Discovering the Structure of a Reactive Environment by Exploration

Michael C. Mozer
Jonathan Bachrach

CU-CS-451-89

University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

Discovering the Structure of a Reactive Environment by Exploration

Michael C. Mozer
Jonathan Bachrach

CU-CS-451-89 November 1989

Michael C. Mozer
Department of Computer Science
and Institute of Cognitive Science
University of Colorado
Boulder, CO 80309-0430

e-mail: `mozer@boulder.colorado.edu`

Jonathan Bachrach
Department of Computer and Information Science
University of Massachusetts
Amherst, MA 01003

e-mail: `bachrach@cs.umass.edu`

Our thanks to Liz Jessup, Rob Schapire, Paul Smolensky, and Rich Sutton for helpful discussions. This work was supported by a grant from the James S. McDonnell Foundation to Michael Mozer, grant 87-2-36 from the Sloan Foundation to Geoffrey Hinton, and grant AFOSR-87-0030 from the Air Force Office of Scientific Research, Bolling AFB, to Andrew Barto.

Abstract

Consider a robot wandering around an unfamiliar environment, performing actions and sensing the resulting environmental states. The robot's task is to construct an internal model of its environment, a model that will allow it to predict the consequences of its actions and to determine what sequences of actions to take to reach particular goal states. Rivest and Schapire (1987a, 1987b; Schapire, 1988) have studied this problem and have designed a symbolic algorithm to strategically explore and infer the structure of "finite state" environments. The heart of this algorithm is a clever representation of the environment called an *update graph.* We have developed a connectionist implementation of the update graph using a highly-specialized network architecture. With back propagation learning and a trivial exploration strategy — choosing random actions — the connectionist network can outperform the Rivest and Schapire algorithm on simple problems. The network has the additional strength that it can accomodate stochastic environments. Perhaps the greatest virtue of the connectionist approach is that it suggests generalizations of the update graph representation that do not arise from a traditional, symbolic perspective.

Discovering the Structure of a Reactive Environment by Exploration

Imagine a robot placed in an unfamiliar environment. The robot is allowed to wander around the environment, performing actions and sensing the resulting environmental state. With sufficient exploration, the robot should be able to construct an internal model of the environment, a model that will allow it to predict the consequences of its actions and to determine what sequence of actions must be taken to reach a particular goal state. In this paper, we describe a connectionist network that accomplishes this task, based on a representation of finite-state automata developed by Rivest and Schapire (1987a, 1987b; Schapire, 1988). We begin by first describing several environments.

Sample environments

In each environment, the robot has a set of discrete *actions* it can execute to move from one environmental state to another. At each environmental state, a set of binary-valued *sensations* can be detected by the robot. Descriptions of four sample environments follow, suggested by the work of Rivest and Schapire.

The little prince world

The robot resides on the surface of a 2D planet. There are four distinct locations on the planet: north, south, east, and west. To the west, there is a rose; to the east, there is a volcano. The robot has two sensations, one indicating the presence of a rose at the current location, the other a volcano. The robot has available three actions: move to the next location in the direction the robot is currently facing, move to the next location away from the direction the robot is facing, and turn head around to face in the opposite direction.

The n-room world

The n-room world consists of n rooms arranged in a circular chain. Each room is connected to the two adjacent rooms: room 1 to rooms n and 2, room 2 to 1 and 3, and so forth. In each room is a light bulb and light switch. The robot can sense whether the light in the room where it currently stands is on or off. The robot has three possible actions: move to the next room down the chain, move to the next room up the chain, and toggle the light switch.

The car radio world

The robot manipulates a car radio that can receive three stations, each of which plays a different type of music: top 40, classical, and jazz. The radio has two "preset" buttons labeled X and Y, as well as "forward seek" and "backward seek" buttons. There are three sensations, one indicating whether the current station is playing top 40 music, another classical, and the last jazz. The robot has six actions available to it: recall the station in preset X or in preset Y, store the current station in preset X or in preset Y, search forwards from the current station for the next station, and search backwards for the next station.

You figure it out

To demonstrate the difficulty of the inference problem, consider an environment with two actions, `A` and `B`, and one binary-valued sensation. Try to predict the sensations that will be obtained given the following sequence of actions.

Action:	`A A B B A B A B B B B A B A A A B A B B A A B A B B`
Resulting Sensation:	`1 0 0 0 1 0 1 1 1 1 1 0 1 0 1 0 0 1 0 1 ? ? ? ? ? ?`

This is a simplified version of the n-room problem with only two rooms. Action `A` toggles the light switch, `B` moves from one room to the other, and the sensation indicates the state of the light in the current room. Initially, both lights are assumed to be off. People generally find this problem extremely challenging, if not insurmountably abstract, even when allowed to select actions and observe the consequences.

Modeling the environment with a finite-state automaton

The environments we wish to consider can be modeled by a finite-state automaton (*FSA*). Nodes of the FSA correspond to states of an environment. Labeled links connect each node of the automaton to other nodes and correspond to the actions that the robot can execute to move between environmental states. Associated with each node is a set of binary values — the sensations that can be detected by the robot in the corresponding environmental state. Figure 1 illustrates the FSA for a three-room world. Each node is labeled by the corresponding environmental state, coded in the form $r\text{-}s_1s_2s_3$, where r is the current room — 1, 2, or 3 — and s_i is the status of the light in room i — 0 for off and 1 for on. The sensation associated with each node is written in parentheses. Links between nodes are labeled with one of the three actions: toggle (T), move up (U), or move down (D).

The FSA represents the underlying structure of the environment. If the FSA is known, one can predict the sensory consequences of any sequence of actions. Further, the FSA can be used to determine a sequence of actions to take to obtain a certain goal state. For example, if the robot wishes to avoid light, it should follow a link or sequence of links for which the resulting sensation is 0.

Although one might try developing an algorithm to learn the FSA directly, there are several arguments against doing so (Schapire, 1988): (1) because many FSAs yield the same input/output behavior, there is no unique FSA; (2) knowing the structure of the FSA is unnecessary because we are only concerned with its input/output behavior; and, most importantly, (3) the FSA often does not capture redundancy inherent in the environment. As an example of this final point, in the n-room world, the T action has the same behavior independent of the current room number and the state of the lights in the other rooms, yet in the FSA of Figure 1, knowledge about "toggle" must be encoded for each room and in the context of the particular states of the other rooms. Thus, the simple semantics of an action like T are encoded repeatedly for each of the $n2^n$ distinct states.

Modeling the environment with an update graph

Rather than trying to learn the FSA, Rivest and Schapire suggest learning another representation of environment called an *update graph*. The advantage of the update graph representation is that in environments with many regularities, the number of nodes in the update graph can be much less than in the FSA (e.g., $2n$ versus $n2^n$ for the n-room world).[1] Schapire (1988) provides a formal definition and explanation of the update graph, which we summarize in the Appendix. In this section, we informally introduce the update graph representation.

Consider again the three-room world. To model this environment, the essential knowledge required is the status of the lights in the current room (CUR), the next room up from the current room (UP), and the next room down from the current room (DOWN). Assume the update graph has a node for each of these environmental variables. Further assume that each node has an associated value indicating whether the light in the particular room is on or off.

If we know the values of the variables in the current environmental state, what will their new values be after taking some action, say U? When the robot moves to the next room up, the new value of CUR becomes the previous value of UP; the new value of DOWN becomes the previous value of CUR; and in the three-room world, the new value of UP becomes the previous value of DOWN. As depicted in Figure 2a, this action thus results in shifting values around in the three nodes. This makes sense because moving up does not affect the status of any light, but it does alter the robot's position with respect to the three rooms. Figure 2b shows the analogous flow of information for the action D. The T action should cause the status of the current room to be complemented while the other two rooms remain unaffected (Figure 2c). In Figure 2d, the three sets of links from Figures 2a-c have been superimposed and have been labeled with the appropriate action.

[1] The downside of the update graph is that in environments without strong regularities, the size of the update graph can be exponentially larger than the size of the FSA.

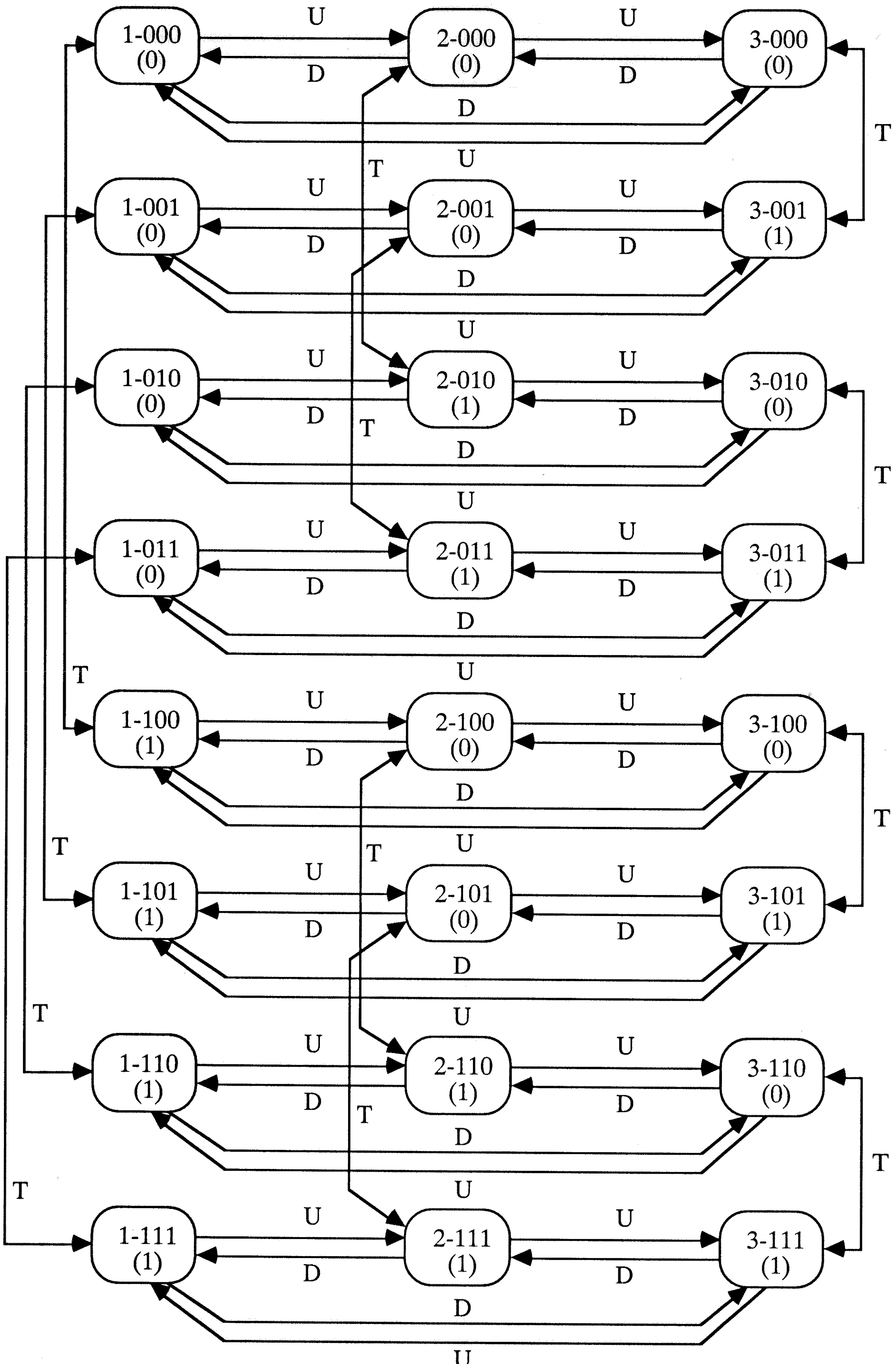

Figure 1. The FSA for a three-room world.

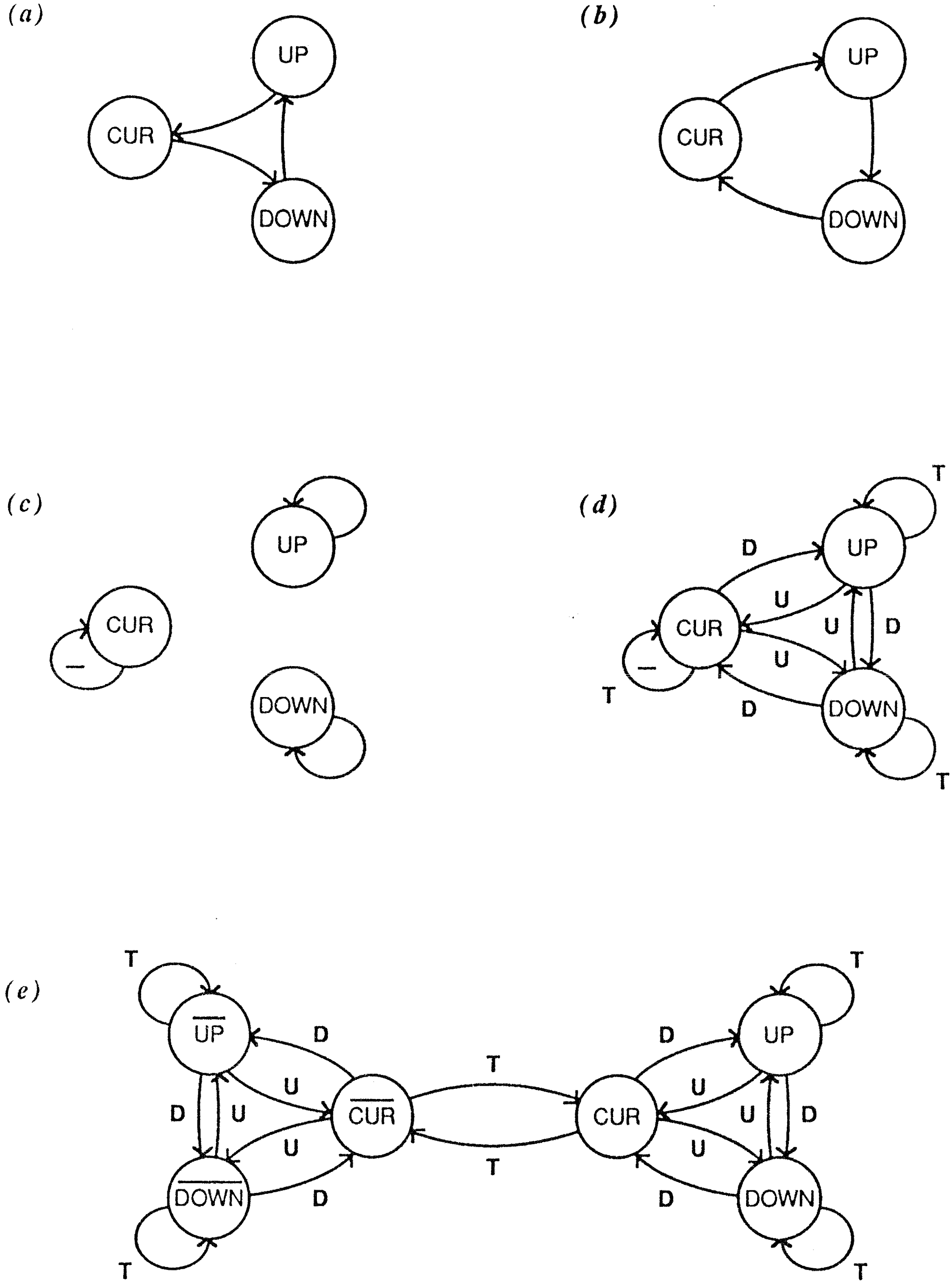

Figure 2. (a) Links between nodes indicating the desired information flow on performing the action `U`. CUR represents that status of the lights in the current room, UP the status of the lights in the next room up, and DOWN the status of the lights in the next room down. (b) Links between nodes indicating the desired information flow on performing the action `D`. (c) Links between nodes indicating the desired information flow on performing the action `T`. The "–" on the link from CUR to itself indicates that the value must be complemented. (d) Links from the three separate actions superimposed and labeled by the action. (e) The complementation link can be avoided by adding a set of nodes that represent the complements of the original set. This is the update graph for a three-room world.

One final detail: The update graph formalism does not make use of the "complementation" link. To avoid it, one may split each node into two values, one representing the status of a room and the other its complement (Figure 2e). Toggling thus involves exchanging the values of CUR and $\overline{\text{CUR}}$. Just as the values of CUR, UP, and DOWN must be rotated for the actions `U` and `D`, so must their complements.

Figure 2e is the update graph representation for the three-room world. Given this graph and the value of each node in the current environmental state, the result of any sequence of actions can be predicted simply by shifting values around in the graph. Thus, as far as predicting the input/output behavior of the environment is concerned, the update graph serves the same purpose as the FSA.

For every FSA, there exists a corresponding update graph. In fact, the update graph in Figure 2e might even be viewed as a distributed representation of the FSA in Figure 1. In the FSA, each environmental state is represented by *one* "active" node. In the update graph, each environmental state is represented by a pattern of activity across the nodes.

One defining and nonobvious (from the current description) property of an update graph is that each node has exactly one incoming link for each action. For example, CUR gets input from $\overline{\text{CUR}}$ for the action `T`, from UP for `U`, and from DOWN for `D`. Table 1, which represents the update graph in a slightly different manner, provides another way of describing the unique-input property. The Table shows node connectivity in the update graph, with a "1" indicating that two nodes are connected for a particular action, and "0" for no connection. The unique-input property specifies that there must be exactly one non-zero value in each row of each matrix.

The Rivest and Schapire algorithm

Rivest and Schapire have developed a symbolic algorithm (hereafter, the *RS algorithm*) to strategically explore an environment and learn its update graph representation. They break the learning problem into two steps: (a) inferring the structure of the update graph, and (b) maneuvering the robot into an environmental state where the

Table 1

update graph connectivity for *move up*

to node	*from node* CUR	UP	DOWN	$\overline{\text{CUR}}$	$\overline{\text{UP}}$	$\overline{\text{DOWN}}$
CUR	0	1	0	0	0	0
UP	0	0	1	0	0	0
DOWN	1	0	0	0	0	0
$\overline{\text{CUR}}$	0	0	0	0	1	0
$\overline{\text{UP}}$	0	0	0	0	0	1
$\overline{\text{DOWN}}$	0	0	0	1	0	0

update graph connectivity for *move down*

to node	*from node* CUR	UP	DOWN	$\overline{\text{CUR}}$	$\overline{\text{UP}}$	$\overline{\text{DOWN}}$
CUR	0	0	1	0	0	0
UP	1	0	0	0	0	0
DOWN	0	1	0	0	0	0
$\overline{\text{CUR}}$	0	0	0	0	0	1
$\overline{\text{UP}}$	0	0	0	1	0	0
$\overline{\text{DOWN}}$	0	0	0	0	1	0

update graph connectivity for *toggle*

to node	*from node* CUR	UP	DOWN	$\overline{\text{CUR}}$	$\overline{\text{UP}}$	$\overline{\text{DOWN}}$
CUR	0	0	0	1	0	0
UP	0	1	0	0	0	0
DOWN	0	0	1	0	0	0
$\overline{\text{CUR}}$	1	0	0	0	0	0
$\overline{\text{UP}}$	0	0	0	0	1	0
$\overline{\text{DOWN}}$	0	0	0	0	0	1

value of each node is known. Step (b) is relatively straightforward. Step (a) involves a method of experimentation to determine whether pairs of action sequences are equivalent in terms of their sensory outcomes. For a special class of environments, *permutation environments*, in which each action can be undone, the RS algorithm allows the robot to infer the environmental structure by performing a number of actions polynomial in the number of update graph nodes and the number of alternative actions available to the robot, given a measure ε of the acceptable probability of error.

While this work is extremely interesting, it has several limitations. First, the algorithm requires an upper bound on the number of nodes in the update graph, and learning time is quite dependent on this bound. Second, the algorithm formulates explicit hypotheses about regularities in the environment and tests these hypotheses one or a relatively small number at a time. Third, the algorithm requires the environment to be deterministic. If, say, the sensations obtained in a particular environmental state are not entirely reliable, the algorithm will fail. To address these limitations, we thought that a connectionist approach might be of interest. This is not to claim that a connectionist approach will supercede the impressive work of Rivest and Schapire, but that it might offer complementary strengths and alternative conceptualizations of the learning problem.

Viewing the update graph as a connectionist network

How might we turn the update graph into a connectionist network? Start by assuming one unit in a network for each node in the update graph. The activity level of the unit represents the truth value associated with the update graph node. Some of these units serve as "outputs" of the network. For example, in the three-room world, the output of the network is the unit that represents the status of the current room. In other environments, there may several sensations (e.g., the little prince world) in which case there will several output units.

What is the analog to the labeled links in the update graph? The labels indicate that values are to be sent down a link when a particular action occurs. In connectionist terms, the links should be *gated* by the action. To elaborate, we might include a set of localist units that represent the possible actions; these units act to multiplicatively gate the flow of activity between units in the update graph. Thus, if a particular action unit is on, the connections between units in the update graph that are gated by this action become enabled.

Because the action units form a local representation, i.e., only one is active at a time, exactly one set of connections is enabled at a time. Consequently, the gated connections can be replaced by a set of weight matrices, one per action (like those shown in Table 1). To predict the consequences of a particular action, say `T`, the weight matrix for `T` is simply plugged into the network and activity is allowed to propagate through the connections. Thus, the network is dynamically rewired contingent on the current action.

The effect of activity propagation should be that the new activity of a unit is the previous activity of some other unit. A simple linear activation function is sufficient to achieve this:

$$\mathbf{X}(t) = \mathbf{W}_{a(t)}\mathbf{X}(t-1)\,, \tag{1}$$

where $a(t)$ is the action selected at time t, $\mathbf{W}_{a(t)}$ is the weight matrix associated with this action, and $\mathbf{X}(t)$ is the activity vector that results from taking action $a(t)$. Assuming weight matrices like those shown in Table 1, which have zeroes in each row except for one connection of strength 1, the activation rule will cause activity values to be copied around the network.

Although generally linear activation functions are not appropriate for back propagation applications (Rumelhart, Hinton, & Williams, 1986), the architecture here permits such a simple function. The update graph architecture is thus a linear system, which is extremely useful because we can bring to bear the tools and methods of linear algebra for analyzing network behavior, as we discuss below.

Training the network to behave as an update graph

We have described a connectionist network that can behave as an update graph, and now turn to the procedure used to learn the connection strengths in this network. We begin by assuming that the number of units in the update graph is known in advance (we will generalize from this below). A weight matrix is required for each action, with a potential non-zero connection between every pair of units. As in most connectionist learning procedures, the weight matrices are initialized to random values; the outcome of learning will be a set of matrices like those in Table 1.

If the network is to behave as an update graph, the critical constraint on the connectivity matrices is that each row of each weight matrix should have connection strengths of zero except for one value which is 1 (assuming Equation 1). To achieve this property, additional constraints are placed on the weights. We have explored a combination of three constraints:

(1) $\sum_j w_{aij}^2 = 1$,

(2) $\sum_j w_{aij} = 1$, and

(3) $w_{aij} \geq 0$,

where w_{aij} is the connection strength to i from j for action a. If all three of these constraints are satisfied, the incoming weights to a unit are guaranteed to be all zeros except for one value which is 1. This can be intuited from Figure 3, which shows the constraints for a two-dimensional weight space. Constraint 1 requires that vectors lie on the unit circle, constraint 2 requires that vectors lie along the line $w_1 + w_2 = 1$, and constraint 3 requires that vectors lie in the first quadrant. Constraint 3 is redundant in a two-dimensional weight space but becomes necessary in higher dimensions.

Constraint 1 is satisfied by introducing a secondary error term,

$$E_{sec} = \sum_{a,i} (1 - \|\mathbf{W}_{ai}\|)^2,$$

where $\mathbf{W}_{ai}$ is the incoming weight vector to unit i for action a. The learning procedure attempts to minimize this error along with the primary error associated with predicting environmental sensations. Constraints 2 and 3 are rigidly enforced by renormalizing the $\mathbf{W}_{ai}$ following each weight update. The normalization procedure finds the shortest distance projection from the updated weight vector to the hyperplane specified by constraint 2 that also satisfies constraint 3.

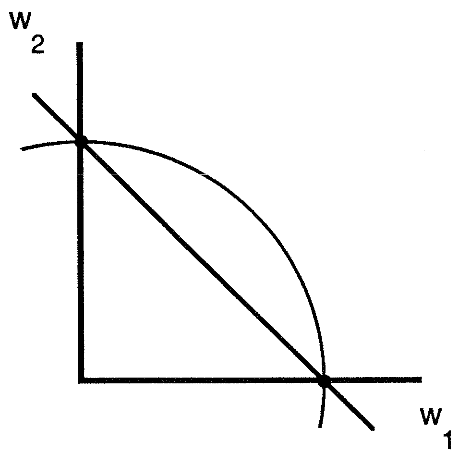

Figure 3. A two-dimensional space representing the weights, w_1 and w_2, feeding into a 2-input unit. The circle and line indicate the subregions specified by constraints 1 and 2, respectively. The two points of intersection are (1,0) and (0,1).

Details of the training procedure

Initially, a random weight matrix is generated for each action. Weights are selected from a uniform distribution in the range [0,1] and are normalized to satisify constraint 2. At each time step t, the following sequence of events transpires:

1. An action, $a(t)$, is selected at random.
2. The weight matrix for that action, $\mathbf{W}_{a(t)}$, is used to compute the activities at t, $\mathbf{X}(t)$, from the previous activities $\mathbf{X}(t-1)$.
3. The selected action is performed on the environment and the resulting sensations are observed.
4. The observed sensations are compared with the sensations predicted by the network (i.e., the outputs of units chosen to represent the sensations) to compute a measure of error.
5. The back propagation "unfolding-in-time" procedure (Rumelhart, Hinton, & Williams, 1986) is used to compute the derivative of the error with respect to weights at the current and earlier time steps, $\mathbf{W}_{a(t-i)}$, for $i=0 \cdots \tau-1$. Error derivatives are also computed based on constraint 1.
6. The weight matrices for each action are updated using the overall error gradient and then are renormalized to enforce constraints 2 and 3.
7. The temporal record of unit activities, $\mathbf{X}(t-i)$ for $i=0 \cdots \tau$, which is maintained to permit back propagation in time, is updated to reflect the new weights. (See further explanation below.)
8. The output units at time t, which represent the predicted sensations, are replaced by the actual observed sensations.[2]

Steps 5-7 require further elaboration. The error measured at step t may be due to incorrect propagation of activities from step $t-1$, which would call for modification of the weight matrix $\mathbf{W}_{a(t)}$. But the error may also be attributed to incorrect propagation of activities at earlier times. Thus, back propagation is used to assign blame to the weights at earlier times. One critical parameter of training is the amount of temporal history, τ, to consider. We have found that, for a particular problem, error propagation beyond a certain critical number of steps does not improve learning performance, although any fewer does indeed harm performance. In the results described below, we set τ for a particular problem to what appeared to be a safe limit: one less than the number of nodes in the update graph solution of the problem.

To back propagate error in time, it is necessary to maintain a temporal record of unit activities. However, a problem arises with these activities following a weight update: the activities are no longer consistent with the weights — i.e., Equation 1 is violated. Because the error derivatives computed by back propagation are exact only when Equation 1 is satisfied, future weight updates based on the inconsistent activities are not assured of being correct. Empirically, we have found the algorithm extremely unstable if we do not address this problem.

In most situations where back propagation is applied to temporally-extended sequences, the sequences are of finite length. Consequently, it is possible to wait until the end of the sequence to update the weights, at which point consistency between activities and weights no longer matters because the system starts fresh at the beginning of the next sequence. In the present situation, however, the sequence of actions does not terminate. We thus were forced to consider alternative means of ensuring consistency. One approach we tried involved updating the weights only after every, say, 25 steps. Immediately following the update, the weights and activities are inconsistent, but after τ steps (when the inconsistent activities drop off the activity history record), consistency is once again achieved. A more successful approach involved updating the activities after each weight change to force consistency (step 7 of

[2] A consequence of this substitution is that error should not be back propagated from output units at time t to output units at times $t-1$, $t-2$, etc. It is not sensible to adjust the response of output units at time, say, $t-1$ to achieve the correct response at time t because it is already known what values the output units at $t-1$ must take. In terms of the unfolded network, the output units at times $t-i$ serve as fixed input values; the network does not have the freedom to adjust their incoming weights, and hence their values.

the list above). To do this, we propagated the earliest activities in the temporal record, $\mathbf{X}(t-\tau)$, forward again to time t, using the updated weight matrices.[3]

The issue of consistency arises because at no point in time is the network instructed as to the state of the environment. That is, instead of being given an activity vector as input, part of the network's learning task is to discover the appropriate activity vector. This might suggest a strategy of explicitly learning the activity vector, that is, performing gradient descent in both the weight space and activity space. However, our experiments indicated that this strategy does not improve the network's performance. One plausible explanation is the following. If we perform gradient descent in weight space based on the error from a single trial, and then force activity-weight consistency, the updated output unit activities are guaranteed to be closer to the target values (assuming a sufficiently small learning rate and that the weight constraints have minor influence). Thus, the effect of this procedure is to reduce the error in the observable components of the activity vector, which is similar to performing gradient descent in activity space directly.

A final comment regarding the training procedure: In our simulations, learning performance was better with target activity levels of -1 and +1 (for *light is off* and *on*, respectively) rather than 0 and 1. One explanation for this is that random activations and random (nonnegative) connection strengths tend to cancel out in the -1/+1 case, but not in the 0/1 case.

Results

Figure 4 shows the weights in the update graph network for the three-room world at three stages of learning. The "step" refers to how many actions the robot has taken, or equivalently, how many times the weights have been updated. The bottom diagram in the Figure depicts a connectivity pattern identical to that presented in Table 1, and corresponds to the update graph of Figure 2e.[4] To explain the correspondence, think of the diagram as being in the shape of a person who has a head, left and right arms, left and right legs, and a heart. For the action `U`, the head — the output unit — receives input from the left leg, the left leg from the heart, and the heart from the head, thereby forming a three-unit loop. The other three units — the left arm, right arm, and right leg — form a similar loop. For the action `D`, the same two loops are present but in the reverse direction. These two loops also appear in Figure 2e. For the action `T`, the left and right arms, heart, and left leg each keep their current value, while the head and the right leg exchange values. This corresponds to the exchange of values between the CUR and $\overline{\text{CUR}}$ nodes of the Figure 2e.

In addition to learning the update graph connectivity, the network has simultaneously learned the correct activity values associated with each node for the current state of the environment. Armed with this information, the network can predict the outcome of any sequence of actions. Indeed, the prediction error drops to zero, causing learning to cease and the network to become completely stable.

Now for the bad news: The network does not converge for every set of random initial weights, and when it does, it requires on the order of 6,000 steps, much greater than the RS algorithm. However, when the weight constraints are removed, we discovered that the network converges without fail and in about 300 steps. It appears that only in extremely small problems do the weight constraints help the network discover a solution. We consider below why the weight constraints are harmful and possible remedies.

---

[3] Keeping the original value of $\mathbf{X}(t-\tau)$ is a somewhat arbitrary choice. Consistency can be achieved by propagating *any* value of $\mathbf{X}(t-\tau)$ forward in time, and there is no strong reason for believing $\mathbf{X}(t-\tau)$ is the appropriate value. We thus suggest two alternative schemes, but have not yet tested them. First, we might select $\mathbf{X}(t-\tau)$ such that the new $\mathbf{X}(t-i)$, $i = 0 \cdots \tau-1$, are as close as possible to the old values. Second, we might select $\mathbf{X}(t-\tau)$ such that the output units produce as close to the correct values as possible. Both these schemes require the computation-intensive operation of finding a least squares solution to a set of linear equations.

[4] The definitive connectionist light bulb joke (courtesy of Thomas Mastaglio):

Q: How many connectionist networks does it take to change a light bulb?
A: Only one, but it requires 6,000 trials.

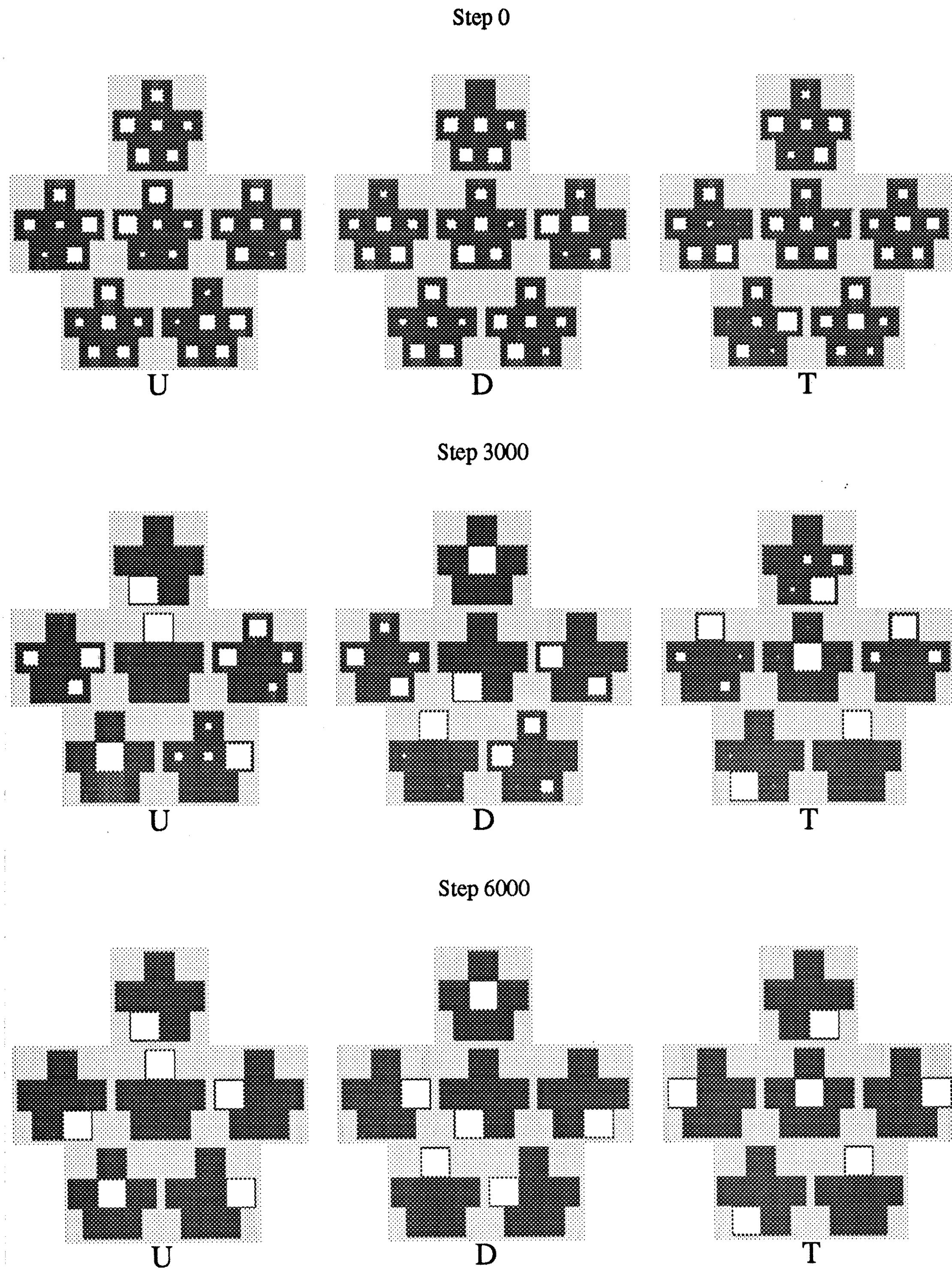

Figure 4. The three-room world. Weights learned by a six-unit network at three stages of learning: step 0 reflects the initial random weights, step 3000 reflects the weights midway through learning, and step 6000 reflects the weights upon completion of learning. Each large diagram represents the weights corresponding to one of the three actions. Each small diagram contained within a large diagram represents the connection strengths feeding into a particular unit for a particular action. There are six units, hence six small diagrams. The output unit, which indicates the state of the light in the current room, is the protruding "head" of the large diagram. A white square in a particular position of a small diagram represents the strength of connection from the unit in the homologous position in the large diagram to the unit represented by the small diagram. The area of the square is proportional to the connection strength.

Without weight constraints, there are two problems. First, the system has difficulty converging onto an exact solution. One purpose that the weight constraints serve is to lock in on a set of weights when the system comes sufficiently close; without the constraints, we found it necessary to scale the learning rate in proportion to the mean-squared prediction error to avoid overstepping solutions. Second, the resulting weight matrix, which contains a collection of positive and negative weights of varying magnitudes, is not readily interpreted (see Figure 5). In the case of the n-room world, one reason why the final weights are difficult to interpret is because the net has discovered a solution that does not satisfy the update graph formalism; it has discovered the notion of complementation links of the sort shown in Figure 2d. With the use of complementation links, only three units are required, not six. Consequently, the three unnecessary units are either cut out of the solution or encode information redundantly. Solutions of the network are much easier to understand when the network consists of only three units. Figure 6 depicts one such solution, which corresponds to the graph in Figure 2d. The network also discovers other solutions in which two of the three connections in the three-unit loop are negative, one negation cancelling out the effect of the other.

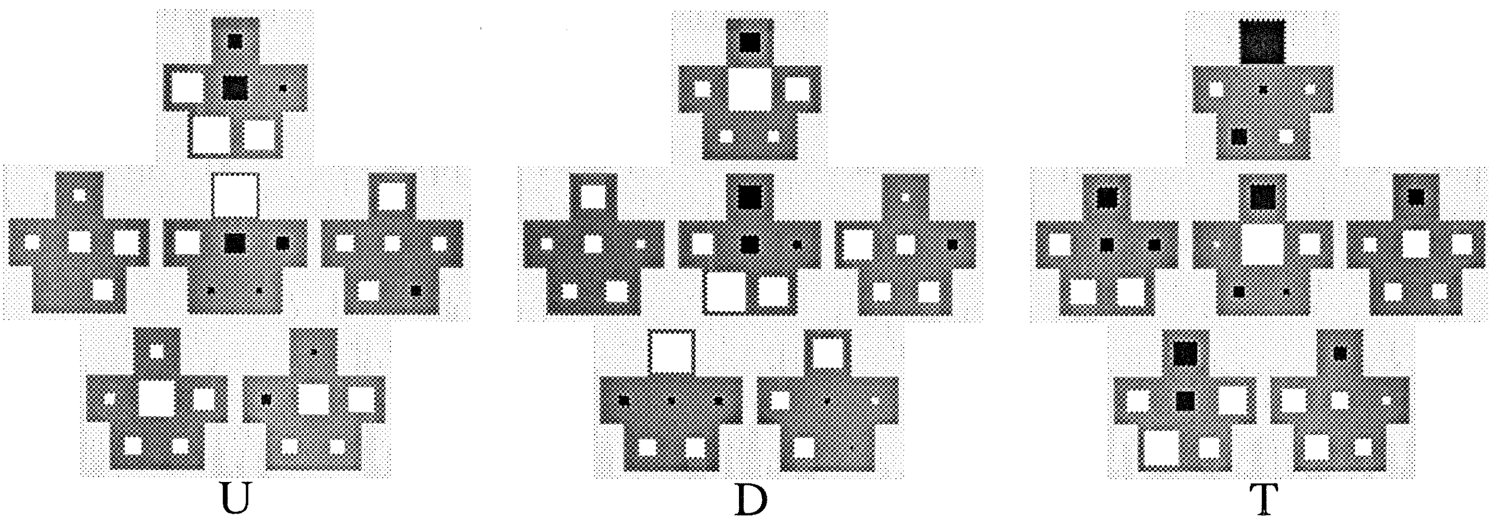

Figure 5. The three-room world. Weights learned by a six-unit network without weight constraints. Black squares indicate negative weights, white positive.

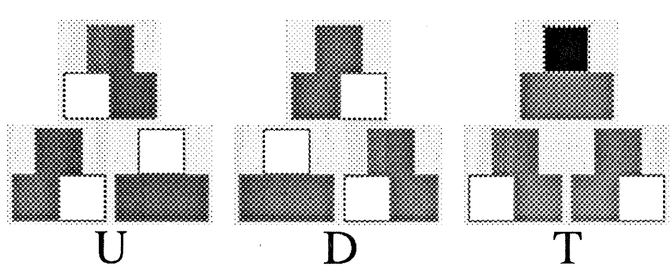

Figure 6. The three-room world. Weights learned by a three-unit network without weight constraints. (The weights have actually been cleaned up slightly to make the result clearer.)

Table 2 compares the performance of the RS algorithm against that of the connectionist network without weight constraints for a sampling of environments.[5] Performance is measured in terms of the number of actions the robot must take before it is able to predict the outcome of subsequent actions, that is, the number of actions required to learn the update graph structure and the truth value associated with each node. The performance data reported for the connectionist network was the median over 25 replications of each simulation. The network was considered to have learned the task on a given trial if the correct predictions were made for at least the next 2,500 steps. All differences between the two algorithms were statistically reliable.

The learning rates used in our simulations were adjusted dynamically every 100 steps by averaging the current learning rate with a rate proportional to the mean squared error obtained on the last 100 steps. Several runs were made to determine what initial learning rate and constant of proportionality yielded the best performance. It turned out that performance was relatively invariant under a wide range of these parameters. Momentum did not appear to help.[6]

In simple environments, the connectionist update graph can outperform the RS algorithm. These results are quite surprising when considering that the action sequence used to train the network is generated at random, in contrast to the RS algorithm, which involves a strategy for exploring the environment. We conjecture that the network does as well as it does because it considers and updates many hypotheses in parallel at each time step.

In complex environments, the network does poorly. By "complex", we mean that the number of nodes in the update graph is quite large and the number of distinguishing environmental sensations is relatively small. For example, the network failed to learn a 32-room world, whereas the RS algorithm succeeded. An intelligent exploration strategy seems necessary in this case: random actions will take too long to search the state space. Search becomes less critical if the robot can directly sense more information about the environment. For example, learning the 32-room world becomes trivial if the network is able to sense the states of all 32 rooms at once.

Noisy environments

The RS algorithm cannot readily be extended to handle environments with stochastic sensations. In contrast, our network's performance degrades gracefully in the presence of noise. For example, the network is able to learn the update graph for the little-prince world even when sensations are unreliable — when sensations are registered incorrectly about 10% of the time or less. To train the network properly in noisy environments, however, the training procedure must be altered. If the observed sensation replaces the network's predicted sensation and the observed sensation is incorrectly registered, the values of nodes in the graph are disrupted and the network requires several noise-free steps to recover. Thus, a procedure might be used in which the predicted sensation is not completely replaced by the observed sensation, but rather some average of the two is computed; additionally, the average should be weighted towards the prediction as the network's performance improves.

Table 2

Number of Steps Required to Learn Update Graph

Environment	RS Algorithm	Connectionist Update Graph
Little Prince World	200	91
Car Radio World	27,695	8,167
Three-Room World	?	298
Four-Room World	1,388	1,509
32-Room World	52,436	fails

[5] We thank Rob Schapire for providing us with the latest results from his work.

[6] Just as connectionist simulations require a bit of voodoo in setting learning rates, the RS algorithm has its own set of adjustable parameters that influence performance. One of us (JB) experimented with the RS algorithm, and without expertise in parameter tweaking, was unable to obtain performance in the same range as the measures reported in Table 2.

Prior specification of update graph size

The RS algorithm requires an upper bound on the number of nodes in the update graph. The results presented in Table 2 are obtained when the RS algorithm knows exactly how many nodes are required in advance. The algorithm fails if it is given an upper bound less than the required number of nodes, and performance degrades as the upper bound increases above the required number. The connectionist network will also fail if it is given fewer units than are necessary for the task. However, performance does not appear to degrade as the number of units increases beyond the minimal number. Table 3 presents the median number of steps required to learn the four-room world as the number of units in the network is varied. Although performance is independent of the number of units here, extraneous units greatly *improve* performance when the weight constraints are applied. Only 3 of 25 replications of the four-room world simulation with 8 units and weight constraints successfully learned the update graph (the simulation was terminated after 100,000 steps), whereas 21 of 25 replications succeeded when 16 units were used.

Generalizing the update graph formalism

Having described the overall performance of the network, we return to the issue of why weight constraints appear to harm performance. One straightforward explanation is that there are many possible solutions, only a small number of which correspond to update graphs. With weight constraints, the network is prevented from finding alternative solutions. One example of an alternative solution is the network with complementation links presented in Figure 6. Allowing complementation links can halve the number of update graph nodes required for many environments.

An even more radical generalization of the update graph formalism arises from the fact that the network is a linear system. Consider a set of weight matrices, $\{\mathbf{W}_a\}$, that correspond to a particular update graph, i.e., each row of each matrix satisfies the constraint that all entries are zero except for one entry that is one (as in Table 1). Further, assume the vector **X** indicates the current values of each node. Given the previously-stated activation rule,

$$\mathbf{X}(t) = \mathbf{W}_{a(t)}\mathbf{X}(t-1)\ ,$$

one can construct an equivalent system,

$$\mathbf{X}'(t) = \mathbf{W}'_{a(t)}\mathbf{X}'(t-1)$$

by substituting

$$\mathbf{X}'(t) \equiv \mathbf{Q}\mathbf{X}(t)$$

and

$$\mathbf{W}'_{a(t)} \equiv \mathbf{Q}\mathbf{W}_{a(t)}\mathbf{Q}^* .$$

Table 3

Four-Room World	
Units in Network	Number of Steps to Learn Update Graph
4	2028
6	1380
8	1509
10	1496
12	1484
14	1630
16	1522
18	1515
20	1565

Here, the $\mathbf{W}_a$ have dimensions $n \times n$, the $\mathbf{W}'_a$ have dimensions $m \times m$, $\mathbf{Q}$ is any matrix of dimensions $m \times n$ and rank n, and $\mathbf{Q}^*$ is the left inverse of $\mathbf{Q}$. The transformed system consisting of $\mathbf{X}'$ and the $\mathbf{W}'_a$ is isomorphic to the original system; one can determine a unique $\mathbf{X}$ for each $\mathbf{X}'$, and hence one can predict sensations in the same manner as before. However, the transformed system is different in one important respect: The $\mathbf{W}'_a$ do not satisfy the one-nonzero-weight-per-row constraint.

Because the set of matrices $\mathbf{Q}$ that meet our requirements is infinite, so is the number of $\{\mathbf{W}'_a\}$ for each $\{\mathbf{W}_a\}$. A network being trained without weight constraints is free to discover virtually *any* of these $\{\mathbf{W}'_a\}$ (the only restriction being that the mapping from $\mathbf{X}'$ to $\mathbf{X}$ is the identity for the output units, because error is computed from $\mathbf{X}'$ directly, not $\mathbf{X}$). Thus, each solution $\{\mathbf{W}_a\}$ that meets the formal definition of an update graph is only a special case — with $\mathbf{Q}$ being the identity matrix — of the more general $\{\mathbf{W}'_a\}$. Requiring the network to discover this particular solution can complicate learning, as we observed when training the network with weight constraints.

If the connectivity restrictions on $\mathbf{W}$ that define an update graph could be generalized to $\mathbf{W}'$, these generalized restrictions could be applied to discover a large set of solutions that nonetheless correspond to an update graph. Unfortunately, it does not appear that the restrictions can be mapped in any straightforward way.

An alternative approach we have considered is to train the network to discover a solution $\{\mathbf{W}'_a\}$ which can then be decomposed into an update graph $\{\mathbf{W}_a\}$ and a transformation matrix $\mathbf{Q}$. The decomposition could be attempted post hoc, but our experiments thus far have consisted of explicitly defining $\mathbf{W}'$ in terms of $\mathbf{Q}$ and $\mathbf{W}$. That is, an error gradient is computed with respect to $\mathbf{W}'$, which is then translated to gradients with respect to $\mathbf{Q}$ and $\mathbf{W}$.[7] In addition, the previously-described constraints on $\mathbf{W}$ are applied. Although the network must still discover the update graph $\{\mathbf{W}_a\}$, we hoped that introducing $\mathbf{Q}$ would allow alternative paths to the solution. This method did help somewhat, but unfortunately, performance did not reach the same levels as training with no weight constraints whatsoever. Figure 7 shows one solution obtained by the network under this training regimen.

Conclusions

We have demonstrated several benefits of a connectionist approach to the problem of learning the structure of a finite-state environment using an update graph.

- Connectionist learning algorithms provide the potential of parallelism. After the outcome of a single action is observed, nearly all weights in the network are adjusted simultaneously. In contrast, the RS algorithm performs actions in order to test one or a small number of hypotheses; that is, each action is used to determine whether or not, say, two particular nodes should be connected. A further example of parallelism is that the connectionist network learns the update graph structure at the same time as the appropriate unit activations, whereas the RS algorithm approaches the two tasks sequentially.

- Performance of the learning algorithm appears insensitive to prior knowledge of update graph complexity. As long as the network is given at least as many units as required, the presence of additional units does not impede learning. The network either disconnects the unnecessary units from the graph or uses multiple units to encode information redundantly. In contrast, efficient performance of the RS algorithm depends on a reasonable estimate of the number of nodes in the update graph.

- During learning, the connectionist network constantly makes predictions about what sensations will result from a particular action. These predictions gradually improve with experience, and even before learning is complete, the predictions can be substantially correct. The RS algorithm cannot make predictions based on its partially constructed update graph. Although the algorithm could perhaps be modified to do so, there would be an associated cost.

---

[7] This is not a simple matter due to the fact that $\mathbf{W}'$ is composed of $\mathbf{Q}^*$ as well as $\mathbf{Q}$, and the $\mathbf{Q}^*$ gradient must be transformed into a $\mathbf{Q}$ gradient. Consequently, we constrained $\mathbf{Q}$ to be an orthogonal matrix. For orthogonal matrices, $\mathbf{Q}^{-1} = \mathbf{Q}^T$, which trivializes the mapping from $\mathbf{Q}^*$ gradients to $\mathbf{Q}$ gradients.

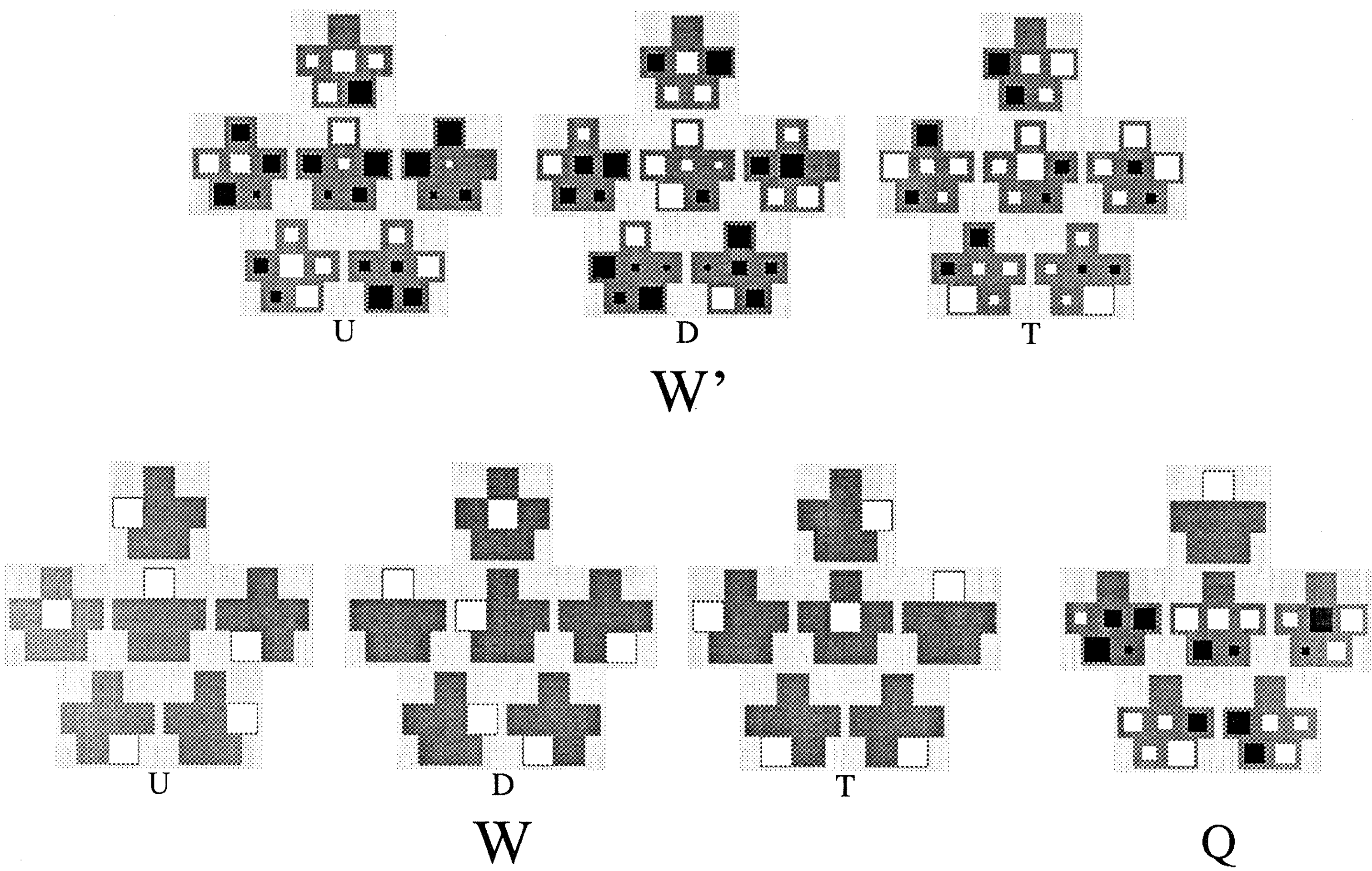

Figure 7. The three-room world. The $\{\mathbf{W}'_a\}$ are weight matrices learned by a six-unit network, along with an activity vector, $\mathbf{X}'$. Although the weights do not appear to correspond to an update graph, they in fact can be decomposed into matrices $\mathbf{Q}$ and $\{\mathbf{W}_a\}$ according to $\mathbf{W}'_a = \mathbf{Q}\mathbf{W}_a\mathbf{Q}^*$.

- Connectionist learning algorithms are able to accomodate environments in which the sensations are somewhat unreliable. The RS algorithm is designed for deterministic environments.

- Treating the update graph as matrices of connection strengths has suggested generalizations of the update graph formalism that don't arise from a more traditional analysis. We presented two generalizations. First, there is the fairly direct extension of allowing complementation links. Second, because the connectionist network is a linear system, any rank-preserving linear transform of the weight matrices will produce an equivalent system, but one that does not have the local connectivity of the update graph. The linearity of the network also allows us to use tools of linear algebra to analyze the resulting connectivity matrices.

We emphatically do not claim that the connectionist approach supercedes the impressive work of Rivest and Schapire. However, it does appear to offer complementary strengths and alternative conceptualizations of the learning problem.

The connectionist approach has two fundamental problems that must be overcome if it is to be considered seriously as an alternative to the symbolic approach. First, using a random exploration strategy, the connectionist network has no hope of scaling to complex environments. An intelligent exploration strategy could potentially be incorporated to force the robot into states where the network's predictive abilities are poor. Second, our greatest successes have occurred when we allowed the network to discover solutions that are not necessarily isomorphic to an update graph. There is promise, however, of using generalizations of the update graph suggested by the connectionist approach to allow a much larger number of formally equivalent solutions.

While the connectionist perspective has provided new insights into a problem that has been studied from a symbolic perspective, the symbolic work — specifically, the update graph representation — has proven beneficial in the development of our connectionist architecture. Our original experiments in this domain involved generic connectionist architectures (e.g., Elman, 1988; Mozer, 1989) and were spectacularly unsuccessful. Many simple environments could not be learned, predictions were often inaccurate, and a great deal of training was required. The update graph representation is a powerful tool. It provides strong constraints on network architecture, dynamics, and training procedure.

Appendix

In explaining the update graph, we informally characterized the nodes of the graph as representing states of the environment relative to the observer's current position. However, the nodes have a formal semantics, which we present in this Appendix, after first introducing a bit more terminology.

A *test* on the environment can be defined as a sequence of zero or more actions followed by a sensation. A test is performed by executing the sequence of actions from the current environmental state and then detecting the resulting sensation. For example, in the n-room world, some tests include: `UU?` (move up twice and sense the state of the light), `UTD?` (move up, toggle the light, move down, and then sense the state of the light), `?` (sense the state of the light in the current room). The value of a test is the value of the resulting sensation.[8]

Certain tests are equivalent. For example, in the three-room world `UU?` and `D?` will always yield the same outcome, independent of the current environmental state. This is because moving up twice will land the robot in the same room as moving down once, so the resulting sensation will be the same. The tests `UUUUU?`, `TUU?`, `DUTDDDD?` are also equivalent to `UU?` and `D?`. Each set of equivalent tests defines an *equivalence class*. Rivest and Schapire call the number of equivalence classes the *diversity* of the environment. The n-room world has a diversity of $2n$, arising from there being n lights that can be sensed, either in their current state or toggled.

Each node in the update graph corresponds to one test equivalence class. The binary variable associated with each node represents the truth value of the corresponding test given the current global state of the environment.

The directed, labeled links of the update graph arise from relations among equivalence classes: There is a link from node α to node β labeled with *action* if the test represented by α is equivalent to (i.e., will always yield the same result as) executing *action* followed by test β. For instance, there is a link from the node representing the class containing `T?` (which we have indicated as $\overline{\text{CUR}}$ in Figure 2e) to the node representing the class containing `UT?` (which we have indicated as $\overline{\text{UP}}$ in Figure 2e), and this link is labeled `D` because executing the action `D` followed by the test `UT?` is equivalent to executing the test `T?` (the `U` and `D` actions cancel).

Based on this definition of nodes and links, it is clear why each node has exactly one incoming link for each action. If a node β received projections from two nodes, α_1 and α_2 for some *action*, this would imply that the equivalence classes α_1 and α_2 both contained the test consisting of *action* followed by test β. If the two equivalence classes contain the same test, they must represent the same equivalence class, and hence will be collapsed together.

[8] In the n-room world, there is only one sensation — the state of the light; thus, each test ends by evaluating this sensation. In environments having multiple sensations, tests can end with different sensations.

References

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

Mozer, M. C. (1989). A focused back-propagation algorithm for temporal pattern recognition. *Complex Systems*. In Press.

Rivest, R. L., & Schapire, R. E. (1987). Diversity-based inference of finite automata. In *Proceedings of the Twenty-Eighth Annual Symposium on Foundations of Computer Science* (pp. 78-87).

Rivest, R. L., & Schapire, R. E. (1987). A new approach to unsupervised learning in deterministic environments. In P. Langley (Ed.), *Proceedings of the Fourth International Workshop on Machine Learning* (pp. 364-375).

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. Volume I: Foundations* (pp. 318-362). Cambridge, MA: MIT Press/Bradford Books.

Schapire, R. E. (1988). *Diversity-based inference of finite automata*. Unpublished master's thesis, Massachusetts Institute of Technology, Cambridge, MA.