Chapter 3

MODELING FINITE STATE AUTOMATA

The architecture of our connectionist network model is based on a representation of finite-state environments developed by Rivest and Schapire [72, 74, 85]. Rivest and Schapire have also developed a symbolic algorithm to infer this representation by exploring the environment. We have taken a connectionist approach to the same problem, and we show that the connectionist approach offers complementary strengths and new insights into the problem.[1] We begin by describing several environments of the sort we wish to model.

3.1 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 discrete-valued *sensations* can be detected by the robot. Descriptions of five sample environments follow, the first four of which come from Rivest and Schapire [85]. Note that in these environments, environmental states are not distinguished by unique sensory configurations; hence, the environmental state cannot be determined from the sensations alone.

[1]This research was conducted in collaboration with M. C. Mozer and is reported in refs. [59, 60, 58, 5].

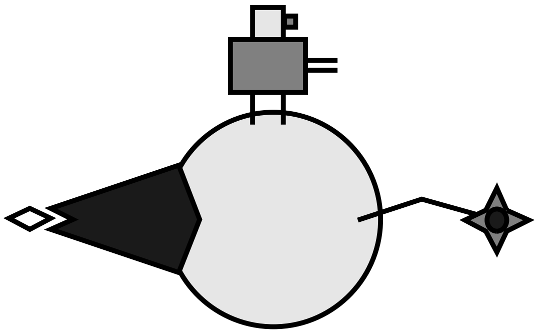


Figure 3.1. The Little Prince Environment.

The Little Prince Environment

The robot resides on the surface of a 2D planet (Figure 3.1). There are four distinct locations on the planet: north, south, east, and west. To the east, there is a rose; to the west, 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 it is currently facing, move to the next location away from the direction it is facing, and turn its head around to face in the opposite direction.

The Car Radio Environment

The robot can fiddle with knobs on a radio that receives only three stations. It can recognize the type of music that the radio is playing as either classical, rock, or news (Figure 3.2). It can tune in a station with left-scan and right-scan buttons, which scan from one station to the next with wraparound. The robot can also manipulate a set of two presets, X and Y. It can save the current station in either preset, and it can recall a setting in either preset.

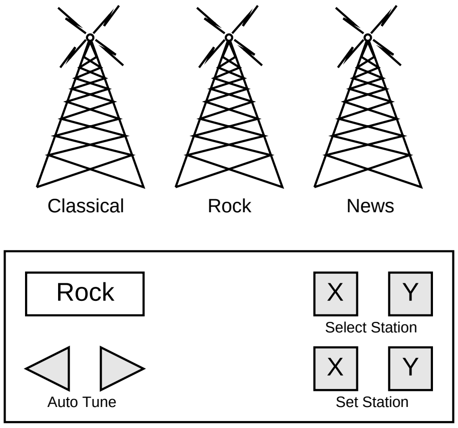


Figure 3.2. The Car Radio Environment.

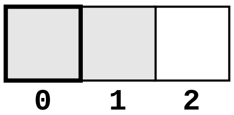


Figure 3.3. An Example of the n-Bit Register Environment. White squares indicate a bit is on, and shaded squares indicate a bit is off. The bold square on the left is the only bit that the robot can sense.

The n-Bit Shift Register Environment

The robot senses the value on the leftmost bit of an n-bit shift register as shown in Figure 3.3. It can rotate the register left or right. The robot can also flip the leftmost bit.

The $n \times n$ Grid Environment

A robot is placed in an $n \times n$ grid with wraparound as shown in Figure 3.4. The robot occupies a square and faces in one of four geographic directions: north, south, east, or west. It can move forward one square or rotate 90° left or right with the left-turn or right-turn actions. Each square is colored either red, green, or blue and the robot has sensors that detect the color of the square it is facing. Moving forward

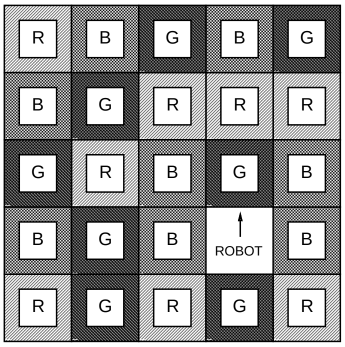


Figure 3.4. An Example of the $n \times n$ Grid Environment.

not only changes the robot's position, but also paints the previous square the color of the square the robot now occupies.

The $n \times n$ Checkerboard Environment

The checkerboard environment consists of an $n \times n$ grid of squares as shown in Figure 3.5. The robot occupies a particular square and can sense the landmark in that square, if any. There are a number of unique landmarks distributed in a checkerboard pattern across the grid. These are depicted as the numbers in the squares. Half of the squares have no landmark and are indistinguishable from each other. At each time step the robot can move either up, down, left, or right one square. Movement off one edge of the grid wraps around to the other side.

3.2 The Update Graph

Rather than trying to learn a state transition graph machine, Rivest and Schapire suggest learning a discrete-distributed machine, called an *update graph*. The advantage of the update graph is that in environments with many regularities, the size of the update graph can be much smaller than the size of the state transition graph machine,

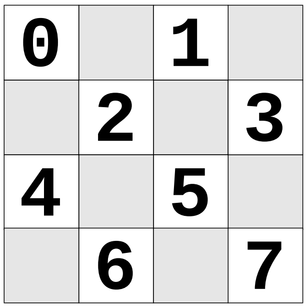


Figure 3.5. An Example of the $n \times n$ Checkerboard Environment.

for example, $2n$ versus 2^n state variables in the n-bit shift register environment.[2] The Appendix presents Rivest and Schapire's formal definition of the update graph. Rivest and Schapire's definition is based on the notion of *tests* that can be performed on the environment, and the equivalence of different tests. In this section, we present an alternative, more intuitive view of the update graph that facilitates a connectionist interpretation.

Consider again the 3-bit shift register environment. To model this environment, the essential knowledge required is the values of the bits. Assume the update graph has a node for each of these environment variables, and each node has an associated value indicating whether the bit 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 `L`? The new value of the leftmost bit, bit 0, becomes the previous value of bit 1; the new value of bit 1 becomes the previous value of bit 2; and the new value of bit 2 becomes the previous value of bit 0. As depicted in Figure 3.6L, this action thus results in shifting values among the three nodes, mimicking the intuitive behavior of a shift register. Figure 3.6R shows the analogous flow of information for the action `R`. Finally, the action `F` should cause

[2]The potential disadvantage is that in degenerate, completely unstructured environments, the size of the update graph can be exponentially larger than the size of the state transition graph machine.

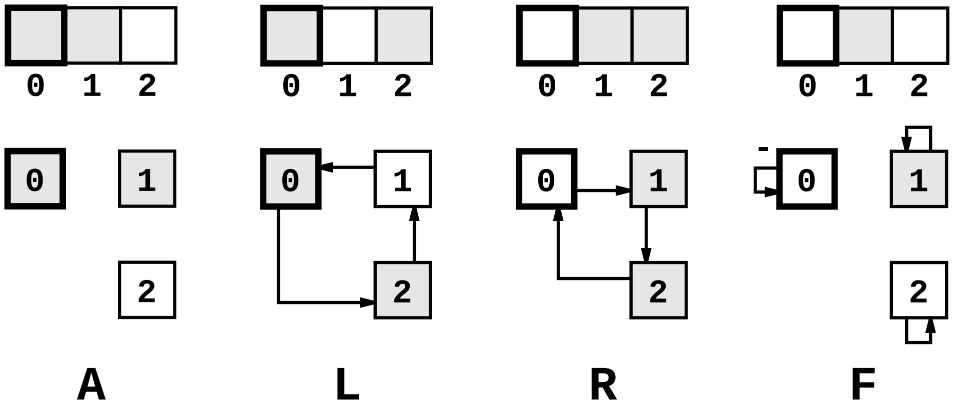


Figure 3.6. The Mechanics of the Update Graph. Panel A shows the 3-bit shift register and the update graph without links. White squares indicate a bit is on, and shaded squares indicate a bit is off. The observable leftmost bit is emphasized with a thicker box. Panels L, R, and F depict the result of performing the designated action in the shift register environment and the corresponding computation of the update graph.

the value of leftmost bit to be complemented while the values of the other bits remain unaffected (Figure 3.6F). In Figure 3.7I, the three sets of links from Panels L, R, and F of Figure 3.6 have been superimposed and have been labeled with their associated actions.

One final detail: the Rivest and Schapire 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 value of a bit and the other its complement (Figure 3.7C). Flipping thus involves exchanging the values of bit 0 and bit $\overline{0}$. Just as the values of bit 0, bit 1, and bit 2 must be shifted for the actions `L` and `R`, so must their complements.

Given the update graph in Figure 3.7C and the value of each node for 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 state transition graph.

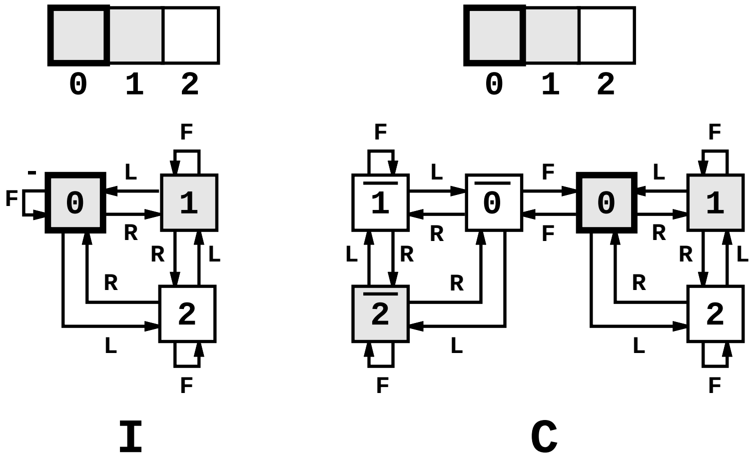


Figure 3.7. The Update Graph for the 3-Bit Shift Register. Panel I shows the complete "intuitive" update graph for the 3-bit shift register environment. Panel C shows the complete update graph (without complementation links) for the 3-bit shift register environment.

For every finite state environment, there exists a corresponding update graph. In fact, the update graph in Figure 3.7C might even be viewed as a distributed representation of the state transition graph in Figure 1.3. In the state transition graph, each environmental state is represented by *one* "active" node. In the update graph, each environmental state represented by a pattern of activity across the nodes.

A defining property of the update graph is that each node has exactly one incoming link for each action. This property, which we call the *one-input-per-action property*, has no intuitive justification based on the description of the update graph we have presented, but see Mozer and Bachrach [60] or Schapire [85] for an explanation. The one-input-per-action property clearly holds in Figure 3.7C; for example, note that bit 0 gets input from bit $\overline{0}$ for the action `F`, from bit 1 for `L`, and from bit 2 for `R`.

In the n-bit shift register environment, it happens that each node has exactly one outgoing link for each action (the *one-output-per-action property*). The one-input-per-action and one-output-per-action properties jointly hold for a class of environments

Rivest and Schapire call *permutation environments*, in which no information is ever lost (i.e., each action can be undone by some fixed sequence of actions). Update graphs of non-permutation environments, such as the car radio environment, do not possess the one-output-per-action property. Thus, one must not consider the one-output-per-action property to be a defining characteristic of the update graph; to do so would restrict the set of environments one could model.

3.2.1 The Rivest and Schapire Learning 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) inferring the values of each node in the update graph. Step (b) is relatively straightforward. Step (a) involves formulating explicit hypotheses about regularities in the environment. These hypotheses are concerned with whether two action sequences are equivalent in terms of their sensory outcome. For example, two equivalent action sequences in the 3-Bit Register environment are `LL` and `R` because shifting left twice yields the same value on the leftmost bit as shifting right once, no matter how the bits are initially set. By conducting experiments in the environment, the RS algorithm tests each hypothesis and uses the outcome to construct the update graph.

For permutation environments (described in the previous section), a special version of the RS algorithm is guaranteed to infer the environmental structure—within an acceptable margin of error—in a number of moves polynomial in the number of update graph nodes and the number of alternative actions. The general case of the RS algorithm makes fewer assumptions about the nature of the environment, but has no proof of probable correctness.

If only one hypothesis is tested at a time, the RS algorithm is "single minded" and does not make full use of the environmental feedback obtained. Consequently, Rivest and Schapire have developed heuristics to test multiple hypotheses at once. These heuristics can improve the efficiency of the basic algorithm by several orders of magnitude [74]. Because reasonable performance is achieved only by incorporating special—and potentially problematic—heuristics to make better use of environmental feedback, it seems worthwhile to explore alternative methods whose natural properties allow them to evaluate multiple hypotheses at once. We have pursued a connectionist approach, which has shown promising results in preliminary experiments as well as suggesting a different conceptualization of the update graph representation.

3.3 Connectionist Approach to Modeling Environments

SLUG is a connectionist network that performs Subsymbolic Learning of Update Graphs. Before the learning process itself can be described, however, we must first consider the desired outcome of learning. That is, what should SLUG look like following training if it is to behave as an update graph? Start by assuming one unit in SLUG for each node in the update graph. The activity level of the unit represents the value associated with the update graph node. Some of these units serve as "outputs" of SLUG. For example, in the 3-bit shift register environment, the output of SLUG is the unit that represents the value of the leftmost bit. In other environments (e.g., the little prince environment), there may be several sensations in which case several output units are required.

What is the analog of 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 units that represent the possible actions; these units act to multiplicatively gate the flow of activity between units in the update graph. Thus, when a particular action is to be performed, the corresponding action unit is turned on, and the connections that are gated by this action are enabled.

If the action units form a local representation, i.e., only one is turned on 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 (as shown in Figure 3.8). To predict the consequences of a particular action, say `F`, the weight matrix for `F` is simply plugged into the network and activity is allowed to propagate through the connections. In this manner, SLUG is dynamically rewired contingent on the current action. Figure 3.9 shows SLUG executing the action sequence `LFLR`.

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

$$\boldsymbol{x}(t) = W_{a(t)}\boldsymbol{x}(t-1), \tag{3.1}$$

where $a(t)$ is the action selected at time t, $W_{a(t)}$ is the weight matrix associated with this action, and $\boldsymbol{x}(t)$ is the activity vector that results from taking action $a(t)$. Assuming weight matrices 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 nonlinearities are introduced by the operation of rewiring SLUG based on the current action, the behavior of SLUG *in a single time step* is indeed linear. This is handy because it allows us to use tools of linear algebra to better understand the network's behavior. We elaborate on this point in Mozer and Bachrach [60].

3.4 Training SLUG

We have described how SLUG could be hand-wired to behave as an update graph, and now we turn to the procedure used to *learn* the appropriate connection strengths. For expository purposes, assume that the number of units in the update graph is known in advance. (This is not necessary, as we show below.) SLUG starts off with this many units, s of which are set aside to represent the sensations. These s units are the output units of the network; the remainder are hidden units. A set

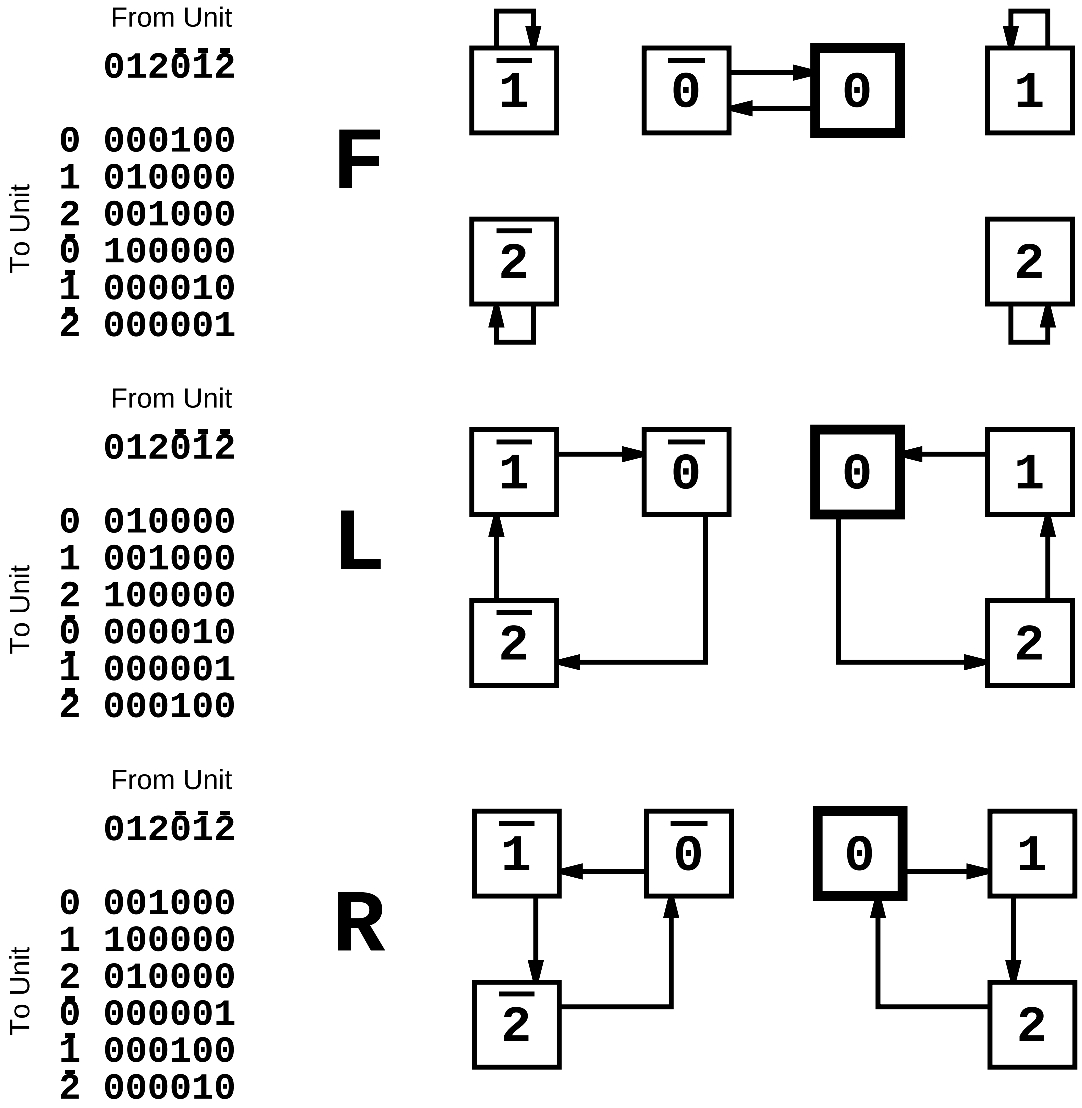


Figure 3.8. SLUG Depicted as a Separate Weight Matrix for Each Action. The left side of the figure shows the separate weight matrices, and the right side shows the connections having nonzero weights.

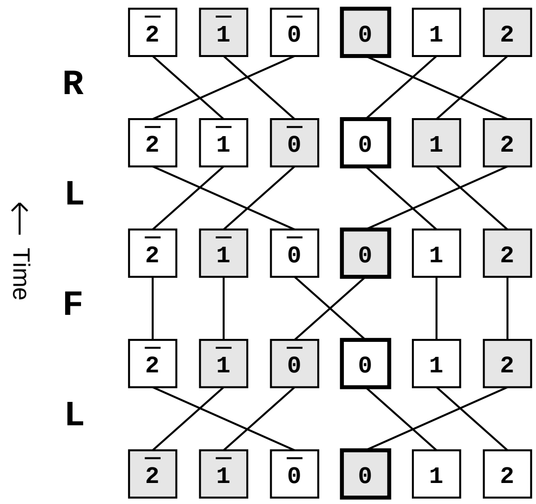


Figure 3.9. SLUG Is Dynamically Rewired Contingent on the Current Action. SLUG is shown executing the action sequence `LFLR`.

of weight matrices, $\{W_a\}$, is constructed—one per action—and initialized to random values. To train SLUG, random actions are performed on the environment and the resulting sensations are compared with those predicted by SLUG. The mismatch between observed and predicted sensations provides an error measure, and the weight matrices can be adjusted to minimize this error.

This procedure in no way guarantees that the resulting weights will possess the one-input-per-action property, which is required of the update graph connectivity matrices. We can try to achieve this property by performing gradient descent not in the $\{W_a\}$ directly, but in an underlying parameter space, $\{Z_a\}$, from which the weights are derived using a normalized exponential transform:

$$w_{aij} = \frac{e^{z_{aij}/T}}{\sum_k e^{z_{aik}/T}}, \tag{3.2}$$

where w_{aij} is the strength of connection to unit i from unit j for action a, z_{aij} is the corresponding underlying parameter, and T is a constant. This approach [3] permits

[3]This approach was suggested by the recent work of Bridle [12], Durbin [17], and Rumelhart [78], who have applied the normalized exponential transform to activity vectors; in contrast, we have applied it to weight vectors.

unconstrained gradient descent in $\{Z_a\}$ while constraining the w_{aij} to nonnegative values and

$$\sum_i w_{aij} = 1. \tag{3.3}$$

By gradually lowering T over time, the solution can be further constrained so that all but one incoming weight to a unit approaches zero. In practice, we have found that lowering T is unnecessary because solutions discovered with a fixed T essentially achieve the one-input-per-action property.

Below, we report on simulations of SLUG using this approach, which we will refer to as the version of the algorithm with *constrained weights*, and also simulations run with unconstrained weights. In the latter case,

$$w_{aij} = z_{aij}. \tag{3.4}$$

Details of the Training Procedure

Before the start of training, initial values of the z_{aij} are randomly selected from a uniform distribution in the range $[-1, 1]$, and the w_{aij} are derived therefrom. The activity vector $\boldsymbol{x}(0)$ is reset to $\mathbf{o}$. At each time t, the following sequence of events transpires:

1. An action, $a(t)$, is selected at random.

2. The weight matrix for that action, $W_{a(t)}$, is used to compute the activities at t, $\boldsymbol{x}(t)$, from the previous activities $\boldsymbol{x}(t-1)$.

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 SLUG (i.e., the activities of the units chosen to represent the sensations) to compute a measure of error.

5. The back-propagation algorithm (Rumelhart et al. [79]) is used to compute the derivative of the error with respect to each weight, $\partial E/\partial w_{aij}$. Because SLUG contains recurrent connections and back-propagation applies only to feedforward nets, the "unfolding-in-time" procedure of Rumelhart et al. is used. This procedure is based on the observation that any recurrent network can be transformed into a feedforward network with identical behavior, over a finite period of time. The feedforward net will have a layer of units corresponding to the units of SLUG at each time step: the top layer of the feedforward net represents $\boldsymbol{x}(t)$, the layer below represents $\boldsymbol{x}(t-1)$, below that $\boldsymbol{x}(t-2)$, and so on back to a layer that represents $\boldsymbol{x}(t-\tau)$. (We discuss the choice of τ below.) The weights feeding into the top layer are $W_{a(t)}$, the weights feeding into the layer below that are $W_{a(t-1)}$, and so forth. Back-propagation can be applied in a direct manner to this feedforward network. The error gradient for some action i, $\partial E/\partial W_i$, is computed by summing the back-propagated error derivatives over all layers l of the feedforward net in which the action corresponding to that layer, $a(l)$, is equal to i.

6. The error gradient in terms of the $\{W_a\}$ is transformed into a gradient in terms of the $\{Z_a\}$. With constrained weights (Equation 3.2), the relation is

$$\frac{\partial E}{\partial z_{aij}} = \frac{w_{aij}}{T}\left[\frac{\partial E}{\partial w_{aij}} - \sum_k \frac{\partial E}{\partial w_{aik}} w_{aik}\right] \tag{3.5}$$

With unconstrained weights (Equation 3.4), the mapping is simply the identity.

7. The $\{Z_a\}$ are updated by taking a step of size η down the error gradient:

$$\Delta z_{aij} = -\eta \frac{\partial E}{\partial z_{aij}}. \tag{3.6}$$

The $\{W_a\}$ are then recomputed from the new $\{Z_a\}$.

8. The temporal record of unit activities, $\boldsymbol{x}(t-i)$ for $i = 0 \ldots \tau$, which is maintained to permit back-propagation in time, is updated to reflect the new weights. This

involves recomputing the forward flow of activity from time $t - \tau$ to t for the hidden units. (The output units are unaffected because their values are forced, as described in the next step.)

9. The activities of the output units at time t, which represent the predicted sensations, are replaced by the observed sensations. This implements a form of *teacher forcing* (Williams and Zipser [106]). A consequence of teacher forcing is that the error derivative for weights feeding into the output units at times $t-1$, $t-2$, etc. should be set to zero in Step 5; that is, error should not be back-propagated from time t to output units at earlier times. It is not sensible to adjust the response properties of output units at some earlier time i to achieve the correct response at time t because the appropriate activation levels of these units have already been established by the sensations at time i.

Steps 5 and 8 require further elaboration. One 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 time steps does not help SLUG to discover a solution more quickly, although any fewer does indeed hamper learning. In the results described below, we arbitrarily set τ for a particular problem to one less than the number of nodes in the update graph solution of the problem. Informal experiments manipulating τ revealed that this was a fairly conservative choice. To avoid the issue of selecting a value for τ, one could instead use the on-line recurrent network training algorithm of Williams and Zipser [105].

To back-propagate error in time, a temporal record of unit activities is maintained. However, a problem arises with these activities following a weight update: the activities are no longer consistent with the weights—i.e., Equation 3.1 is violated. Because the error derivatives computed by back-propagation are exact only when Equation 3.1 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 afresh at the beginning of the next sequence. In the present situation, however, the sequence of actions does not terminate. We were thus forced to consider alternative means of ensuring consistency. One approach we tried involved updating the weights only after every, say, 25 time steps. Immediately following the update, the weights and activities are inconsistent, but after τ time 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 8 on the list above). To do this, we propagated the earliest activities in the temporal record, $\boldsymbol{x}(t-\tau)$, forward again to time t, using the updated weight matrices.[4]

The issue of consistency arises because at no point in time is SLUG instructed as to the state of the environment. That is, instead of being given an activity vector as input, part of SLUG'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 indicate that this strategy does not improve SLUG's rate of learning. One plausible explanation is the following. If we perform gradient descent in just

[4]Keeping the original value of $\boldsymbol{x}(t-\tau)$ is a somewhat arbitrary choice. Consistency can be achieved by propagating *any* value of $\boldsymbol{x}(t-\tau)$ forward in time, and there is no strong reason for believing $\boldsymbol{x}(t-\tau)$ is the appropriate value. We thus suggest two alternative schemes, but have not yet tested them. First, we might select $\boldsymbol{x}(t-\tau)$ such that the new $\boldsymbol{x}(t-i)$, $i = 0 \ldots \tau-1$, are as close as possible to the old values. Second, we might select $\boldsymbol{x}(t-\tau)$ 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.

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. 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 $+1$ (indicating that the leftmost bit of the shift register is *off* or *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. For other arguments concerning the advantages of symmetric activity levels see Stornetta and Huberman [91].

3.5 Results

Figure 3.10 shows the weights in SLUG for the 3-bit shift register environment at three stages of training. The "step" refers to how many moves SLUG has taken, or equivalently, how many times the weights have been updated. The bottom diagram in the figure corresponds to the update graph of Figure 3.7C. 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 `L`, 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 leg, and right arm—form a similar loop. For the action `R`, the same two loops are present but in the reverse direction. These two loops also appear in Figure 3.7C. For the action `F`, 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 bit 0 and bit $\overline{0}$ nodes of the Figure 3.7C.

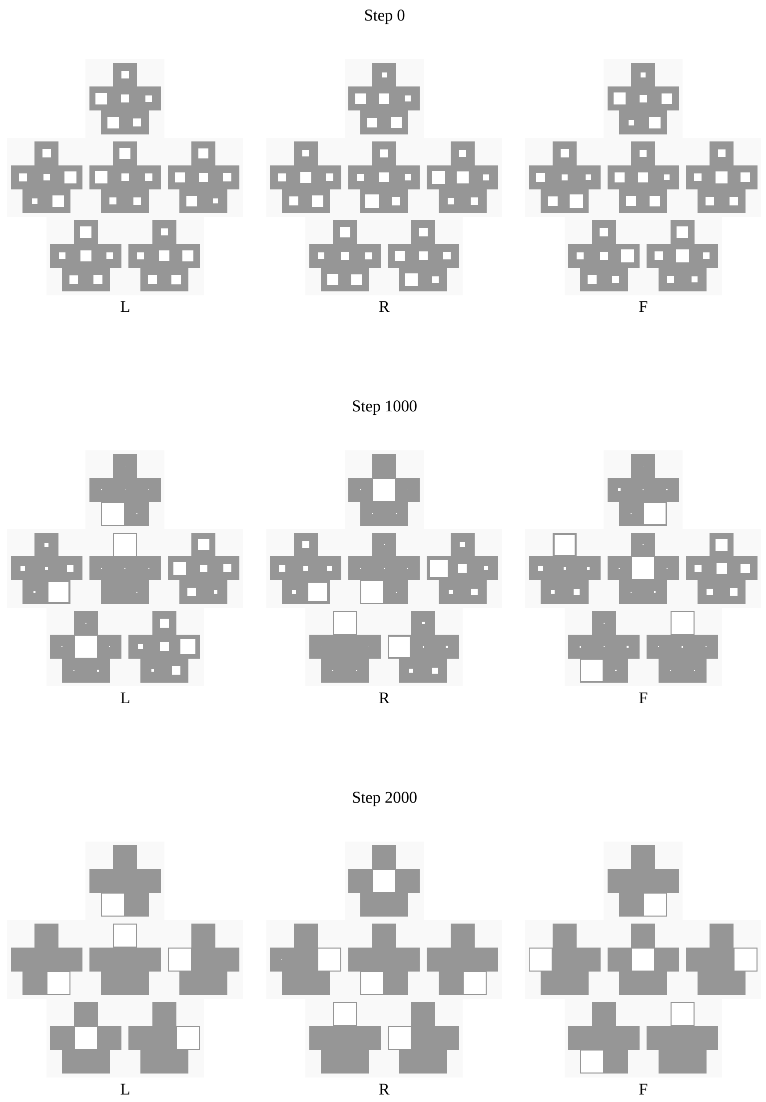


Figure 3.10. SLUG's Weights at Three Stages of Training for the 3-Bit Shift Register Environment. Step 0 reflects the initial random weights, Step 1000 reflects the weights midway through learning, and Step 2000 reflects the weights upon completion of learning. Each large diagram (with a light gray background) represents the weights corresponding to one of the three actions. Each small diagram contained within a large diagram (with a dark gray background) represents the connection strengths feeding into a particular unit for a particular action. There are six units, hence six small diagrams within each large diagram. 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, with the largest square having the value 1.

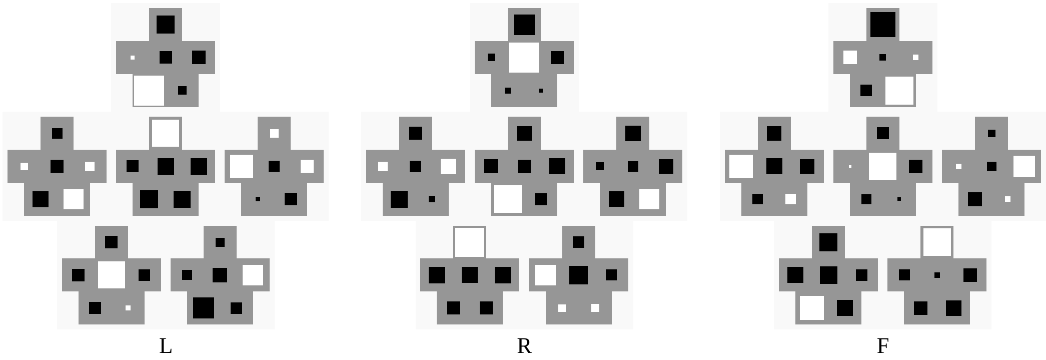


Figure 3.11. The Underlying z_{aij} Parameters. These parameters correspond to the weights in the bottom diagram of Figure 3.10. White squares indicate positive values, black squares negative values. The largest square represents a value of 4.5.

The weights depicted in Figure 3.10 are the w_{aij}'s. These weights are derived from the underlying parameters z_{aij}. The values of z_{aij} corresponding to the weights at Step 2000 are shown in Figure 3.11. The normalized exponential transform maps z_{aij} values in the range $-\infty \rightarrow +\infty$ to values in the range $0 \rightarrow 1$.

In addition to learning the update graph connectivity, SLUG has simultaneously learned the correct activity values associated with each node for the current state of

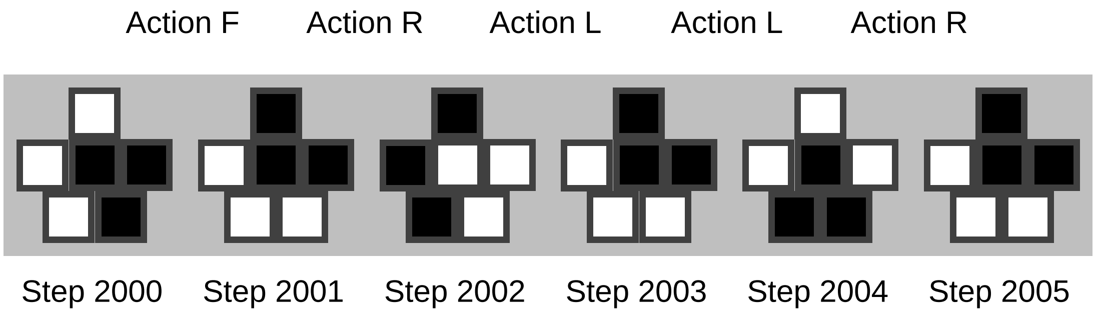


Figure 3.12. The Pattern of Activity in SLUG at Six Consecutive Time Steps. White squares indicate a unit with activity level of +1, black squares a unit with activity level -1. The arrangement of units in this figure matches the arrangement of weights in the previous two figures. Thus, the output unit is the protruding "head" of each diagram. The transition between two consecutive states arises from performing the action printed between the two states.

the environment. Armed with this information, SLUG can predict the outcome of any sequence of actions. Indeed, the prediction error drops to zero, causing learning to cease and SLUG to become completely stable. Figure 3.12 shows the evolution of activity in SLUG at six consecutive time steps. At Step 2000, the value of the leftmost bit is 1, as indicated by the white square in the diagram's head. When the action `F` is performed, the head and right leg exchange values while all other units maintain their current value. Between Steps 2001 and 2005, the action sequence `RLLR` is executed. Because shifting left and right are complementary actions, SLUG's state at Step 2005 is the same as at Step 2001.

SLUG shows two advantages over other connectionist approaches to learning finite-state automata (e.g., Elman [22]; Pollack [68]; Servan-Schreiber, Cleeremans, and McClelland [89]). First, because the ultimate weights and activities are discrete, SLUG can operate indefinitely with no degradation in its ability to predict the future. Other connectionist networks have the drawback that states are not represented discretely. Consequently, minor numerical imprecisions can accumulate, causing the networks to wander from one state to another. For example, Pollack [68] trained a network to accept strings of the regular language `1*` (i.e., any number of 1's). However, the network would accept strings containing a `0` if followed by a long enough string of `1`'s. The network would eventually "forget" that it was in the reject state. SLUG with constrained weights is not susceptible to this problem, as illustrated by the fact that the state at Step 2001 in Figure 3.12 is *exactly* the same as the state at Step 2005. The second advantage of SLUG over other connectionist approaches is that the network weights and activities can readily be interpreted—as an update graph. Consequently, the correctness of SLUG's solution can be assessed analytically.

To evaluate SLUG, we examined its performance as measured by the number of actions that must be executed before the outcomes of subsequent actions can be predicted, that is, before a perfect model of the environment has been constructed. One means of determining when SLUG has reached this criterion is to verify that the weights and activities indeed correspond to the correct update graph. Such a determination is not easy to automate; we instead opted for a somewhat simpler criterion: SLUG is considered to have learned a task by a given time step if the correct predictions are made for at least the next 2500 steps. In all simulation results we report in this paper, performance is measured by this criterion—the number of time steps before SLUG can correctly predict the sensations at the 2500 steps that follow. Because results vary from one run to another due to the random initial weights, we report the median performance over 25 replications of each simulation. On any replication, if SLUG was unable to converge on a solution within $100,000$ steps, we stopped the simulation and counted it as a failure.

Now for the bad news: SLUG succeeded in discovering the correct update graph for the 3-bit shift register environment on only 15 of 25 runs. The median number of steps taken by SLUG before training terminated—either by reaching the performance criterion on successful runs or by reaching the maximum number of steps on failed runs—was 6,428. Even on failed runs, however, SLUG does learn most of the update graph, as evidenced by the fact that after $10,000$ steps SLUG correctly predicts sensations with an average accuracy of 93%.

With unconstrained weights, SLUG's performance dramatically improves: SLUG reaches the performance criterion in a median of 298 steps. From our experiments, it appears that constraints on the weights seldom help SLUG discover a solution. In the remainder of this article, we therefore describe the performance of the version of SLUG with unconstrained weights. In Mozer and Bachrach [60], we consider why constraining the weights is harmful and possible remedies.

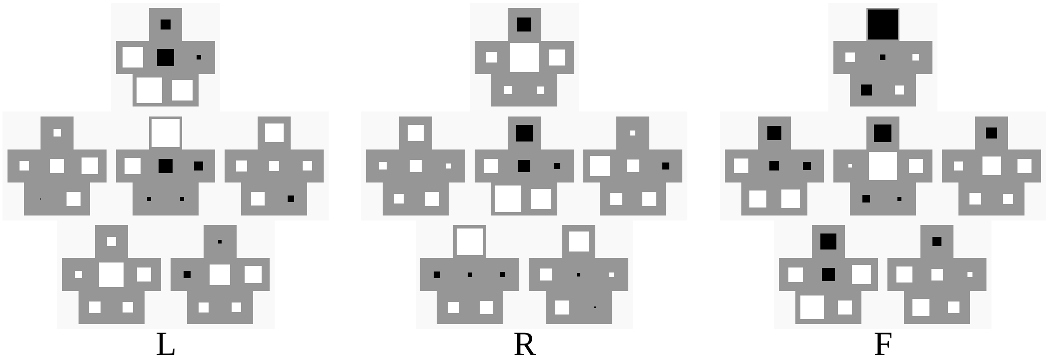


Figure 3.13. Weights Learned by SLUG with Six Units and Unconstrained Weights for the 3-Bit Shift Register Environment. White squares indicate positive weights, black negative.

3.6 Comparing SLUG and Other Connectionist Approaches

With unconstrained weights, SLUG suffers from the two drawbacks common to other connectionist approaches that we described earlier. First, slight inaccuracies in the continuous-valued weights and activities can cause degeneration of SLUG's predictive abilities in the distant future. Consequently, it is essential that SLUG converge on an exact solution. To assist towards this end, we scaled down the learning rate, η, as SLUG approached a solution. (That is, η was set in proportion to the mean squared prediction error.) The second advantage lost when weights are unconstrained is that, although SLUG learns effectively, the resulting weight matrices, which contains a collection of positive and negative weights of varying magnitudes, is not readily interpretable (see Figure 3.13).

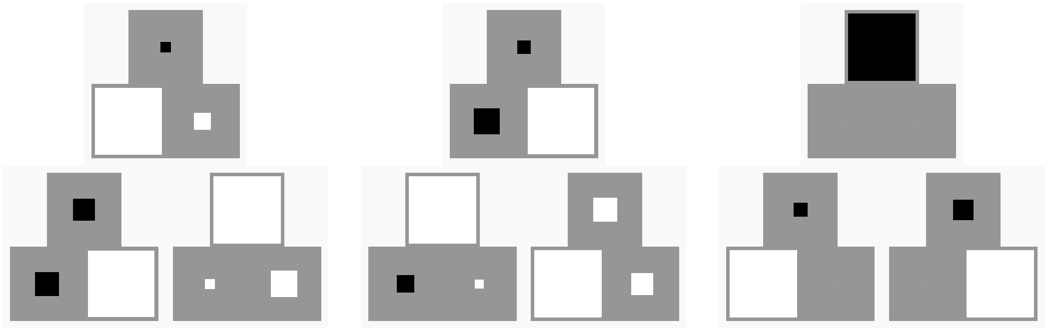
The loss of these two benefits might seem to indicate that SLUG is no better than other connectionist approaches to the problem of inducing finite-state environments, but comparisons of SLUG and a conventional connectionist recurrent architecture indicate that SLUG achieves far superior performance. The "conventional" architecture we tested was a three layer network consisting of *input, hidden, and output* units, with feedforward connections from input to hidden and hidden to output layers as well as recurrent connections within the hidden layer (Bachrach [3]; Elman [22]; Mozer [56]; Servan-Schreiber et al. [88]). Input units represent the current sensations and an action, output units represent the sensations predicted following the action. Our experiments using the conventional architecture were spectacularly unsuccessful. We were unable to get the conventional architecture to learn the 3-bit shift register environment on even one run, despite our best efforts at varying the number of hidden units, the number of steps τ through which error was back-propagated, and learning rates in every conceivable combination. We lowered our sights somewhat and attempted to train the conventional architecture on a simpler environment, the little prince environment. The conventional architecture was able to learn this environment on 24 of 25 replications, with a median of $27,867$ steps until the prediction task was mastered. In contrast, SLUG learned perfectly on every replication and in a median of 91 steps.

Besides testing the conventional architecture on the environment inference problem, we also tested SLUG on problems studied using the conventional architecture.[5] Servan-Schreiber et al. [88] have worked on the problem of inferring regular languages from examples, which is formally identical to the environment inference problem except that an exploration strategy is unnecessary because sample strings are given to the system. They trained the conventional architecture on strings of a language. After $200,000$ sample strings, the network was able to reject invalid strings. In contrast, SLUG required a mere 150 training examples on average before it mastered the acceptability judgement task.[6]

The fact that SLUG achieves considerably better performance than conventional connectionist approaches justifies its nonintuitive network architecture, dynamics,

[5]This comparison was performed in collaboration with Paul Smolensky.

[6]The training procedure was slightly different for the two architectures. Servan-Schreiber et al.'s network was shown positive examples only, and was trained to predict the next element in a sequence. SLUG was shown both positive and negative examples, and was trained to accept or reject an entire string. We see no principled reason why one task would be more difficult than the other.



L R F

Figure 3.14. Weights Learned by SLUG with Three Units and Unconstrained Weights for the 3-Bit Shift Register Environment.

and training procedure.[7] Thus, Rivest and Schapire's update graph representation, which motivated the architecture of SLUG, has proven beneficial, even if what SLUG (with unconstrained weights) learns does not correspond exactly to an update graph.

Interpreting unconstrained solutions discovered by SLUG

One reason why the final weights in SLUG are difficult to interpret in the case of the 3-bit shift register environment is that SLUG 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 3.7I. 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. SLUG's solutions are much easier to understand when the network consists of only three units. Figure 3.14 depicts one such solution, which approximates the graph in Figure 3.7I. Ignoring the connections of small magnitude, the three units are connected in a clockwise loop for action `L`, a counterclockwise loop for action `R`, and the output unit toggles its value while the two internal units maintain their values for action `F`. Although a

[7]It remains to be seen which properties of SLUG that differ from the conventional connectionist approach are crucial for SLUG's improved performance. One key property is undoubtably the gating connections between actions and state units. See Giles et al. [26] for a related architecture that also shares this property.

comparison of Figures 3.14 and 3.7I might seem to indicate that the connections of small magnitude in Figure 3.14 introduce noise, this is in fact not the case. The solution discovered by SLUG is exact—it predicts sensations accurately because the effects of these connections exactly cancel out. SLUG also discovers other solutions in which two of the three connections in the three-unit loop are negative, one negation canceling out the effect of the other. Allowing complementation links can halve the number of update graph nodes required for many environments. This is one fairly direct extension of Rivest and Schapire's update graph formalism that SLUG suggests.

Treating the update graph as matrices of connection strengths has suggested another generalization of the update graph formalism. Because SLUG 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. Thus, one can view the Rivest and Schapire update graph formalism as one example of a much larger class of equivalent solutions that can be embodied in a connectionist network. While many of these solutions do not obey constraints imposed by a symbolic description (e.g., all-or-none links between nodes), they do yield equivalent behavior. By relaxing the symbolic constraints, the connectionist representation allows far greater flexibility in expressing potential solutions. See Mozer and Bachrach [60] for a further elaboration of this point.

Comparing SLUG and the RS Algorithm

Table 3.1 compares the performance of the RS algorithm and SLUG for a sampling of environments. [8] In these simulations, we ran the version of SLUG with unconstrained weights. All runs of SLUG eventually converged on an adequate set of weights. The learning rates used in our simulations were adjusted dynamically every 100 steps by averaging the current learning rate with a value proportional to

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

Table 3.1. Number of Steps Required to Learn Update Graph.

Environment	Size of Update Graph	Median Number of Actions	
		The RS Algorithm	SLUG
Little Prince	4	200	91
Car Radio	6	27,695	8,167
3-Bit Shift Register	8	408	298
4-Bit Shift Register	9	1,388	1,509
5×5 Grid	27	583,195	fails
32-Bit Shift Register	32	52,436	fails
6×6 Checkerboard	36	96,041	8,142

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. Including momentum in the back-propagation algorithm did not appear to help significantly.[9]

In simple environments, SLUG can outperform the RS algorithm. These results are quite surprising when considering that the action sequence used to train SLUG is generated at random, in contrast to the RS algorithm, which involves a strategy for exploring the environment. We conjecture that SLUG does as well as it does because it considers and updates many hypotheses in parallel at each time step. That is, after the outcome of a single action is observed, nearly all weights in SLUG are adjusted simultaneously. In contrast, the RS algorithm requires special-purpose heuristics that are not necessarily robust in order to test multiple hypotheses at once.

A further example of SLUG's parallelism is that it learns the update graph structure at the same time as the appropriate unit activations, whereas the RS algorithm approaches the two tasks sequentially. During learning, SLUG continually

[9]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. We experimented with the RS algorithm, and without expertise in parameter tweaking, were unable to obtain performance in the same range as the measures reported by Rivest and Schapire.

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.

In complex environments—ones in which the number of nodes in the update graph is quite large and the number of distinguishing environmental sensations is relatively small—SLUG does poorly. Two examples of such, the 32-bit shift register environment and the 5×5 grid environment, cannot be learned by SLUG whereas the RS algorithm succeeds. An intelligent exploration strategy seems necessary in complex environments: with a random exploration strategy, the time required to move from one state to a distant state becomes so great that links between the states cannot be established.

The 32-bit shift register environment and the 5×5 grid environment are extreme: all locations are identical and the available sensory information is meager. Such environments are quite unlike natural environments, which provide a relative abundance of sensory information to distinguish among environmental states. SLUG performs much better when more information about the environment can be sensed directly. For example, learning the 32-bit shift register environment is trivial if SLUG is able to sense the values of all 32 bits at once (the median number of steps to learn is only 1209). The checkerboard environment is another environment as large as the 32-bit shift register environment in terms of the number of nodes SLUG requires, but it is much easier to learn because of the rich sensory information.

Noisy environments. The RS algorithm originally could not handle environments with unreliable sensations, although a variant of the algorithm has recently been designed to overcome this limitation (Schapire [84]). In contrast, SLUG, like most connectionist systems, deals naturally with noise in that SLUG's ability to predict degrades gracefully in the presence of noise. To illustrate this point, we trained

SLUG on a version of the little prince environment in which sensations are registered incorrectly 10% of the time. SLUG was still able to learn the update graph. However, to train SLUG properly in noisy environments, we needed to alter the training procedure slightly, in particular, the step in which the observed sensations replace SLUG's predicted sensations. The problem is that if the observed sensations are incorrectly registered, the values of nodes in the network will be disrupted, and SLUG will require a series of noise-free steps to recover. Thus, we used a procedure in which the predicted sensations were not completely replaced by the observed sensations, but rather some average of the two was computed, according to the formula

$$(1-\omega)p_i + \omega o_i, \tag{3.7}$$

where p_i and o_i are the predicted and observed values of sensation i, and ω is a weighting factor. [10] The value of ω is a function of SLUG's performance level, as measured by the mean proportion of prediction errors. If SLUG is making relatively few errors, it has learned the correct environment model and ω should be set to 0—the observed value should be ignored because it is occasionally incorrect. However, if SLUG is performing poorly, the prediction p_i should not be relied upon, and ω should have a value closer to 1. Rather than computing ω as a function of SLUG's performance, another possibility is to adjust ω via gradient descent in the overall error measure.

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 3.1 are obtained when the RS algorithm knows in advance exactly how many nodes are required. The algorithm fails if it is given an upper bound less than the required number of nodes, and performance—measured as the number of steps required to discover the solution—degrades as the upper bound increases above the

[10] A more principled but computationally more expensive technique for updating the predicted sensations can be derived using Kalman filtering theory (Gelb [25]).

Table 3.2. Number of Steps Required to Learn Update Graph as the Number of Units in SLUG Is Varied.

Units in SLUG	Median 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

required number. SLUG will also fail to learn perfectly 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 3.2 presents the median number of steps required to learn the 4-bit shift register environment as the number of units in SLUG (with unconstrained weights) is varied. Although performance is independent of the number of units here, extraneous units greatly *improve* performance when the weights are constrained: only 4 of 25 replications of the 4-bit shift register environment simulation with 8 units and constrained weights successfully learned the update graph, whereas 19 of 25 replications succeeded when 16 units were used.

3.7 Limitations of SLUG

The connectionist approach to the problem of inferring the structure of a finite-state environment 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, SLUG has no hope of scaling to complex environments. An intelligent strategy could potentially be incorporated to encourage SLUG to explore unfamiliar regions of the environment. One approach we are currently investigating, based on the work of Cohn et al. [14], is to have SLUG select actions that result in maximal uncertainty in its predictions, where uncertainty is defined as how far a predicted sensation is from one of the discrete sensation values. Second, our greatest successes have occurred when we allowed SLUG to discover solutions that are not necessarily isomorphic to an update graph. One virtue of the update graph formalism is that it is relatively easy to interpret; the same cannot generally be said of the continuous-valued weight matrices discovered by SLUG. However, as we discuss in Mozer and Bachrach [60], there is promise of developing methods for transforming the large class of formally equivalent solutions available to SLUG into the more localist update graph formalism to facilitate interpretation.