pattern, each visible unit will receive exactly enough input to maintain its clamped activation. In other words, we want to train the network so that the clamped units can be released and still hold the correct values at which they were clamped. This is similar to the learning procedure in a Boltzmann machine.

### 2.2.3 A learning rule for the non-resonant case

We see from (1) that unit $i$ is at equilibrium if

$$a_i(t+\eta) - a_i(t) = \eta net_i(t) - \eta decay(a_i(t)) = 0,$$

or in other words,

$$net_i(t) = decay(a_i(t)). \tag{3}$$

This means that to stay at its current activation, a unit must get just enough input to balance its rate of decay. Note that it is decay that keeps $a_i$ from becoming infinite.[13] We may call this a *subsistence input*. Since clamped nodes are supposed to get just enough input to maintain their activations, our learning rule will try to make the above equation hold for all clamped

[13]Since *decay* is required to be a continuous invertible function, the equilibrium value of $a_i$ under constant input is uniquely given by the continuous function $decay^{-1}(net_i)$. Thus if $net_i$ does not increase without bound while the network is running, $a_i$ will not do so either. This condition on $net_i$ is easy to arrange. Choosing the logistic function for $f$ restricts the output values of all units $j$ to the interval $(0, 1)$; then for fixed weights $w_{ij}$, the input to $i$ is bounded.

nodes. That is, the nodes' actual activations, to which they are clamped, ought to be their equilibrium activations as well.

For a given clamp pattern, then, we define the local error at a clamped node $i$ by

$$e_i = net_i - decay(a_i). \tag{4}$$

We want a learning rule that minimizes the global error measure $E = \frac{1}{2}\sum_i e_i^2$ for all patterns. Following the approach of Rumelhart, Hinton, and Williams (1986), we will derive a rule that descends against the gradient of $E$ in weight space.

We begin by considering a special case, where there are no connections among the hidden units. A hidden unit may connect only to visible units. This is an easy case because when the visible units are clamped, there is no resonant behavior in such a network. Changing a weight will affect the activation of one unit at most, and in a straightforward way.

When we release the clamped units, of course, resonance will become possible. But if we have trained the network well, no activations will change. There will indeed be recurrent cycles operating in the network, but they are now exactly suited to maintain the correct activations. Where the visible units formerly produced the output that they were instructed to, now they produce the same output freely.

The gradient of $E$ for this case is computed as follows:

$$\frac{\partial E}{\partial w_{ij}} = \sum_{k \text{ clamped}} e_k \frac{\partial e_k}{\partial w_{ij}} \tag{5}$$

There are two cases for determining $\partial e_k / \partial w_{ij}$. If $i$ is a clamped node,

$$\frac{\partial e_k}{\partial w_{ij}} = \frac{\partial net_k}{\partial w_{ij}} = \delta_{ik} f(a_j) \tag{6}$$

(The Kronecker delta, $\delta$, represents the identity matrix. It has the value 1 if its subscripts are equal, 0 otherwise. Here it is used as a notational convenience.)

On the other hand, if $i$ is free, we can write

$$\frac{\partial e_k}{\partial w_{ij}} = \frac{\partial e_k}{\partial a_i} \frac{\partial a_i}{\partial w_{ij}} \tag{7}$$

$$= \frac{\partial net_k}{\partial a_i}\frac{\partial a_i}{\partial w_{ij}} \tag{8}$$

$$= \frac{\partial net_k}{\partial f(a_i)}\frac{df(a_i)}{da_i}\frac{\partial a_i}{\partial net_i}\frac{\partial net_i}{\partial w_{ij}} \tag{9}$$

$$= w_{ki} f'(a_i) decay^{-1}{}'(net_i) f(a_j) \tag{10}$$

(The final line of this derivation is the one that takes advantage of our assumptions. The substitution in the third term requires that the network has settled—it follows from the equilibrium condition (3)—and the substitutions in the first and fourth terms only work for the present non-resonant case.)

We can summarize these results as

$$\frac{\partial E}{\partial w_{ij}} = \sum_{k \text{ clamped}} e_k \frac{\partial e_k}{\partial w_{ij}} \tag{11}$$

$$= \begin{cases} e_i f(a_j) & \text{if } i \text{ is clamped;} \\ (\sum_{k \text{ clamped}} e_k w_{ki}) f(a_j) f'(a_i) decay^{-1}{}'(net_i) & \text{if } i \text{ is free.} \end{cases} \tag{12}$$

In general, $\partial E/\partial w_{ij}$ is given by multiplying an error term at $i$ by the output of $j$, and by an additional factor that describes how the output of $i$ changes with its net input. This is not unlike the rule for back propagation.

### 2.2.4 Extending our rule to the resonant case

It turns out to be very difficult to extend this learning rule to the more general case. If we permit resonance during training, it is simply too much work to calculate the weight changes.

Briefly, the problem lies in the computation of $\partial a_i/\partial w_{ij}$. Suppose that $f(j)$ is positive and we increase $w_{ij}$. Then the input to $i$ from $j$ will certainly increase. The effect of that increase on $a_i$, however, depends on $i$'s interaction with the rest of the network. If $i$ excites a node that excites it in turn, for example, then the effect will be augmented through that other node. On the other hand, if excitation to $i$ causes $i$ to be inhibited more (or excited less) by other units, then the effect of the additional input will be damped.

This interaction of $i$ with the rest of the network has nothing in particular to do with $j$ or $w_{ij}$. We can isolate the interaction itself by introducing a further abstraction. Let $stim_i$ be the input from an imaginary electrode stimulating node $i$. We redefine $net_i$ to add this new source of input. In

practice, $stim_i$ is always zero—but we can still differentiate with respect to it. So we displace the infinitesimal change to $w_{ij}$ onto $stim_i$, allowing us to write

$$\frac{\partial a_i}{\partial w_{ij}} = \frac{\partial a_i}{\partial stim_i} f(j).$$

Let $c_{ij}$ stand for $\partial a_i / \partial stim_j$. We merely need to compute $c_{ii}$ for each unit $i$ in the network. Unfortunately, this turns out to be an inherently serial computation. If we derive an equation relating the values of $c$ in the network,

$$c_{ij} = decay^{-1\prime}(net_i) \sum_k w_{ik} f'(a_k) c_{kj},$$

we see that they cannot all be determined from each other by a single relaxation computation. The only interrelated values of $c$ are those with the same second subscript. In order to find $\partial a_i / \partial stim_j$, we must actually stimulate $j$ and measure the resulting changes in the network's activation vector; so finding values for all the $c_{ij}$ requires us to stimulate all the $j$, one at a time. This is an ugly result. It is similar to saying that the only way to determine the error gradient in weight space is to tweak each weight individually and see what happens to the error.

If we require that weights be symmetric, which in a Boltzmann machine is the characteristic that allows error gradients to be determined locally, we can get so far as proving that $c_{ij} = c_{ji}$ for all $i$ and $j$. Even so, we still have to compute $c_{ii}$ separately for each $i$. A simple example suggests that no amount of cleverness will make it possible to compute the $c_{ii}$ in parallel. Let $i$ be a clamped unit and $j$ be any other unit. Then $c_{ii} = 0$, and $c_{ij} = c_{ji} = 0$, but we have no idea from these numbers what $c_{jj}$ is.

As a last hope, one might try generalizing the non-resonant rule from its form alone. There are a few generalizations that look promising. When tested empirically, however, none of them can reliably minimize error on more than one pattern at a time. One presumes this is because they do not perform gradient descent.[14]

[14]Even if weights move in the right direction on a given pattern, usually enabling the network to learn that pattern in isolation, they do not necessarily move at the right speed. This means that when multiple patterns are being trained, the sum of the weight changes for an epoch may not have the right sign (even if all the individual summands do).

### 2.2.5 The problem with this approach

The above discussion illustrates the difficulty of computing the correct weight changes for recurrent nets. It is far harder to minimize our error measure on a resonant net than on a non-resonant one.

The constraint satisfaction problem is harder still, because the derivation above was oversimplified. The analysis was only valid with respect to the error measure given. To solve the constraint satisfaction problem, a network's error measure actually has to be more sophisticated than that.

To see why, suppose we have trained all the putative error out of the net, so that $E = 0$. For every clamped pattern, the network now succeeds in delivering subsistence inputs to the clamped visible nodes. The nodes will maintain their activations when they are freed; the system knows not to stray from the right answer. This means that each pattern has become an equilibrium point in activation space.

This is well and good, but the problem is that the network may have many other equilibrium points as well. Even if we assume that a pattern is an attractor, we cannot be sure that it has a very large basin. The system may not settle to the attractor unless it starts close by. So we can have a perfectly trained network that is nonetheless unable to generate the correct solutions without prompting.

For difficult tasks, this situation almost always happens in practice. If we train the reasonable non-resonant network of section 2.2.3 on the symmetric XOR problem, whose clamped patterns are 000, 010, 101, and 110, it is usually able to reduce $E$ to 0 within a dozen or so epochs.[15] However, it has not really solved the symmetric XOR problem in any useful sense. If we set the three visible units to the activations $(1, 0, 0.5)$ and clamp the first two, the third unit may assume an activation close to 1—but a ludicrous activation like -2.31 is just as likely.

Why exactly should this be so? We have trained the system so that, when a pattern $p_0$ is clamped on the visible nodes, the hidden units assume a state $S_0$ that supports $p_0$. However, if we do not clamp $p_0$ exactly, the hidden units may not move toward state $S_0$ at all—or toward $S_1$, $S_2$, or any state that supports a pattern in the training set. We have left some visible nodes free, and their values may be crucial in getting the hidden units to assume a

[15] This result is for networks with one to four hidden units and no visible-visible connections.

particular state.

How can we fix this problem? Again, it is useful to regard the network as a dynamical system. The problem at hand is *not* strictly a problem about keeping the visible nodes in place. Rather, it is a problem about getting them to move to the right place and *then* keeping them there. In other words, the learning rule needs to extend the basin of an attractor that it already knows how to create: it needs to control the dynamics of the system.

## 2.3 Summary of Theoretical Observations

We have now defined a straightforward architecture for a gradual-response recurrent network, and discussed how such a network might learn to do temporal pattern processing and constraint satisfaction. In both cases, we have run up against the problem of training the network to be some particular dynamical system.

What would be really helpful is a learning rule that could train the network to assume any desired dynamics. It is just such a rule that we will develop in the next section.

Adopting arbitrary dynamics is an intrinsically difficult problem, and it should be noted that it is more general and perhaps more difficult for the network to solve than the particular dynamical problems we have encountered so far. Simpler algorithms may exist for special cases. It may be, for example, that there are easy ways to train a cluster to be a quantum unit.

Nonetheless, in the interest of generality, the remainder of this paper will focus on a learning rule that is sufficient to cover any dynamics we want to induce in a network, recurrent or otherwise. We will see that such a rule exists, derive its form, and attempt to apply it to particular learning situations.

# 3 A General Model and its Learning Rule

## 3.1 Conception of the general model

### 3.1.1 Molding a dynamical system

Let us return briefly to the unsuccessful constraint satisfaction model of section 2.2. That model learned how to maintain visible nodes already at

their target values. It was at a loss, however, when it had to force the visible nodes to change their activations.

A better model would be able to make the visible nodes change their activations. Consider what should happen if the activation of unit $i$ is wrong. If the activation is too low, we are only interested in stimulating it enough to make it increase, at whatever speed; any large enough input will do. Similarly, if the activation is too high, all the model must do is to deliver any input low enough for the node to decrease. The input to a node must be precise only when the model needs to sustain the node exactly at its target.

We can design an error measure that recognizes these conditions. Such a measure must pay attention to whether activations are increasing or decreasing. In particular, it must rely on equation (1), which shows that $a_i$ will increase whenever $net_i > decay(a_i)$, and decrease whenever $net_i < decay(a_i)$.

Indeed, we can develop error measures to prescribe the dynamics of the system in any detail whatsoever. Most traditional error measures have only considered $a_i$ for each unit. When working with dynamic recurrent nets, however, we may also want to consider $da_i/dt$. It follows immediately from (1) that

$$\frac{da_i}{dt} = net_i - decay(a_i). \tag{13}$$

This result holds for discrete-time simulations, where $\eta > 0$, as well as in the limit case where $\eta$ approaches 0. It provides a local definition of $da_i/dt$ that an error measure can easily take into account.

In general, we can choose error measures that consider the system's position in `A`, its direction of movement in `A`, and the current time. For example, a measure might require that the system's direction be related to its position in some time-dependent way. The formal algorithm that we are soon to consider is able to minimize such an error measure—unlike other learning rules.

### 3.1.2 The role of input

At the same time that the net is being required to perform certain stunts in activation space, it may also receive some sort of input. Typically, this input will be useful in helping the network achieve the desired behavior (much as knowing the problems on a test helps one get the right answers!).

The environment can supply the network with input in one of two ways. For tasks where the network needs only one input per trajectory, the starting activation $A_0$ can play this role. Different behaviors are required of the network depending on where it starts in A. (Note that the error measure may vary for different values of $A_0$.)

For other tasks, the input may change over time. Here the preferred way of presenting input is to clamp some of the units to input values. Changing the clamping pattern changes the input. (Jordan's (1986) plan vector works this way.) This method conforms well to the dynamical system perspective. We always want the network's behavior to be some function of its input. In this case, that means its behavior is to be a function of its current position in A.

### 3.1.3 How to use the error measure

Once we have chosen an error measure $E$ for a particular dynamical problem, we want to choose weights that will minimize it. But $E$ is defined at every point of activation space. At which points do we need to minimize it?

Let A represent activation space. In order to induce the correct dynamics for the entire dynamical system, we might try to minimize the surface integral of $E$ over all of A. This would create a very robust network. Regardless of the place in activation space where the net was released, it would behave according to the dynamics prescribed for it.[16]

Unfortunately, there is usually no set of weights that will achieve low error *everywhere* in the space. We have seen that a network often requires hidden units in order to achieve interesting behavior on its visible units. These hidden units have no say in determining the *desired* behavior of the visible units—but so long as they can influence the rest of the network, they certainly have a say in determining those units' *actual* behavior. Thus they control the discrepancy between desired and actual behavior. Even with the best weights possible, then, a network can never get low error everywhere in activation space. We can still increase or decrease its error simply by changing the activations of some hidden units.

[16] The current weights specify an error surface in activation space; the idea is to minimize the volume under this surface. For practical purposes, this means minimizing total E over a lattice of points distributed evenly through activation space (under the assumption that E is continuous).

For example, suppose our error measure requires unit $i$ to always be moving toward an activation of +2. The desired direction of movement is thus determined by the current value of $a_i$. But the direction in which $a_i$ actually moves is also dependent on $net_i$. Different points in activation space may have the same value of $a_i$, but still deliver very different amounts of input to $i$. These points cannot all have low error.

The right approach is instead to train the system *dynamically*. In other words, we want to run the network as we would in practice, watch the trajectories it actually follows through activation space, and minimize error along those. Thus we only try to control the network's dynamics in the important parts of activation space.[17]

Our strategy, then, is to minimize the result of integrating $E$ over the trajectories that the system actually follows. This raises an interesting question that has not been asked before. Should we integrate by length or by time? Some thought suggests that integrating by length makes more sense.

First of all, fast trajectories should not be able to escape getting blamed for error. If error is integrated with respect to time, however, an erroneous part of the trajectory can speed up and automatically reduce its contribution to the total error.[18]

One might attempt to make a similar case against the alternative. If the learning rule integrates by length, a network that is required to make a series of complex loops in `A` could reduce its error substantially by just cutting through, i.e., taking an erroneous but short detour through that region. However, a time-integrating network does no better at learning to trace such a Gordian knot; it profits from the same shortcut. So length-integrating networks seem to have the overall advantage.

There is another, more significant advantage to integrating by length. Suppose we are training a network to settle at a particular activation vector $A$. If the network settles at some other point $A'$ instead, its failure to continue moving toward $A$ should contribute to the error for the trajectory. But how much should it contribute? If we integrate by time, the answer depends

[17]Similarly, when adjusting the second level of weights in a three-layer feedforward net, the object is not to produce the right outputs given *any* activations of the hidden units, but only for the activations that the hidden units presently assume in response to input.

[18]In some cases, we may actually want the learning rule to favor fast trajectories, but such a property should not be intrinsic to the network. The error measure is the appropriate place to specify such a preference.

on how long we let the network remain at $A$. For every additional second that elapses, the erroneousness of the system's behavior at $A'$ becomes more significant to the system (and soon overwhelms the error on the rest of the trajectory). We cannot get around this in a principled way by stopping the network as soon as it settles, because the network may stay forever in the neighborhood of $A'$ without ever quite converging to it—and if we respond with an approximate criterion to decide that the network has come "pretty close" to settling, then our error gradient is going to depend critically on the strictness of this criterion.

Integrating by length solves this problem nicely. It ensures that a trajectory generates less error per unit time as it slows down. When the network reaches $A'$ and is no longer moving, it can remain there for an instant or forever without making any difference to the computation of total error or the error gradient. Moreover, once the trajectory has gotten close to $A'$, it has generated virtually all of the error that it ever will; so we can legitimately stop the network when it is "pretty close" to settling.

So on each trajectory T, our goal is to minimize

$$\begin{aligned} \mathtt{E}_T &= \int E dL && (14) \\ &= \int E \sqrt{\sum_i (da_i)^2} && (15) \\ &= \int E \left( \sqrt{\sum_i \left(da_i/dt\right)^2} \right) dt && (16) \end{aligned}$$

(In the standard fashion, $dL$ represents an infinitesimal step along the length of a one-parameter function, $E(t)$ in this case.)

One final point. Usually we will want the network to minimize the total error for several patterns. Each pattern has a different starting point in activation space and hence a different trajectory. The network is supposed to minimize the sum of the errors on these trajectories.

As things stand, if trajectory $T_1$ is longer than trajectory $T_2$, $T_1$'s error will have extra clout. This is usually inappropriate. For example, suppose that $T_2$ begins at an equilibrium point of the system and never moves; its error will not be counted at all relative to $T_1$'s! To remove this effect, we

divide the error on each trajectory by the length of the trajectory:

$$\mathtt{E}_T = \frac{\int_T E dL}{\int_T dL}. \tag{17}$$

One can think of this as $\int E ds$, where each $s \in [0,1]$ represents a point some fraction of the distance along the trajectory; e.g., $s = 0.5$ is the halfway point.

### 3.1.4 How weight changes shift the trajectory

The previous section describes the desired learning rule as minimizing an error function along a trajectory in activation space. It is to accomplish this by changing the weights in the network. However, changing the weights will not only change the value of the error function at points along the trajectory. Changing the weights will also shift the actual *course* of the trajectory.

Might such a shift interfere with the attempt to reduce error? Not if the error function is continuous over $\mathtt{A}$. Suppose the trajectory formerly passed through a point $A \in \mathtt{A}$, but infinitesimally adjusting the weights has shifted it an infinitesimal distance, so that it now passes through $A'$. Because $E$ is continuous, the error gradient is virtually the same at $A$ and $A'$. The two gradients only differ by an infinitesimal. Hence reducing error at $A'$ requires weight changes in the same direction as reducing error at $A$. The weight changes that we made for the old trajectory are also correct for the new one.[19]

One consequence of this argument is that weight changes can be calculated independently at different points in the trajectory. Reducing error at $A_1$ may mean that we will no longer pass through $A_2$, but it doesn't affect our computation of weight changes for $A_2$.

The trajectory shifts are important nonetheless. The error measure of (17) asks for each $s \in [0,1]$: How could I change weights to reduce error $s$ of the distance along the trajectory? All we have shown is that this question asked at $s_1$ is independent of the same question asked at $s_2$. It doesn't mean

[19] In short, moving infinitesimally along a gradient doesn't really change the gradient. This happens to be part of the definition of gradient. If it weren't true, the function wouldn't be differentiable.

In practice we will have to move a little faster than infinitesimally, of course, but (according to the standard excuse) not *much* faster.

that we can answer the question at $s_1$ without considering the total effect of weight changes on the trajectory from $s = 0$ to $s = s_1$.

Indeed, changes to the trajectory are often necessary to reduce error. One reason to change weights is to ensure that the hidden units will have more helpful activations at some $s_1$. This means that we must redirect the trajectory so that at $s_1$ it is passing through a more helpful part of activation space.

So to figure out how infinitesimal weight changes will affect the network's activations at $s_1$, we have to consider the cumulative effect of those changes over the distance $[0, s_1]$. A little thought shows that this is actually possible.

For each weight $w$, we can determine the effect on the trajectory of an infinitesimal change to $w$ alone. The technique boils down to this: While simulating the net with the actual weights, we also simulate a hypothetical net where $w$ is replaced by $w+dw$. This is a straightforward way to determine the hypothetical trajectory and thus the value of $\partial \mathtt{E}/\partial w$. If we carry out the procedure for every modifiable weight $w$ in the system, we will have enough information to determine the gradient of `E` in weight space.

This is the same technique used by Williams and Zipser (1988). The resulting algorithm will be computationally intensive. It essentially requires us to run $n$ simultaneous copies of the network, where $n$ is the number of modifiable weights.

Unfortunately, such a method seems necessary to solve the general problem. We could rewrite it somewhat to reduce the actual requirements on the network. For instance, the learning algorithm could repeatedly choose a weight at random, make some tentative modification to it, and later compute a permanent modification based on the observed change in error. Such a learning algorithm would not require any storage space of its own; but it would learn more slowly and be no more elegant. The fundamental difficulty of the problem would be unchanged.

## 3.2 Formal derivation of the general model

This section gives the technical details of the algorithm, and may be skipped by the general reader.

### 3.2.1 Notation

Some additional notation will be necessary.

$\mathtt{A}$ denotes activation space, $\mathtt{A}'$ its differential space.

$\mathtt{W}$ denotes weight space. $\mathtt{T}$ denotes the timeline [0, +infinity).

We use conventional variables $A \in \mathtt{A}$, $D \in \mathtt{A}'$, $W \in \mathtt{W}$, $t \in \mathtt{T}$. Units in the network are enumerated, and the integer variables $i$, $j$, $k$, and $l$ stand for particular units.

The components of a vector $A$ are represented by $a_1, a_2, a_3 \ldots$. The same convention is used for $D$. $W$ is a square matrix; the component $w_{ij}$ represents the weight to unit $j$ from unit $i$, and is considered to be 0 if no connection exists.

The following functions are assumed to be continuous and differentiable on all their arguments, except as noted.

$\mathbf{A} : \mathtt{W} \times \mathtt{A} \times \mathtt{T} \rightarrow \mathtt{A}$ describes possible trajectories for a net of fixed topology. We write $\mathbf{A}_{W,A_0}(t)$ to denote the position that the system will arrive at when given weight vector $W$, initialized at $A_0$, and allowed to run for a time interval $t$. Generally we leave the subscripts off, so that $\mathbf{A}(t)$ denotes the activation vector of a particular network at time $t$, assuming some particular vector of starting activations.

$\mathbf{D} : \mathtt{W} \times \mathtt{A} \rightarrow \mathtt{A}'$ describes the dynamics of the net. We write $\mathbf{D}_W(A)$ for $\lim_{t \rightarrow 0} \frac{\mathbf{A}_{W,A}(t) - \mathbf{A}_{W,A}(0)}{t}$, i.e., for $\mathbf{A}_{W,A}{}'(0)$. $\mathbf{D}_W(A)$ need not be differentiable with respect to $A$, though it must be continuous.

$\mathbf{E} : \mathtt{T} \times \mathtt{A} \times \mathtt{A}' \rightarrow \mathtt{R}^+$ describes the error of the system at a particular time, given its position and direction in activation space. We usually abbreviate $\mathbf{E}_t(\mathbf{A}_{W,A_0}(t), \mathbf{D}_W(\mathbf{A}_{W,A_0}(t)))$ simply as $\mathbf{E}(t)$, since $W$ and $A_0$ are usually clear from context. $\mathbf{E}$ may depend on $t$, even discontinuously.

The derivation below will sometimes use expressions like

$$\frac{\partial f(\hat{x}, x^3)}{\partial x} \text{ and } \frac{\partial f(x, \widehat{x^3})}{\partial x^3}.$$

These stand respectively for the first and second partials of $f$, evaluated at the point $(x, x^3)$. The peaked hat on top indicates which argument is the variable. This notation is a little easier to follow than the standard

$$\left.\frac{\partial f(a,b)}{\partial a}\right|_{(x,x^3)} \text{ and } \left.\frac{\partial f(a,b)}{\partial b}\right|_{(x,x^3)}.$$

In the same style, we may write $\partial f(\widehat{x}, x^3)/\partial t$ to stand for

$$\left.\frac{\partial f(a,b)}{\partial a}\right|_{(x,x^3)} \cdot \frac{dx}{dt}.$$

The idea is simply to describe how $f$ changes over $t$ when only its first argument is allowed to vary.

### 3.2.2 Calculating the gradient in weight space

We want an expression for $\partial \mathbf{E}_{W,A_0}(t)/\partial W$. We can find this by expressing $W$ in terms of its basis vectors, so we only need to find $\partial \mathbf{E}_{W,A_0}(t)/\partial w_{ij}$ for each weight $w_{ij}$.

$$\frac{\partial \mathbf{E}_{W,A_0}(t)}{\partial w_{ij}} = \frac{\partial \mathbf{E}(\mathbf{A}_{W,A_0}(t), \mathbf{D}_W(\mathbf{A}_{W,A_0}(t)))}{\partial w_{ij}} \tag{18}$$

$$= \frac{\partial \mathbf{E}(\widehat{\mathbf{A}(t)}, \mathbf{D}(t))}{\partial \mathbf{A}(t)} \cdot \frac{\partial \mathbf{A}(t)}{\partial w_{ij}} + \frac{\partial \mathbf{E}(\mathbf{A}(t), \widehat{\mathbf{D}(t)})}{\partial \mathbf{D}(t)} \cdot \frac{\partial \mathbf{D}(t)}{\partial w_{ij}} \tag{19}$$

Note that the multiplicands are vectors, and their dot product a scalar.

The first term of each product is simply a partial derivative of $\mathbf{E}$, and can be found directly from $\mathbf{E}$'s definition (whatever it is). The interesting terms are $\partial \mathbf{A}(t)/\partial w_{ij}$ and $\partial \mathbf{D}(t)/\partial w_{ij}$. To derive them, we must specify $\mathbf{A}$ and $\mathbf{D}$ for the particular network architecture we're using.

We define $\mathbf{D}_W(A)$ as follows:

$$d_i = \begin{cases} net_i - decay_i & (\text{where } net_i(t) = \sum_j f(a_j) w_{ij}), \\ & \text{or 0 if } i \text{ is clamped.} \end{cases} \tag{20}$$

Now we define $\mathbf{A}_{W,A_0}(t_1)$ in terms of $\mathbf{D}$:

$$\mathbf{A}_{W,A_0}(t_1) = A_0 + \int_{t=0}^{t_1} \mathbf{D}_W(\mathbf{A}_{W,A_0}(t))dt. \tag{21}$$

In the case where time is discrete, this is approximated by

$$\mathbf{A}_{W,A_0}(t_1) = A_0 + \sum_{(t/\eta)=0}^{(t_1/\eta)-1} \eta \mathbf{D}(\mathbf{A}_{W,A_0}(t)) \tag{22}$$

$$= \mathbf{A}_{W,A_0}(t_1 - \eta) + \eta \mathbf{D}(\mathbf{A}_{W,A_0}(t_1 - \eta)) \text{ with } \mathbf{A}_{W,A_0}(0) = A_0 \tag{23}$$

which is exactly how we simulate the net iteratively.

In order to continue, we will also have to find the partial derivatives of $\mathbf{D}_W(A)$.

$$\frac{\partial \mathbf{D}_{\widehat{W}}(A)}{\partial w_{ij}} \text{ is given by } \frac{\partial d_k(A)}{\partial w_{ij}} = \frac{\partial net_k}{\partial w_{ij}} \tag{24}$$

$$= \frac{\partial}{\partial w_{ij}} \sum_l f(a_l) w_{kl} \tag{25}$$

$$= \sum_l f(a_l) \frac{\partial w_{kl}}{\partial w_{ij}} \tag{26}$$

$$= f(a_j)\delta_{ik}. \tag{27}$$

$$\frac{\partial \mathbf{D}_W(\widehat{A})}{\partial a_l} \text{ is given by } \frac{\partial d_k(\widehat{A})}{\partial a_l} = \frac{\partial}{\partial a_l} net_k - \frac{\partial}{\partial a_l} decay(a_k) \tag{28}$$

$$= f'(a_l) w_{kl} - \delta_{kl} decay'(a_l). \tag{29}$$

Now we have all the terms necessary to find the effect of individual weight changes on $\mathbf{D}$ and $\mathbf{A}$. The two equations are written recursively in terms of each other.

$$\frac{\partial \mathbf{D}_W(A(t))}{\partial w_{ij}} = \frac{\partial \mathbf{D}_{\widehat{W}}(A(t))}{\partial w_{ij}} + \frac{\partial \mathbf{D}_W(\widehat{A(t)})}{\partial A(t)} \cdot \frac{\partial A(t)}{\partial w_{ij}} \tag{30}$$

$$\text{i.e., } \frac{\partial d_k}{\partial w_{ij}} = \frac{\partial d_k(A)}{\partial w_{ij}} + \sum_l \frac{\partial d_k(\widehat{A})}{\partial a_l} \cdot \frac{\partial a_l(t)}{\partial w_{ij}} \tag{31}$$

$$\frac{\partial \mathbf{A}_{W,A_0}(t_1)}{\partial w_{ij}} = \begin{cases} \int_{t=0}^{t_1} \frac{\partial \mathbf{D}_W(\mathbf{A}_{W,A_0}(t))}{\partial w_{ij}} dt & \text{(continuous case)} \\ \frac{\partial \mathbf{A}_{W,A_0}(t_1-\eta)}{\partial w_{ij}} + \eta \frac{\partial \mathbf{D}_W(\mathbf{A}_{W,A_0}(t_1-\eta))}{\partial w_{ij}} dt & \text{(discrete case)} \\ & \text{with } \frac{\partial \mathbf{A}_{W,A_0}(0)}{\partial w_{ij}} = \frac{\partial A_0}{\partial w_{ij}} = 0. \end{cases} \tag{32}$$

### 3.2.3 An algorithm

The above equations, together with (14) and (17), lead directly to the following learning algorithm. For a network with a given set of weights $W$, this algorithm describes how to change the weights so as to reduce the error along the trajectory that starts at $A_0 \in \mathtt{A}$.

1. Start the trajectory at $A_0$: Set $\mathbf{A}(0) = A_0$ and $\partial\mathbf{A}(0)/\partial w_{ij} = 0$ (for each weight $w_{ij}$).

2. For each sampled time $t$, where $t$ takes on the values $t_0 = 0, t_1 = t_0 + \Delta t_0, t_2 = t_1 + \Delta t_1, \ldots$ :

   (a) Compute $\mathbf{D}(\mathbf{A}(t))$ and $\partial\mathbf{D}(\mathbf{A}(t))/\partial w_{ij}$ from $\mathbf{A}(t)$, $\partial\mathbf{A}(t)/\partial w_{ij}$, and $W$. This determines the network's direction from its current position, and its hypothetical direction (under a small weight change) from its hypothetical position.

   (b) Using the definition of $\mathbf{E}$, compute the partials of $\mathbf{E}(\mathbf{A}(t), \mathbf{D}(\mathbf{A}_{W,A_0}(t)))$ with respect to $\mathbf{A}(t)$ and $\mathbf{D}(\mathbf{A}_{W,A_0}(t))$. These numbers indicate how $\mathbf{E}$ would change if the system's position in `A` changed but not its dynamics, or vice-versa.

   (c) From the quantities mentioned in the above two steps, calculate the partials of $\mathbf{E}$ with respect to each $w_{ij}$. This gives the gradient of $\mathbf{E}$ in `W` at time $t$.

   (d) Calculate the distance $\Delta L = (\Delta t)\sqrt{\sum_i d_i^2}$ that the network will move in `A` over the next time step, where $\Delta t$ is the expected duration of that time step.

   (e) Accumulate weight changes against the gradient of $\mathbf{E}$, in proportion to the learning rate and the instantaneous distance $\Delta L$.

   (f) Find $\mathbf{A}(t+\Delta t)$ from $\mathbf{A}(t)$ and $\mathbf{D}(\mathbf{A}(t))$; and find $\partial\mathbf{A}(t+\Delta t)/\partial w_{ij}$ from $\partial\mathbf{A}(t)/\partial w_{ij}$ and $\partial\mathbf{D}(\mathbf{A}(t))/\partial w_{ij}$. That is, determine where the network will be (and would be with different weights) on the next time step.

3. When the trajectory is complete, divide the accumulated weight changes by the total length of the trajectory and institute them. (Depending on the problem, we may call the trajectory complete when it has settled, used up more than its allotted time or distance, run out of input, finished producing its output, accumulated too much error, etc.)

So long as all the $\Delta t$'s are small,[20] there is no reason that they must be the same for every time step. For example, when processing $t_i$, we might

[20] In fact, the learning algorithm will still work with large values of $\Delta t$, so long as they are consistent from one run of the trajectory to the next. However, using large values makes

choose $\Delta t_i$ so that $\Delta L$, the distance the network travels on the following time step, is a constant. Later we will also see discontinuous error measures that require irregularly spaced time steps.

## 3.3 Summary of the general model

The above learning algorithm is more powerful than any in the literature to date. The strongest previous algorithm was only able to prescribe a target activation at every point in time (Williams & Zipser, 1988). The error measure of this algorithm, by contrast, can consider any or all of three quantities: the network's position $\mathbf{A}(t)$ in activation space, its instantaneous velocity vector $\mathbf{D}(t)$, and the current time $t$. This permits it to explicitly prescribe certain kinds of behaviors that have previously been neglected; in particular, time-dependent and position-dependent dynamics. In light of the discussion in section 2, these capabilities could be quite useful.

The learning algorithm performs gradient descent on an error surface by changing the network weights. For each weight $w$, the algorithm must determine the effect on the whole trajectory of a small change to $w$. It does so by gradually accumulating the small effects of such a change. As it figures out how a small change to $w$ would change the trajectory, the algorithm also calculates the error for the changed trajectory. This tells it whether increasing $w$ would increase or decrease error, and thus which way it should change $w$.

It is important that the formulation given above does not commit itself to any particular error measure. Any real-valued continuous function on $\mathtt{T} \times \mathtt{A} \times \mathtt{A}'$ will do. Together with the new ability to control dynamics, this gives one an enormous amount of flexibility in the kinds of behavior one can require of a network.

---

the network a different kind of computational device. For example, recurrent networks with large time steps are less likely to settle (mutually inhibitory units will oscillate, etc.). Section 2.1.1 outlines some other reasons to be interested in networks with small or infinitesimal time steps.

# 4 Particular Models

The general model described above permits us to define our network's error virtually any way we care to. Now we discuss some actual error measures that both demonstrate this flexibility and are useful for particular learning problems.

## 4.1 Some models of potential interest

Let us begin by outlining a number of interesting error measures. Only a few will be developed in full in this paper; but the discussion below should illustrate the generality of the overall approach. With appropriate error measures, it seems, this architecture can train a number of important behaviors, some of which have already been studied individually.

1. *Williams and Zipser problems.* The net's activation is prescribed at every time $t$. If $\bar{\mathbf{A}}(t)$ is the desired trajectory through $\mathtt{A}$, the error at $t$ can be defined as the square of the Euclidean distance between $\mathbf{A}(t)$ and $\bar{\mathbf{A}}(t)$.[21] When input to the system is provided by clamping nodes, this error measure yields the model derived in Williams and Zipser (1988).[22]

2. *Generalized Jordan problems.* The net's activation is prescribed only at certain moments $t_1, t_2, \ldots$. In between, it is free to take whatever path is easiest for it. $\mathbf{E}$ here is discontinuous. At the $t_i$ it is computed as in number 1; it is 0 the rest of the time. The system's actual trajectory will be continuous, of course.

   Note that when simulating a network for this problem, we must be careful to include the important moments $t_1, t_2, \ldots$ among our time

[21] For simplicity, we will often refer to $\bar{\mathbf{A}}(t)$ and other prescribed activations as points in $\mathtt{A}$. In actuality they usually happen to be hyperplanes; i.e., they do not prescribe individual activations for every unit of the system. Thus we are really talking at the moment about the distance from a point to a hyperplane. Later we will talk about training the trajectory to pass through points, which really means training it to pass through hyperplanes.

[22] The two models are almost identical. Williams and Zipser's is slightly different in that it always has evenly spaced time steps and integrates error by time, not by length. Also, their model uses dedicated input lines (which our model's clamped nodes can easily simulate, in the style of the IA model discussed earlier).

samples. For the similar problems considered by Jordan (1986), the $t_i$ were evenly spaced and constituted the only time samples for his (discrete-time) network.

3. *Mapping problems.* For a large class of problems, we may want the network's activation to settle in the long run to some target activation $\bar{\mathbf{A}}$. We only care that the network reaches this target; we are indifferent as to the path it takes.

   There are several ways to ensure that the network reaches the target. We may simply take the approach of number 2 and require that the network be at or near the target after some reasonable period of time. (We also need to require that the network is not moving from the target.) Alternatively, we may ask that each target node be constantly moving toward the target, or that the network as a whole moves closer in activation space to the target.

4. *Repeated mapping problems.* Another important class of tasks may be described as *repeated mapping problems.* If we ask a network to do many mapping problems in succession, it may be able to exploit regularities in the order of the problems it is given. Consider a network that is to transcribe disconnected speech. Each word is a separate mapping problem: from the phonemes, the network must derive a written representation. However, the previous words in the network's input can help it decode the current word.

   We can take this situation to extremes. The cumulative XOR problem, mentioned earlier (1.1.2), simply cannot be solved as a sequence of individual mappings. It *requires* that the network pay attention to the previous mappings.

5. *Constraint satisfaction.* A constraint satisfaction or pattern completion network just solves a special kind of mapping problem. The desired mappings have a special consistency. For example, if pattern $A \cup P$ maps to pattern $Q$, then $A \cup Q$ can legitimately map to $P$, insofar as the network is capable of implementing both mappings.

   We can reuse the error measures of number 3, which are suited for general mapping problems. We might also arrange error measures that

take advantage of the redundancy in the mappings. For example, perhaps the network will learn faster if while we train it to bring free nodes to their correct activations, we simultaneously train it to deliver subsistence inputs to the clamped nodes. This will make it a little easier to train the clamped nodes when they are the free nodes of some other pattern. Essentially we are training on two patterns at once.

We can make this technique even more useful by releasing the clamped nodes. After a short period of time, when the free nodes have had a chance to approach their targets, we can release the clamped nodes. If we continue to train the network, asking all the nodes to move to their targets, we are extending the basin around the target point in activation space.

6. *Released mappings.* In general, it is possible to do mapping problems without clamped nodes (using the same error measures). We can require that a system, when released at a particular point $A$ of activation space, moves to its associated point $A'$ and stays there.

   Such a scheme is especially well-suited to perform transformations on input. Consider the case where $A$ represents *ring* + *PAST* phonetically, and $A'$ represents *rang* + $\phi$. A released activation network can learn to automatically transform one to the other. Like the past-tense network of Rumelhart and McClelland (1986), such a network might be able to generalize from *ring/rang* to *sing/sang* and other similar pairs. Moreover, it has at least one major advantage over the direct mapping approach of that network. It manages to explain why the mapping *sing/sang*, which preserves most of the phonetic information, is easier to learn than one like *sing/flka*. This is a point on which Pinker and Prince (1988) severely criticize the Rumelhart and McClelland model.

7. *Position-independent dynamics.* The flip side of number 1 is to prescribe, not the network's activation at each time $t$, but its direction $\bar{\mathbf{D}}(t)$. The error measure at time $t$ is then the square of the distance between $\mathbf{D}(t)$ and $\bar{\mathbf{D}}(t)$. This is not very different from number 1. Telling the network what path to follow is the same problem, after all, regardless of whether the path is specified in terms of its position over time or its direction over time.

However, the present error measure successfully abstracts the idea of a trajectory's shape. If we want the network to generate a local pulse in $\mathtt{A}$ regardless of whether it is released from $A_0$, $A_1$, or some other (possibly unanticipated) characteristic starting activation, this error measure is the natural one to use. Moreover, it forgives different kinds of error than the measure of number 1 does. If it is easiest for the network to move a bit to the right as it begins to generate its pulse, this error measure doesn't mind much, whereas the other considers all the points along the shifted pulse to be "wrong," and may try hard to correct it at the expense of other desirable trajectories elsewhere in the space.

8. *Time-independent dynamics.* We may wish to induce certain dynamical properties in the network, such as limit cycles. The error measure in this case depends on $\mathbf{A}$ and $\mathbf{D}$ only. It is given at time $t$ by the Euclidean distance of $\mathbf{D}(\mathbf{A}(t))$ from its desired value, $\bar{\mathbf{D}}(\mathbf{A}(t))$. Alternatively, we can work up error measures that put less exact constraints on $\mathbf{D}(t)$. We might just require that the $\mathbf{D}(t)$ fall in some particular hyperquadrant of $\mathtt{A}'$. This means that all the activations are moving in particular directions, at whatever speeds.

9. *Gradient descent.* A special case of number 8 is gradient descent. The learning rule already performs gradient descent in activation space on $\mathbf{E}$. But we can actually train the network to perform gradient descent in activation space on some other measure $\mathbf{G}$. That is, let $\mathbf{G}$ be a differentiable function from $\mathtt{A}$ to $\mathtt{R}$. At any point $A \in \mathtt{A}$, we want to make the net's direction of movement $\mathbf{D}(A)$ proportional to that prescribed by the gradient of $-\mathbf{G}$ at $A$.

   A system that has learned these dynamics correctly will have attractors at the local minima of $G$ (and nowhere else). We will see later that this technique can be used to implement a content-addressable memory, which is a particular kind of constraint-satisfaction device.

## 4.2 Some topologies of potential interest

For a given task, a learning rule's success may depend on the topology of the network that is trying to learn the task. Section 2.1.3 has already explored

in detail the potential of small recurrent clusters. Here we mention two other topological properties.

Symmetric weights are useful for constraint satisfaction problems. In fact, Boltzmann machines (Hinton & Sejnowski, 1986) use them exclusively. They are well suited to such problems, because they capture the idea of a correlation or anticorrelation between two units.

Symmetric weights are very easy to implement. To implement the weight $w$ connecting $i$ and $j$, we can simply regard it as two separate but equal weights, $w_{ij}$ and $w_{ji}$. Each has its own effect on the error, so that $\partial E/\partial w = \partial E/\partial w_{ij} + \partial E/\partial w_{ji}$. In order to make a weight change to $w$, we change it along the error gradient for $w_{ij}$, and then along the error gradient for $w_{ji}$.

Sigma-pi (multiplicative) connections are also useful for constraint satisfaction. Sigma-pi connections make it possible to solve the symmetric XOR problem with no hidden units at all. The solution is very simple: each unit, when active, changes the connection between the other two units from mutually excitatory to mutually inhibitory. This exactly captures the meaning of symmetric XOR.

In general, the use of both symmetric and sigma-pi connections may be very helpful. When a gated symmetric connection links units $i$ and $j$, the rest of the network is determining the degree of correlation between the two. This is a sensible paradigm for constraint satisfaction.

The derivation of a learning rule for sigma-pi units is relatively straightforward. The only changes are to the expressions for $net_i$, $\mathbf{D}(t)$, and the partials of $\mathbf{D}(t)$. The formulas are mildly messy, however, and not really worth including here.

## 4.3 Detailed derivation of particular error measures

### 4.3.1 Mapping model I: Nodes toward targets

Section 2.2.5 observed that tasks requiring a network to settle to particular values are really requiring certain dynamics of the network. The network will settle to a point $A \in \mathtt{A}$ if and only if it has established $A$ as an attractor whose basin includes the network's starting point.

There are several types of error measure that might encourage the network to establish such attractors. Since we are interested in exploring the general model's ability to consider network dynamics, however, we will choose one

that confines itself to prescribing dynamics. Specifically, we will require that each node move toward its current target at all times.

This is a strong condition, in that it requires good behavior from every target node. However, it has the advantage of being local, and therefore easy to compute.[23] It is also quite lenient with respect to the nodes' exact behavior. Each node must be moving toward its target with some minimum speed, but otherwise is free to travel as quickly or slowly as is convenient.

Let $\epsilon > 0$ be the required minimum speed of a node $i$ toward its target. Then we want the node's local error $e_i$ to be 0 when its actual speed $d_i$ or $-d_i$ exceeds $\epsilon$, and otherwise the amount by which it falls short of $\epsilon$. (This could be substantial if it is moving in the wrong direction.) We set the overall error, $E$, to $\sum_i e_i^2$.

This is the ideal measure, but it is not continuous at 0, or differentiable when the node is moving at exactly the minimum speed. We can fix it up as follows. Instead of using a constant $\epsilon$, let $\epsilon = \bar{\epsilon}|a_i - t_i|^n$, where $n > 1$ and $\bar{\epsilon}$ is a constant. This move makes $E$ continuous and differentiable at 0. It means that error gets more serious when the node is far from the target (and substantially more serious for large $n$).

Now in the case where $a_i$ is less than or equal to its target value $t_i$, we determine $e_i$ as

$$e_i = \begin{cases} -d_i + \epsilon & \text{if } d_i < 0, \\ 0 & \text{if } d_i > \epsilon, \text{ or} \\ (d_i - \epsilon)^3/\epsilon^2 + 2(d_i - \epsilon)^2/\epsilon & \text{if } d_i \in [0, \epsilon]. \end{cases} \tag{33}$$

The derivatives are straightforward to find. If $a_i > t_i$, the only difference is that instead of wanting $d_i > \epsilon$, we want $-d_i > \epsilon$; so we simply substitute $-d_i$ for $d_i$. (It is unnecessary to also reverse the sign of $e_i$, since it will be squared anyway.)

When integrating this measure over time, one should realize that it may be impossible for all units to move toward their targets from the very beginning. The hidden units may have to "charge up" before the system starts moving in the right direction. One way to take this into account is to multiply E at every step by a factor like $(1 - e^{-t/\tau})$, where $\tau$ is a constant. This is called a *soft start*.

[23] There is certainly no philosophical advantage to a local error measure in this very nonlocal algorithm. However, if a local method for training dynamic behavior is discovered in future, such a measure might indeed be desirable.

### 4.3.2 Mapping model II: System toward target

We can define a similar but nonlocal error measure that makes fewer demands on the individual units. We simply ask the system to continually reduce its Euclidean distance toward the target.

In actual fact, it is easiest to have it reduce the square of that distance. Let $z_i = a_i - t_i$ for all units $i$ with target values. Then the square of the distance to the target is $Z = \sum_i z_i^2$ (summing over these units). That value is changing with respect to time at a rate of

$$\frac{dZ}{dt} = 2\sum_i z_i \frac{dz_i}{dt} = 2\sum_i z_i \frac{da_i}{dt} = 2\sum_i z_i d_i. \tag{34}$$

Let us require this value to be negative, and less than some quantity $-2\epsilon$. Then we can define our error as

$$E = \max(0, x) \tag{35}$$

where

$$x = \epsilon + \sum_i z_i d_i \tag{36}$$

To make this continuous, we actually use $E = xh(x)$, where x is a sharp logistic function.

The partial derivatives of this error measure are given by

$$\begin{aligned}
dE &= dx\,h(x) + x h'(x) dx && (37)\\
&= dx\,(h(x) + x h'(x)) && (38)\\
\frac{\partial E}{\partial a_i} &= \frac{\partial x}{\partial a_i}(h(x) + x h'(x)) && (39)\\
&= d_i (h(x) + x h'(x)) && (40)\\
\frac{\partial E}{\partial d_i} &= \frac{\partial x}{\partial d_i}(h(x) + x h'(x)) && (41)\\
&= z_i (h(x) + x h'(x)) && (42)
\end{aligned}$$

This measure has the apparent advantage that it will permit some units to move slightly away from their targets in order for the system as a whole to get closer to its target in A. In other words, the system's possible trajectories are somewhat less constrained. There is a geometrical interpretation of this

fact. Instead of having to approach the target by penetrating the corners of successively smaller cubes centered at the target, as with the previous error measure, the system only has to move inside successively smaller spheres.

### 4.3.3 General gradient-descent model

The general gradient-descent model is very simple. Let **G** be an energy function on A. The system's dynamics are to ensure that the system will follow the gradient of $G$ to a local minimum. In other words, we want $\mathbf{D}(A) = -\xi \text{Gradient}_A G$, for some $\xi > 0$.

Component-wise, for a given A we require that $d_i = \bar{d}_i$, where

$$\bar{d}_i = -\xi \frac{\partial G(A)}{\partial a_i}. \tag{43}$$

For this or any case where the learning rule prescribes a particular dynamical system over activation space, we can define **E** and find its derivatives as follows.

$$\begin{aligned}
\mathbf{E}(A, D) &= \frac{1}{2} \sum_i (e_i(A, D))^2 && (44)\\
\text{where } e_i(A, D) &= d_i(A) - \bar{d}_i(A) && (45)\\
\partial \mathbf{E}(A, D) &= \sum_i e_i(A, D) \partial e_i(A, D) && (46)\\
\frac{\partial e_i(\widehat{A}, D)}{\partial a_j} &= -\delta_{ij} \frac{\partial \bar{d}_i(A)}{\partial a_j} && (47)\\
\frac{\partial e_i(A, \widehat{D})}{\partial d_j} &= \delta_{ij}. && (48)
\end{aligned}$$

Hence

$$\begin{aligned}
\frac{\partial \mathbf{E}(\widehat{A}, D)}{\partial a_j} &= e_j(A, D) \cdot \left( -\frac{\partial \bar{d}_j(A)}{\partial a_j} \right) && (49)\\
\frac{\partial \mathbf{E}(A, \widehat{D})}{\partial d_j} &= e_j(A, D). && (50)
\end{aligned}$$

To teach a network to do gradient descent on $G$, then, we only need to specify expressions for $\bar{d}_i = -\xi \partial G(A)/\partial a_i$ (equation 43) and $\partial \bar{d}_i(A)/\partial a_i$ (equation 49).

### 4.3.4 Content-addressable memory model

With an appropriate definition of $G$, training the network to do gradient descent can get it to act as a content-addressable memory. This is a form of constraint satisfaction. The network is to have a number of "memories," represented by particular points in $\mathtt{A}$. If the network is released anywhere in activation space, it should end up settling at one of these memories.

The network should tend to converge to memories with activations similar to its starting point. However, a relatively "strong" memory should be able to attract it from farther away.

For each pattern $p \in \mathtt{A}$ that serves as a memory, we define

$$g_p(A) = \frac{1}{2}\sum_i (a_i - p_i)^2. \tag{51}$$

This defines a large bowl centered on $p$ with $g_p(p) = 0$ as the only minimum.

Now let

$$G(A) = \prod_p g_p(A)^{s_p}, \tag{52}$$

where $s_p > 0$ measures the strength of pattern $p$, and where $\sum_p s_p = 1$.

The surface $G$ has its minima at the patterns $p$. $G(p) = 0$ for each $p$, and $G(p) > 0$ in the immediate neighborhood of $p$. $s_p$ shapes the surface depressions surrounding each pattern. If $s_p$ is very small, then the function ${g_p}^{s_p}$ is close to 1 everywhere, flat except for a deep pockmark immediately surrounding $p$, where its value decreases to 0. If $s_p$ is comparatively large, on the other hand, ${g_p}^{s_p}$ describes a wide bowl the way that $g_p$ does.

Multiplying all the terms together yields a continuous surface, with zeroes at all the memories, sides sloping down towards the strong memories, and pockmarks at the weak ones. Gradient descent on $G$ will settle at memories according to the conditions described earlier in this section.[24]

This measure $G$ rolls all the patterns together in a complex way. We want to train the network to follow the gradient of $G$. The surprising result of this section is that there is a seemingly practical algorithm to do this. It turns

[24] One minor exception. It may happen that, as the network is descending toward a strong memory, it passes through the attractor basin of some weak memory. In this case, the network will end up settling at a weak memory that wasn't close to its starting point. This case will rarely occur in practice, however, since weak memories have small attractor basins.

out that the network can learn the correct behavior by separate study of the individual patterns, two at a time. Specifically, the network needs to move in weight space so as to better follow $G$ at some point $A$—and its required direction of movement in $\mathtt{W}$ can be given as the sum of many small direction vectors, each term determined by an individual pair of patterns.

For the network to learn to follow the gradient, the learning rule of section 4.3.3 needs to know what the gradient actually is. So we must compute $\partial G(A)/\partial a_i$.

First of all, we note that

$$\frac{\partial g_p(A)}{\partial a_i} = \frac{\partial}{\partial a_i}\frac{1}{2}(a_i - p_i)^2 = a_i - p_i. \tag{53}$$

If $G(A) = 0$, then there is some pattern $q$ for which $g_q(A) = 0$. It follows that $a_i = q_i$, and that $\partial g_q(A)/\partial a_i = 0$ (from (53)). Moreover, $\partial g_q(A)^{s_q}/\partial a_i = s_q g_q(A)^{s_q - 1}(\partial g_q(A)/\partial a_i) = 0$. Then we see that

$$\begin{aligned}
\frac{\partial G(A)}{\partial a_i} &= \frac{\partial}{\partial a_i}\left(g_q(A)^{s_q}\prod_{p\neq q} g_p(A)^{s_p}\right) &(54)\\
&= \left(\frac{\partial}{\partial a_i}g_q(A)^{s_q}\right)\prod_{p\neq q} g_p(A)^{s_p} + g_q(A)^{s_q}\left(\frac{\partial}{\partial a_i}\prod_{p\neq q} g_p(A)^{s_p}\right) &(55)\\
&= 0 + 0 = 0. &(56)
\end{aligned}$$

If $G(A) \neq 0$, on the other hand, we are permitted to factor out a $G(A)$ term and write

$$\begin{aligned}
\frac{\partial G(A)}{\partial a_i} &= G(A)\left(\sum_p \frac{\partial(g_p(A)^{s_p})/\partial a_i}{g_p(A)^{s_p}}\right) &(57)\\
&= G(A)\sum_p \frac{s_p g_p(A)^{s_p-1}(\partial g_p(A)/\partial a_i)}{g_p(A)^{s_p}} &(58)\\
&= G(A)\sum_p \frac{s_p(\partial g_p(A)/\partial a_i)}{g_p(A)} &(59)\\
&= G(A)\sum_p s_p\frac{a_i - p_i}{g_p(A)}. &(60)
\end{aligned}$$

Now we can define the quantities required by section 4.3.3. We want $\bar{d}_i$ to be proportional to $\partial G(A)/\partial a_i$. We are actually going to set it proportional to

$(\partial G(A)/\partial a_i)/G(A)$. This amounts to the same thing, since $G(A)$ is positive and constant at $A$. $\bar{\mathbf{D}} =< \bar{d}_1, \bar{d}_2, \ldots >$ will still point in the same direction as the negative gradient, although it will be scaled by a factor of $\xi G(A)$.

$$\bar{d}_i = -\xi \sum_p s_p \frac{a_i - p_i}{g_p(A)}, \text{ or 0 if some } g_p(A) = 0 \tag{61}$$

$$\frac{\partial \bar{d}_i}{\partial a_i} = -\xi \sum_p s_p \frac{(a_i - p_i)^2 - g_p(A)}{g_p(A)^2}, \text{ or 0 if some } g_p(A) = 0. \tag{62}$$

Now we go ahead and derive the expression for the error gradient of the network. From equations (49–50), and taking advantage of the fact that $\sum_p s_p = 1$, we deduce that

$$\frac{\partial \mathbf{E}}{\partial d_i} = e_i = d_i - \bar{d}_i \tag{63}$$

$$= d_i + \xi \sum_p s_p \frac{a_i - p_i}{g_p(A)} \tag{64}$$

$$= \sum_p s_p \xi \left(\frac{d_i}{\xi} + \frac{a_i - p_i}{g_p(A)}\right) \tag{65}$$

$$\frac{\partial \mathbf{E}}{\partial a_i} = e_i \left(-\frac{\partial \bar{d}_i}{\partial a_i}\right) \tag{66}$$

$$= \left(\sum_p s_p \xi \left(\frac{d_i}{\xi} + \frac{a_i - p_i}{g_p(A)}\right)\right)\left(\sum_q s_q \xi \frac{(a_i - q_i)^2 - g_q(A)}{g_q(A)^2}\right) \tag{67}$$

$$= \sum_p \sum_q \left[ s_p s_q \xi^2 \left(\frac{d_i}{\xi} + \frac{a_i - p_i}{g_p(A)}\right)\left(\frac{(a_i - q_i)^2 - g_q(A)}{g_q(A)^2}\right)\right] \tag{68}$$

Our weight changes are prescribed in terms of (63–68) by the usual rule,

$$\Delta w_{ij} = -\mu \frac{\partial \mathbf{E}}{\partial w_{ij}} = -\mu \sum_i \left(\frac{\partial \mathbf{E}}{\partial a_i}\frac{\partial a_i}{\partial w_{ij}} + \frac{\partial \mathbf{E}}{\partial d_i}\frac{\partial d_i}{\partial w_{ij}}\right).$$

In practice, we need not compute the full summations of equations (63–68) above. The following approach suffices. When training the network on

a given trial, we simply pick two patterns, $p$ and $q$. We make our weight changes under the pretense that

$$\frac{\partial \mathbf{E}}{\partial d_i} = \xi(\frac{d_i}{\xi} + \frac{a_i - p_i}{g_p(A)}) \tag{69}$$

$$\frac{\partial \mathbf{E}}{\partial a_i} = \xi^2(\frac{d_i}{\xi} + \frac{a_i - p_i}{g_p(A)})(\frac{(a_i - q_i)^2 - g_q(A)}{g_q(A)^2}) \tag{70}$$

If the choices of $p$ and $q$ are independent, and each pattern $p_0$ is chosen $s_{p_0}$ of the time, then the average per-trial weight changes match those prescribed by (63) and (66).

This is a nice result. It means that the network will be trained to follow the gradient of $G(A) = \prod_p g_p(A)^{f_p}$, where $f_p$ represents the frequency with which the network sees pattern $p$. In other words, the frequency with which the pattern is presented will exactly equal the strength of the pattern.

Note that $\xi$ controls the speed at which the trained network is expected to trace the gradient of $G$. If we are willing to use a small value of $\xi$, the term $\partial \mathbf{E}/\partial a_i$ becomes insignificant. This is the only term that involves the pattern $q$. For small $\xi$, we can safely ignore that term and still make approximately the correct weight changes.

That is, suppose there are $n$ training patterns. A sufficiently small $\xi$ can free us from having to show the network all $n^2$ possible pattern pairs $(p, q)$ on each training epoch. In effect, by thus ignoring the $\partial E/\partial a_i$ factor, we are asking the system to simply achieve a better gradient in the region of its trajectory, and not worry about whether its trajectory also shifts in such a way as to decrease error.

# 5 Simulation Results

The error measures developed above were subsequently tested on various forms of the XOR problem. XOR problems were deliberately chosen so as to make things difficult for the learning rules. The traditional sum-of-squares expression (i.e., sum of distances squared) is a very straightforward error measure. If we are to replace it with any of the less obvious measures developed above, we ought to make sure that the replacements are able to achieve good performance on difficult problems.

XOR is a good test for two reasons. First, it cannot be solved unless the network develops specialized hidden units. Second, gradient descent solutions to XOR usually spend a tremendous amount of time in a saddle point of the error surface before they solve the problem. They move quickly into a highly stable state where they output 0.5 in response to all inputs, and develop the necessary hidden units only with great difficulty. Using a different error surface is unlikely to remove this saddle point, since its existence seems to result from the lack of any zero-order or first-order structure in the XOR test patterns. However, a different error measure might have different curvature there, making it easier or harder for the system to escape.

There is one important detail about the simulator that needs to be described. Different error measures have different optimal learning rates. In order that the measures could be compared without having to find optimal learning rates for each, and simply to increase performance, the weights were updated after each epoch according to the delta-bar-delta rule of Jacobs (1989). This is an improved version of the momentum heuristic (Rumelhart, Hinton & Williams, 1986). In momentum, the weight change vector is averaged with other, recent weight change vectors, so that oscillation on any dimension will cancel itself out. Like momentum, delta-bar-delta does not follow the gradient exactly. Each weight has its own, variable learning rate, empirically determined from the local curvature of the error surface in that dimension. The rule attempts to find optimal learning rates for all weights; it is applicable not only to back propagation, but to gradient descent techniques in general.[25]

Note also that the error measures above all require the network to achieve correct *activations* for their target nodes, not merely correct *outputs*. Outputs are given by logistic functions, and hence would always be very close to 0 or 1. One advantage of our gradual-response networks over a Boltzmann machine is that its inputs and outputs need not have this binary distinction. Continuous I/O variables can be implemented through setting and examining activations. This is a particular advantage for a net that is to make distinctions among

[25] The delta-bar-delta rule seemed to normalize learning in these experiments, although not as much as one would hope. In particular, scaling the error measure by a constant factor did result in changes in the learning rate. Why would this be? One possibility is that the network was at the bottom of a longitudinally curving ravine. In such a case, the learning rates will be presumably be forced to change on most time steps, so that the parameters that control how much they change per step become significant.

real-world inputs, or act as a memory.

The network was deemed to have solved a problem if its actual activations were within 0.1 of their targets. For binary targets, this is a stricter criterion than having output within 0.1 of target. There is no logistic function to force outputs toward 0 or 1, or to restrict them to the range $(0, 1)$. Indeed, a key property of continuous readout is that the value can err by being either too large *or* too small. If network output were restricted to $(0, 1)$, however, the network could be fairly sloppy, because it would be impossible for the output to overshoot a target of 1, or fall below a target of 0.

## 5.1 Results for feedforward XOR

As a basic test of the error measures, the system was asked to solve the ordinary XOR problem (almost two dozen times, using different parameters). The network used a feedforward 2-2-1 topology.

To check the model, it was first run using the standard back propagation definition of error, measured only at the very end of the trajectory. (That is, $E$ was integrated only over the last time step. A Williams and Zipser network would also be capable of doing this, with minimal modification.) The two runs required 1660 and 1876 epochs to converge.

As a further check on these runs, the system was asked to continually compare its weight changes with the weight changes prescribed by back propagation.[26] The two prescriptions were always identical to within a few percent.

Next, the network was given the "nodes toward targets" (NTT) error measure of section 4.3.1, using $\bar{\epsilon} = 1.0$ and $n = 1.005$. The results were excellent. On one run, the network took 1687 epochs to converge on the solution. On the other, it managed to satisfy the 10% criterion in only 942 epochs, and when allowed to continue running, passed the 1% criterion at epoch 1175. This was easily the best performance that any network achieved on the problem.

Hence, at least under these limited conditions, NTT was able to solve the problem in fewer epochs than back propagation (or Williams and Zipser).

[26] When $decay(a_i) = a_i$, as it did here, the asymptotic activations of the gradual-response net are identical to the activations of an ordinary feedforward net that uses the same weights. Since both nets compute the same mappings, and use the same error measure in this case, then they should compute the same gradient.

Of course, each epoch is computationally very demanding—but only because the algorithm can work equally well in recurrent networks. The point is that the NTT error surface has at least as clear a path to the zero-error minima as does sum-of-squares. Since the network's convergence is actually evaluated on sum-of-squares, the high performance of NTT is not a trivial result.

The Euclidean measure, at least for the tested values of $\epsilon$, actually converged far more slowly. Its short-term error reductions were only in the fourth decimal place; at 2000 epochs it was usually still trying to edge out of the saddle point. When there is only one target node, the Euclidean measure is similar to the NTT measure. The difference is that where NTT requires nodes that are far from their targets to approach faster, the Euclidean measure—since it asks the *square* of the distance to decrease at a constant rate—actually makes a lesser demand on faraway nodes.

This problem was first addressed by replacing $\epsilon$ in the error measure with $\epsilon\sqrt{Z}$. In other words, distance squared was required to decrease at a constant proportion of the distance, i.e., distance had to decrease at a constant rate. This revised Euclidean measure still took almost 3500 epochs to converge. Finally, $\epsilon$ was simply replaced with $\epsilon Z$, to get the same effect as NTT. In this case, the Euclidean measure was able to solve the problem in a (still slow) 2300 epochs.

It turns out that the NTT and Euclidean measures are sensitive to their parameters. If a network using one of these measures is required to move too quickly toward its targets, it will arrive very quickly at an unfortunate (but innovative) local minimum. Ignoring its input, it will simply dart far below 0 (or far above 1), incurring some error, and then begin to return at the required speeds. In this situation, it is moving toward its target regardless of whether that target happens to be 0 or 1!

Although the network's weights in this case do not solve the XOR problem, they demonstrate dramatically that the error measure *is* training dynamics: the model is learning how to move rather than how to be somewhere. Indeed, it has learned how to use its hidden units to generate a pulse. This is one of the behaviors noted for clusters in section 2.1.3. Unexpectedly, the network has achieved it without any recurrent connections, relying only on hidden units that pass threshold at different times.

The network also adopted this dodge on the two occasions it was given a "soft start" (see 4.3.1). It took advantage of the error measure's early leniency to quickly get below 0, then returned as required. One possible

fix for such aberrations is to use a hybrid measure, averaging together an ordinary sum-of-squares measure and one of the dynamic measures. This would require units to be both near the target and moving toward it. On two trials, the NTT measure was combined in this way with sum-of-squares. The hybrid measure did solve the XOR problem successfully in 1689 epochs. However, increasing $\epsilon$ only brought back the original difficulties.

## 5.2 Other tasks

The feedforward XOR tasks showed quite clearly that the model worked, and that the network was capable of learning particular dynamics at all. Furthermore, they showed that error measures based on network dynamics work even for problems whose solutions are static.

The other tests were less satisfying. Learning in recurrent networks appeared to work, but proved too slow to test fully. In addition, the content-addressable memory model turned out to have a serious flaw.

Recurrent-network learning was tested using a symmetric XOR problem. Since the general algorithm of section 3 is essentially a generalization of Williams and Zipser's (1988) algorithm, it must share that algorithm's ability to teach complex tasks to recurrent networks. The question is whether it can teach them effectively using something other than a sum-of-squares error measure.

The network was asked to solve symmetric XOR six times, using each of three error measures on both a two-hidden-unit and a three-hidden-unit topology (both fully recurrent). On one occasion, NTT was permitted to run for a long time, and found a solution at epoch 8336. For the other tests, which were terminated after 4000 epochs, no solution was found. In each case, error was reduced steadily—indeed, more steadily than in the feedforward task—but at an excruciatingly slow rate. When applying NTT to either network, for example, 4000 epochs were only enough to bring error down from a per-pattern average of 0.25 (the saddle point) to slightly over 0.24.

As for content-addressable memory, the first test revealed an unfortunate flaw in the approach of section 4.3.4.[27] The learning rule tries to make the

[27]The test involved a single visible unit that was supposed to learn three real-number "memories," of varying strengths, with the help of three hidden units and fully recurrent

network descend along the gradient of $G$ in activation space. It does indeed get the direction of the gradient correct at every point: it tries to achieve a velocity vector at each point $A$ that is exactly $-\xi/G(A)$ times the gradient at $A$. The difficulty is that $G(A)$ may be very small. A trajectory that comes too near a training pattern will thus have enormous weight changes prescribed for it in that region. As the simulated network approaches a solution, it is driven away again, often with enough force that it ends up oscillating across weight space.

There seems to be no way to avoid this behavior, except to have the network somehow compute $G(A)$ or a function thereof. (Indeed, it follows from the derivation at 4.3.4 that the net must know to make no weight changes on any pattern when $G(A) = 0$.) In order to compute $G(A)$, the model must take all the patterns into account at once. It cannot simply add up weight changes prescribed by the individual patterns.[28]

The model could be salvaged, of course, by having it explicitly compute $G(A)$ from all the patterns, at each $A$. But this fix makes it a far less attractive design for a content-addressable memory.

# 6 Conclusions

This research has attempted to understand the relationship between connectionist networks, especially recurrent networks, and dynamical systems. It has been instructive on several counts.

First of all, the dynamical systems perspective has proved to be fruitful. A key property of gradual-response networks is that their states can change gradually over time. We have seen the practical results of this even in tiny recurrent clusters and gradual-response feedforward nets. Apparently it does not take a very complex network to produce non-trivial kinds of movement through activation space.

It is important to stress this perspective because, until very recently, it has been ignored. Traditionally, the object of training a network has been to have the network produce certain static behaviors—constant output vectors. Networks are capable of much more than this, however. Their dynamics may

connections. In the terms of section 2.1.3, it was supposed to learn how to be a quantum unit.

[28] Logarithmic manipulations do not help.

have significant qualitative characteristics. The differences between a single attractor, a double attractor, and a limit cycle are far more pronounced than the difference between outputs of 0 and 1.

Second, it is possible to explicitly train networks to have particular dynamical behaviors. This was not known before. Even Jordan (1986), who described his model as a dynamical system, simply trained it to pass through particular points on particular time steps; Williams and Zipser (1988) did the same. In the experiments reported here, however, networks were not taught to be anywhere in activation space at any particular time, but only to move in a general direction toward their targets. The networks nonetheless succeeded in getting to their targets under these rather lenient conditions, establishing attractor basins around the targets. Moreover, they occasionally managed to fulfill the dynamical conditions in unexpected ways that they would not have found under the "equivalent" static conditions.

There are really two new results here. The theoretical result is that an algorithm exists to train dynamics. The experimental result is that even simple networks are actually capable of performing the behaviors they are being asked to learn.

Third, training a network's dynamic characteristics may not be any harder than training its static characteristics. On a mildly difficult mapping task, a network was discovered to learn equally well regardless of whether its position or its direction was prescribed.

Fourth, while the training techniques involved are difficult, they are not prohibitively difficult. A single general approach is sufficient to train networks to perform any of a wide class of behaviors. The approach is no more computationally intensive than its predecessor, the Williams and Zipser gradient descent algorithm for arbitrarily recurrent networks. Yet it extends the power of that algorithm well into dynamical systems territory.

There is more work left to be done. It is not yet clear what kinds of dynamics are easy for a network to learn and what kinds are hard; nor has the appropriateness of different error measures been systematically studied. Just as important, no one knows how topology affects the learning of dynamics—for instance, whether the cluster architectures explored in section 2.1.3 are as promising as they seem to be.

Be that as it may, the work here may very well keep its promises. Throughout the research, the dynamic properties of connectionist nets have proved to be continually interesting, sometimes surprising, and often encouraging

with respect to their overall significance for connectionism. With luck, these initial findings will have a chance at further development.

# References


[1] Elman, J. L. (1988). Finding structure in time (CRL Technical Report 8801). La Jolla: University of California, San Diego, Center for Research in Language.

[2] Hinton, G. E., & Sejnowski, T. J. (1986). Learning and relearning in Boltzmann machines. In D. E. Rumelhart & J. L. McClelland (Eds.), *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, 1* (pp. 282-317). Cambridge, MA: MIT Press.

[3] Hopfield, J. J. (1982). Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the National Academy of Sciences, USA, 81*, 6871-6874.

[4] Jacobs, R. A. (1987). Increased rates of convergence through learning rate adaptation (COINS Technical Report 87-117). Amherst, MA: University of Massachusetts, Department of Computer & Information Science.

[5] Jordan, M. I. (1986). Serial order: A parallel distributed processing approach (ICS Report 8604). La Jolla: University of California, San Diego, Institute for Cognitive Science.

[6] McClelland, J. L., & Rumelhart, D. E. (1981). An interactive activation model of context effects in letter perception: Part 1. An account of basic findings. *Psychological Review, 88*, 375-407.

[7] Pinker, S., & Prince, A. (1988). On language and connectionism: Analysis of a parallel distributed processing model of language acquisition. *Cognition 28*, 73-193.

[8] Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). Learning internal representations by error propagation. In D. E. Rumelhart & J. L. McClelland (Eds.), *Parallel Distributed Processing: Explorations*

*in the Microstructure of Cognition, 1* (pp. 318-364). Cambridge, MA: MIT Press.

[9] Rumelhart, D. E., & McClelland, J. L. (1986). On learning the past tenses of English verbs. In J. L. McClelland & D. E. Rumelhart (Eds.), *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, 2* (pp. 216-271). Cambridge, MA: MIT Press.

[10] Sejnowski, T. J., & Rosenberg, C. R. (1987). Parallel networks that learn to pronounce English text. *Complex Systems, 1*, 145-168.

[11] Williams, R. J., & Zipser, D. (1988). A learning algorithm for continually running fully recurrent neural networks (ICS Report 8805). Boston: Northwestern University, Dept. of Computer Science.