Chapter 5

Reinforcement

There is no programmer for the brain but instead it can guess programs that are evaluated in terms of how they fulfill the animal's needs. A neurotransmitter dopamine *is the currency used to do this. Dopamine is a secondary reward that estimates primary reward. This introduces a delicacy into the scoring system that can lead to addictive behaviors being rewarded highly.*

Everyday behavior requires that the brain comes up with programs. These are sequences of abstract actions to achieve goals. Most of these involve physical behaviors wherein the body is directed to do something. Consider making a cup of tea. The following program should work:

get the kettle
fill it with water
put the water on to boil
get a cup
get a tea bag
put the tea bag in the cup
when the water is boiling, fill the cup
discard the tea bag
add milk

In the state-action characterization of programs, the above description lists the actions and leaves the state necessary before each action implicit. But in the brain both would have to be explicitly coded in some way. Then this recipe for tea-making would be carried out by the body.

But how do we come up with these recipes? For many such things we can be taught by getting verbal instructions or having someone else demonstrate. But for a vast amount of behaviors we are left to our own devices and have to generate the recipes alone. Ultimately we are programmed to propagate our genes but primate life is long and complicated and it is not always clear how the behaviors of the moment are related to this ultimate goal. Nonetheless this

is the problem we solve. Figure **??** shows the situation in the abstract with a graph of states and actions that represent a model of the world. This graph is the source of particular programs that are found by searching possible paths in this space of alternative programs.

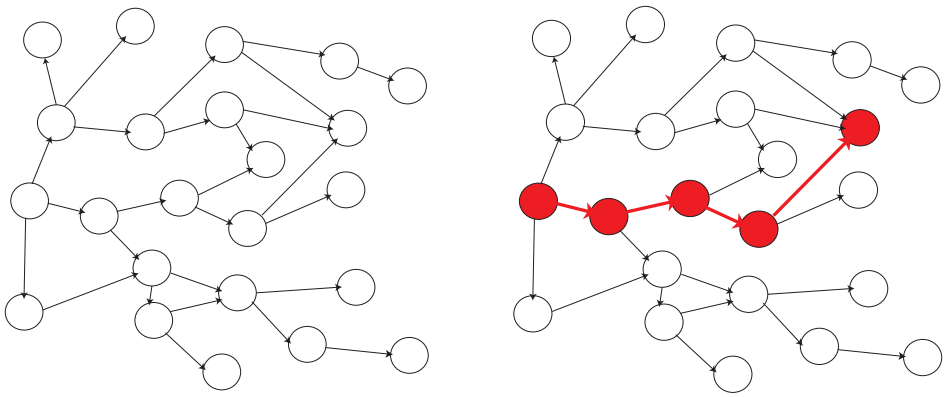

Figure 5.1: A) The problem of finding a behavioral program can be cast abstractly as searching through a maze of states and actions. B) A successful program can be represented as a specific path through the space.

It would be easier to solve if we had some way of rating the value of doing things. The programs could be scored and the best ones picked. However getting the information needed for this rating system is not straightforward at all. Consider eating an apple. It is easy enough to design an internal system that measures the caloric value of food and reduces appetite when 'full.' So conceptually at least we can design a system for doing this. But there is a problem here and that is the fact that the reward is delayed. We have to eat the apple first to know its value exactly. In this particular case we can imagine that the relatively small delay between eating and scoring is handled somehow, but for longer delays this turns out to be a big problem. Consider the value of foraging to find an apple? We could off course set off and just hope that apples are on our route, but what if there are multiple possible routes with probabilities of success dependent on current conditions? Certainly if the effort that would need to be expended exceeds the value of the apple then one shouldn't set off in the first place. Unlike the first calculation for eating an apple, this harder calculation has to be done much more abstractly. We have to evaluate a plan for getting the apple that involves evaluating these possibly risky uncertain steps. At least for the apple the end product is something of concrete value. Very abstract programs with uncertain consequences are much harder to evaluate.

The phrase "evaulate a plan" is very compact because it involves two very different but important steps: generating a plan from a repertoire of possible steps and scoring the plan. As far as our current understanding goes, the first part, plan generation, is the most difficult part and the details of how it is done contain many open questions. We know how to search through possibilities when there is a modest number, but when we have to choose from a vast number of plausible alternatives, there is no general agreement on how to narrow down the possibilities. It could be the case that we are stuck with thinking of plans that are 'close' in some way to previous successful plans. This would allow them to be discovered by brute force searching methods. In this case the alternative ways of doing the task are enumerable in some way and we systematically try them out. In fact there is a very good way to conduct this search in the case where their is a deterministic model of the search space. If we can depend on the actions from a given state taking us to another specific state, then we can exploit this property by starting from the goal and working backwards as shown in Fig 5.2.

Another aspect that has many loose ends is the way actions in a plan are coded. In this chapter the specification of action will be very abstract. In tea making, 'put the water on to boil' is actually a complex action involving many sub-actions. But for the purpose of planning, we will assume it is a primitive choice that the body knows how to execute. In fact the body has enormous amounts of detail encoded in the spinal cord that can direct the execution of actions including complex reflexive contingencies. These details are important and crucial to understanding behavior; so much so that they will be described separately in the next chapter.

The second step, scoring a plan, is better understood. Whereas something like an apple has primary reward - it has nutritional value for keeping an animal alive - when an animal has an elaborate plan to get the reward, the reward has not happened yet and may be far in the future. Yet to choose the plan, the animal has to have some representation of the expectation of getting reward. In other words, there needs to be an internal scoring mechanism that evaluates possible behaviors. In Chapter 1 we pointed out that there is good agreement on the brain's scoring mechanism thanks to Nobelists Greengard, Kandel, and Carlsson. They shared the 2001 prize in Biology for their work in pinpointing dopamine as the brain's major reward signal. Dopaminergic neurons are in the brainstem and enervate vast amounts of cortex, selectively emphasizing the most abstract areas. In honor of Europe's unified currency, the euro, i term dopamine the brain's "neuro." This vast internal dopamine-dispensing network has access to the brain's representations of potential plans of action and can evaluate them in terms of how many neuros they are worth. Off course many technical problems have to be solved in order for this to happen, but in broad outline there is an emerging consensus that the brain's way of solving this problem takes this route. Perhaps it is not surprising that most drugs work by breaking into the brain's dopamine storehouse. Why work all week for a modest amount of neuros when you can ingest some cocaine and effectively rob the neuro bank? Drug addiction in fact highlights the fact that the process of generating

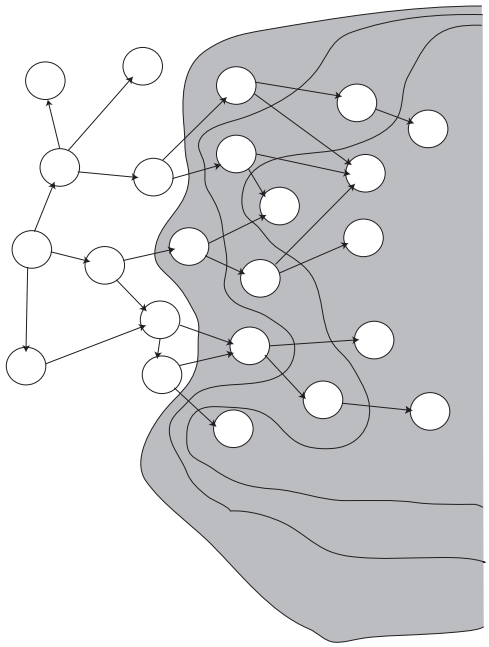

Figure 5.2: In the case of a reliable model there is an efficient way to enumerate the possiblities. Start from the possible end states from a given state and back up to the state that you came from. record the best action from the penultimate state. Next, repeat this process to the prior states. When the process reaches the initial state, the best program is known.

programs to control behaviors might be a mechanical process where programs are syetmatically evaluated. A program can be one that is socially destructive but from the perspective of the animal's internal bookkeeping system, it seams like the best thing to do.

5.1 Monkeys Code Secondary Reward

Evidence is mounting from research that monkeys use dopamine to compute the score of a plan. Some of the most compelling evidence comes from Wolfram Schultz's experiments with monkeys[2]. He is able to record the responses from neurons in the Substantia Nigra, a brainstem area rich in dopaminergic neurons, while monkeys are engaged in a task that requires them to synthesize a program. See BOX DOPAMINE REWARDS.

DOPAMINE REWARDS

Initially monkeys are trained to reach into a box for a piece of apple. In these cases, immediately after the apple is obtained the dopamine neurons fire, signaling that, as far as the brain's understanding of the plan is concerned, reward was obtained. Now if the task is increased in difficulty by adding a trigger signal. So the monkey has to wait for a light signal to change before she can begin her reach. Surprisingly you can see that the reward is now handed out at the time the trigger signal changes, in other words at the beginning of the program. The monkey has apparently encapsulated the new sequence of instructions and given it a score when it is initiated. Making the task still more difficult, the protocol is changed so that a light appears before the trigger that announces whether or not an apple piece is in the box. Now the reward signal shifts to the time of this signal, or the beginning of the more elaborate program. The monkey doles out reward based on its expected consequences.

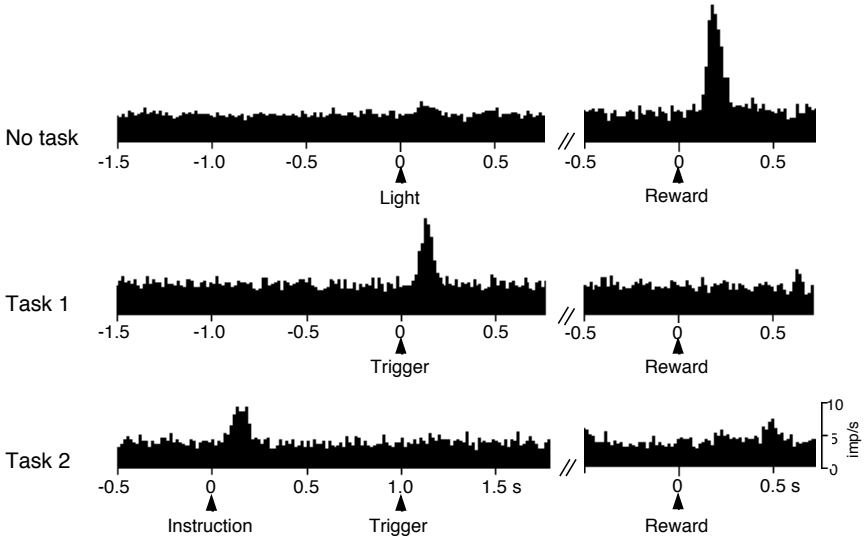

Figure 5.3: Histograms recored from the substantia nigra of a monkey. The substantia nigra neurons deal out dopamine rewards for behavioral programs. Shown are recordings from the same cell under three different conditions. (*No task*) the monkey reaches into a box for a piece of apple. (*Task 1*) The monkey has to wait for a light trigger before reaching. (*Task 1*) The monkey is given an instruction that indicates whether or not the box will contain an apple.

If you examined Figure 5.3 closely, you might have already noticed that the dopamine signal is lessened depending on how far in the future the actual reward is. Researchers are excited about this as it may be evidence that the neurons are using *discounting*. The idea behind discounting is that future rewards are

uncertain as the world has a way of intervening to thwart your goals: You had an important work assignment due but were sidelined by a case of flu. As we will show in a minute, the right thing to do mathematically is to pick a number which you use as you personal rating of how much you value rewards in the far future. The accepted notation is to use the symbol γ for the discount and the values used range from 1(no discount) to 0(complete discount). If you are addicted to cocaine your value of γ presumably is nearer zero: to have any value reward must be immediate.

5.2 Reinforcement learning algorithms

At this point we can talk about how the brain's programs might be learned. We will have to assume a lot, but we can still make some headway. The biggest thing we will assume is that we can describe the venue of such programs in terms of states that the animal is in and possible actions that she can take from those states. We are in fact pretty sure that the states and actions are not exactly those coded in the brain, but the intent is that they mirror in some essential sense the actual states that an animal might use. Once we assume this much though, we can say rather precisely how to go about scoring those states and generating programs.

To see the computations involved, consider the very simple example of a rat following a maze shown in Figure 5.4. The rat's formal problem is to learn what to do for this maze. Of course what to do in this case is to follow the maze in a way to get cheese (Don't think this is so easy - a book on management techniques titled *Who moved my cheese?* sold thousands of copies in airports). The protocol in our case is that the rat is somehow returned to this maze again and again so she had best save a map of what to do as a *policy* that will specify for each state the best action to move the rat toward the goal. The actions that the rat can use are to try to move in one of four compass directions, as shown in Figure 5.4. If there is a wall in that direction, the action has no effect, leaving the rat in the state (maze square) from which the action was taken. You can form a cartoon image of the poor rat bumping into the wall and tottering backward if this helps keep the dry mathematics at bay. If there is no wall then the rat progresses to the next square and consults the table again. Cartoon rats are much smarter than there living counterparts and it helps in this case because the rat knows where she is by reading numbers written on the floor. These are enough to define states.

These effects for the example maze are shown as a *transition function* $T(x_t, u_t)$ in Table 5.1. The final part of the problem specification is the reward function. The rat gets one hundred neuros for getting the cheese, otherwise at any point along the way she gets nothing. So the reward schedule is:

$$R(x_t) = \begin{cases} 100 & x_t = 10 \\ 0 & \text{otherwise} \end{cases}$$

Here x_t is of course the state and $R(x_t)$ is the reward given out in that state.

Table 5.1: The transition function for the maze problem.

	1	2	3	4	5	6	7	8	9	10
N	2	2	3	4	5	4	6	8	9	8
E	1	3	4	5	5	6	8	9	9	10
S	1	1	3	6	5	7	7	10	9	10
W	1	2	2	3	4	6	7	7	8	10

The result of reinforcement learning is an optimal policy, shown in pictorial form in Figure 5.4 at the bottom. For every state, the policy stores what direction to head from that state. By repeatedly consulting the tale, the rat gets to the goal. The goal of a reinforcement learning algorithm is to calculate this policy. If you know anything about rats at all, you might be perplexed by this example, as real rats have exquisite senses of smell (imagine what a human would look like if their noses where in proportion to that of a rat!). So to make the problem make sense we'll have to assume that our pretend rat has been genetically engineered to be without this capability. Thinking about smell is of great help though, because it allows you to image what the solution found by reinforcement looks like. At every state the value of following the policy is computed and that value gets larger as the rat gets closer.

To be biologically plausible, it is best if our policy learning algorithm only has to make local improvements. Another reason is the all the rest of the program machinery also has this local property: To know what to do next, you only have to know the current state. What you did many time steps ago is inconsequential. This *Markov* property allows actions to be chosen based only on knowledge about the current state and the states reachable by taking the actions available at that state.

5.3 Learning

You might have already had the thought that a smart rat - remember these are cartoon rats - will solve this problem by making a map of the maze and going back wards from the goal. In this case the algorithm would efficiently visit each mental state only once. You can do this as an exercise now: back up square by square lowering the value of each square by the value of the square you came from times the discount factor. As we already discussed, if in fact you have a model this is the right thing to do. But in more complicated real-life situations we assume that you would not have a model either because the problem is to complicated to model or that your actions are uncertain in this case. Thus, in reinforcement learning, an assumption is that the rat is repeatedly plonked down in the maze and is forced to wander around. Such searches are called *forward iterations*. Again, forward iterations are necessary to learn the control policy because the transition function of the system can not be known with

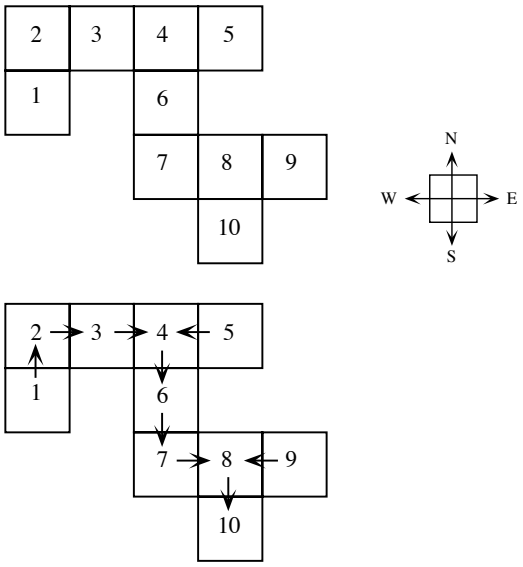

Figure 5.4: (*top*) A simple maze illustrates the mechanics of reinforcement learning. (*middle*) The transition function. For each state listed in the top row, the effects of the four possible actions are given in terms of the resultant state. (*bottom*) The optimal policy for the maze problem. From every state the arrow shows the direction in which to go in order to maximize reward.

certainty but must be learned by experiments. The only way to find out what happens in a state is to pick a control, execute it, and see where you end up. It turns out that this brute force strategy works! In fact the rationale for its success can be formalized in the *policy improvement theorem*.

The policy improvement theorem requires a specification of a policy and an additional bookkeeping mechanism to keep track of the value of states and actions. Figure 5.5 depicts the basic notation. Let $f(\boldsymbol{x})$ be the policy for every state $\boldsymbol{x}$. The policy is simply the action to take in state $\boldsymbol{x}$. Denote $V_f(\boldsymbol{x})$ as the value of the policy. This is the expected reward for following policy f from state $\boldsymbol{x}$. Now let $Q(\boldsymbol{x}, \boldsymbol{u})$ be the *action-value* function, or Q-function, for policy f. This is the expected return of starting in state $\boldsymbol{x}$, taking action u, and then following policy f thereafter. The value function is related to the Q-function by

$$V_f(\boldsymbol{x}) = \max_{\boldsymbol{u}}[Q(\boldsymbol{x}, \boldsymbol{u})]$$

In other words, the value of a state is the value of the best action available from that state. Thus it is easy to calculate V from Q.

Now we are ready for the theorem. Suppose you have a policy, $f(\boldsymbol{x})$ that tells you which action to pick for every state, but is not the best policy. Now

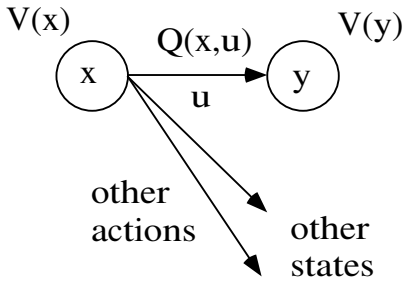

Figure 5.5: The basic vocabulary for reinforcement learning algorithms. The value function V rates each state. The action-value function $Q(\mathbf{x}, \mathbf{u})$ rates the value of taking action $\mathbf{u}$ from state $\mathbf{x}$.

consider a new policy $g(\boldsymbol{x})$ that chooses the best action based on the Q-values, takes that action, and subsequently follows policy f. If that policy is better than f, then it is always better to choose g because after you arrive at the next state, you can apply the same reasoning to this new state and you will wind up choosing g again. In coarse terms, what this analysis is telling us, is that we can repeatedly arbitrarily improve the policy locally and get increasingly close to the best possible policy.

The idea of choosing an action from a state, executing that action and repeating the process from the state you wind up in has been formalized as *temporal difference learning*. The best way to understand temporal difference learning is to examine its learning rule. If $\boldsymbol{x}_t$ and $\boldsymbol{x}_{t+1}$ are two states where the latter is arrived at by taking an action from the former, we want to adjust the relative values of these two states so that they are related by the discount factor. That is, we want

$$V(\boldsymbol{x}_t) = \gamma V(\boldsymbol{x}_{t+1})$$

so lets use

$$\Delta V(\boldsymbol{x}_t) = \alpha[V(\boldsymbol{x}_t) - \gamma V(\boldsymbol{x}_{t+1})]$$

After a lot of experimenting the values will settle down to the point where they are all correctly related by the discount factor.

5.4 Using Error to Set Synapses

What we will do at this point is take some time to put together the scoring ideas from this Chapter with the representation ideas from the previous Chapter. Interestingly, these ideas come up in the context of a neural network invented by Gerald Tesauro that learns to play the game of backgammon[**?**]. Backgammon

is a board game with two opponents that must advance their pieces using the roll of dice. It is basically a race between the two to see who can get their pieces around the board first and win. But, when we get into the rules in more detail, we will see the subtle interactions between the pieces that make the game interesting and difficult. Tesauro's network learned to play Backgammon using the reinforcement learning algorithm just described. In fact it learned to play so well that it plays on par with the world's best players.

In order to play backgammon the value of different board configurations must be evaluated. If the board defines the state, we need to know how to estimate the value of that state in neuros. For the moment we will assume we know the value and want to train the synapses to reproduce it. The next section shows how to generate the targets online.

In Chapter 3 you saw that the synapses of a neuron could be set by requiring that they reconstruct the input as best they could when limited to using a small set of active neurons at a time. The setting there was neurons that were very close to the input stage, the LGN. But what about the other end of the abstraction hierarchy? Could we not have a circuit there that minimizes some abstract quantity such as reward expectation? We can and to do so we will introduce an algorithm that is somewhat similar to the previous one but that has this different perspective.

Consider a single-layered network with two inputs and one output, as shown in Figure 5.6. The network uses the same linear summation activation function that was used in Chapter 4. Thus the output is determined by

$$y = \boldsymbol{w} \cdot \boldsymbol{x}$$

The problem is to "train" the network to produce an output y for an input pattern $\boldsymbol{x}$ such that the error between the desired output and the actual output y produced by the input pattern acting on the synapses is minimized. Of course there would usually be several patterns, but this would complicate the notation a little bit so we will stick with one pattern. If there are more than one pattern, the contributions from each pattern can be added together.

As before the variables in the problem are the synaptic weights in the network. They are to be adjusted to minimize the error. Define an error function

$$E(\boldsymbol{w}) = \frac{1}{2}(y^d - y)^2$$

which is the same as

$$E(\boldsymbol{w}) = \frac{1}{2}\left(y^d - \boldsymbol{w} \cdot \boldsymbol{x}\right)^2$$

One way to minimize the cost function $E(\boldsymbol{w})$ is to use gradient search. Starting from an initial guess, use the derivative $\frac{\partial E(\boldsymbol{w})}{\partial w_k}$ to successively improve the guess.

$$\Delta w_k = -\alpha \frac{\partial E(\boldsymbol{w})}{\partial w_k}$$

The parameter α controls the size of the change at each step. The changes are discontinued when the improvements become very small. One way of measuring

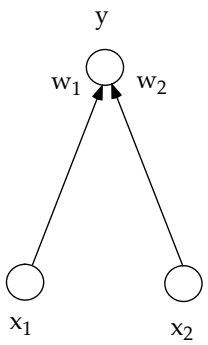

Figure 5.6: A simple case for optimization: a function of one variable minimizes error.

the improvement is $\sum_k \|\Delta w_k\| < \varepsilon$. For an appropriately chosen *alpha* this procedure will converge to a *local minimum* of $E(\boldsymbol{w})$.

Since E is a function of other variables besides $\boldsymbol{w}$, the *partial derivative* is used in the preceding equation to denote that we only want to consider the variations directly caused by changes in $\boldsymbol{w}$:

$$\frac{\partial E(\boldsymbol{w})}{\partial w_k} = -(y^d - y)x_k$$

To derive this we had to use the chain rule from calculus. First let $u = y^d - \boldsymbol{w} \cdot \boldsymbol{x}$ Then $\frac{\partial E(\boldsymbol{w})}{\partial w_k} = \frac{\partial E(\boldsymbol{w})}{\partial u} \frac{\partial u}{\partial y} \frac{\partial y}{\partial w_k}$. The first of these terms is just u, the second -1 and the third x_k. This particular weights is known as the *Widrow-Hoff learning rule*.

We described the rule for just one output, but if there were more, a vector of outputs

$$\boldsymbol{y} = (y_1, y_2, \ldots, y_n)$$

then the rule for changing the corresponding synapses $\boldsymbol{w}_k = (w_{1k}, w_{2k}, \ldots, w_{nk})$ would be

$$\Delta \boldsymbol{w}_k = \alpha(\boldsymbol{y}^d - \boldsymbol{y})x_k$$

This rule makes sense in view of the fact that it is similar to the earlier rule in Chapter 3. For each input, x_k, each of its synaptic contacts can be grouped as a vector $\boldsymbol{w}_k$. So what the algorithm does is moves this vector towards the desired pattern along the direction $\boldsymbol{y}^d - \boldsymbol{y}$ as depicted in Figure 5.7.

What has been done is to make the synapses at the final layer of neurons sensitive to error. The example network only has one layer of neurons with adjustable synapses. It turns out that networks that have multiple layers can do much better in reducing error but they need to have a nonlinear activation function. A way to do this that is sympathetic to a neuron's actual input/output properties is to use the function

$$g(u) = frac11 + e^{-u}$$

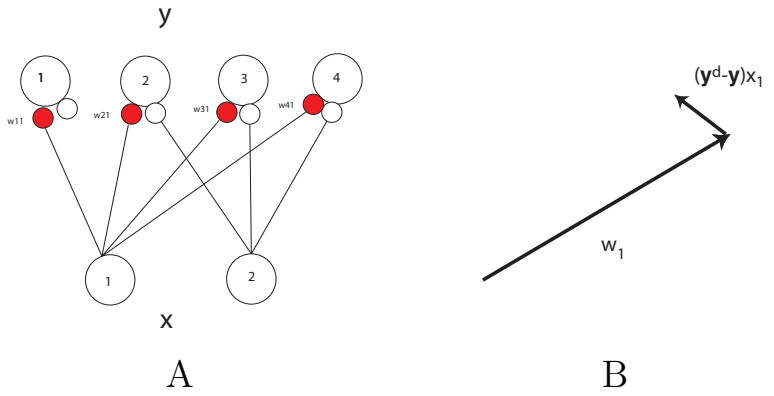

Figure 5.7: A) The input to the neurons representing the output can be grouped such that they can be thought of as a vector. B)The learning rule moves this vector in a direction that will lower $\boldsymbol{y}^d - \boldsymbol{y}$.

This function is zero when u is very negative and one when u is very positive. Thus each output y is now given by

$$y = g(\boldsymbol{w} \cdot \boldsymbol{x})$$

Of course this change will effect the computation of the derivative, but the modification to the equation is straightforward. The main new problem to be dealt with is that now synapses not directly connected to the output layer need to be modified. They can be if the effect of the error is carefully tracked by the right derivatives. Figure 5.8 shows the neurons that contribute to the formula. The error at an internal neuron depends of the errors at the neurons it can effect. Furthermore these errors can be 'backpropagated' to the neuron in question. We won't produce the formulas here but they can be found in [1]. The important point is that a multilayer network's synapses can be programmed by feeding an error signal into its final layer. This fact is going to be very important for the neural backgammon program as the multilayer network is going to take as input the board position and produce as output the value of that position.

5.5 Learning to Play Backgammon

In the Backgammon board game two opponents have 15 checkers each that move in opposite directions along a linear track. Figure 5.9 shows a version of the standard backgammon board. The player with the black checkers must move them clockwise from the position at the lower right toward the position at the upper right, and finally off the board. The player with the white checkers moves

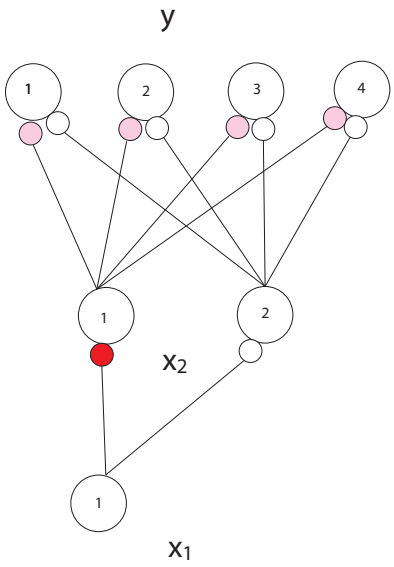

Figure 5.8: .

in the opposite direction. The movement of the checkers is governed by the roll of two dice.

Figure 5.10 shows a possible position in the middle of the game. Player 1, with the black checkers, has just rolled $(6, 3)$. There are several options for playing this move, one of which is shown. Player 2 has a checker on the bar and must attempt to bring it onto the board on the next move. The game ends when all of one player's checkers have been "borne off," or removed from the board. At any time a player may offer to double the stakes of the game. The opposing player must either agree to play for double stakes or forfeit the game. If the player accepts, then that player has the same option at a later time.

The setting is that of a feedforward network that is trying to estimate the value function. In other words, given the set of actions a_t available at time t, the right thing to do is to test them and record the resultant state y. Each of the new states is evaluated in turn using the network, and the one that produces the highest estimate of reward dictates the action that is chosen to move forward in state space. In following this procedure the network learns, for each state y, to produce an estimate of the value function V_t. This estimate in turn can be used to modify the network. The way this modification is achieved is to change the network's output unit weights according to

$$\Delta w = \eta(r_t + \lambda V_{t+1} - V_t) \sum_{k=1}^{k=t} \lambda^{t-k} \frac{\partial V_k}{\partial w}$$

where λ is a parameter that can range between 0 and 1 and η is a learning rate

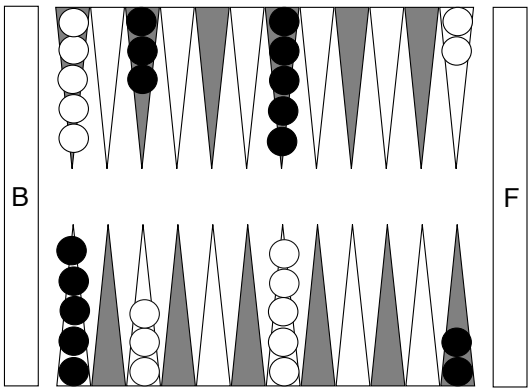

Figure 5.9: The backgammon board in its initial configuration. Opponents have checkers of a given color that can reside on the "points," shown as triangles. Players alternate turns. A turn consists of a roll of the dice and an advancement of checkers according to the dice roll.

parameter that must be handpicked. To understand how this method might work, consider the case for λ very small. In this case only the $k = t$ term counts, and the formula reduces to

$$\Delta w = \eta(r_t + \lambda V_{t+1} - V_t)\frac{\partial V_t}{\partial w}$$

This is just a version of the backpropagation rule where the desired state is V_{t+1}. Thus the network is doing a "smoothing" operation that tries to make sucessive states make consistent predictions. Where there is actual reward, then the actual reward is used instead of the predicted reward. Thus the network works just like Q-learning. Reward centers provide concrete centers that propagate back to more distal states. You can show formally that any value of λ between 0 and 1 works, but that proof is too involved to be presented here.

The complete algorithm is given as Algorithm 5.1.

The backgammon network (Figure 5.11) consists of 96 units for each side to encode the information about the checkers on the board. For each position and color, the units encode whether there are one, two, three, or greater than three checkers present. Six more units encode the number of checkers on the bar and off the board, and the player to move, for a total of 198 units. In addition the network uses hidden units to encode whole-board situations. The number of hidden units used ranges from none to 40.

It is interesting to inspect the features of the network after training, as they reveal the codes that have been selected to evaluate the board. For example, Figure **??** (top) shows large positive weights for black's home board points, large positive weights for white pieces on the bar, positive weights for white blots, and negative weights for white points on black's home board

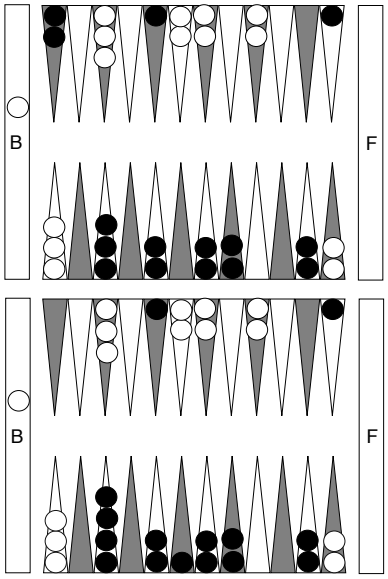

Figure 5.10: (*top*) Player 1, black, has just rolled $(6, 3)$. (*bottom*) There are several options for playing this move, one of which is shown.

These features are all part of a successful black attacking strategy. Figure **??** (bottom) exhibits increasingly negative weights for black blots and black points, as well as a negative weighting for white pieces off and a positive weight for black pieces off. These features would naturally be part of an estimate of black's winning chances.

5.6 Backgammon as a metaphor

We spent a lot of time on the backgammon example but not without purpose. In the movie *The Karate Kid* there is a pivotal moment where the hero realizes that the drudge work he had been doing is actually training for his ultimate goal. With respect to sketching an understanding of brain function, we at at that moment now.

Think of the backgammon problem as a metaphor for any problem the brain has to solve. In the general case visual perception has to come up with an initial coding for the problem. We assume that is done here by choosing the inputs to the network to be just right for capturing the board. Next we need a body to find and roll the dice. Again that happens magically and on cue in our example. Given the dice roll, the different moves - the actions in reinforcement learning parlance - are used to adjust the board and the resultant board is evaluated by the network. The best move is taken and the synapses to the neurons in the network are adjusted by the learning rule.

Algorithm 5.1 Temporal Difference (TD) Algorithm

Initialize the weights in the network to random values.
Repeat the following until the policy converges:

1. Apply the current state to the network and record its output V_t.
2. Make the move suggested by the network, and roll the dice again to get a new state.
3. Apply the new state to the network and record its output V_{t+1}. if the state is a win, substitute $V_{t+1} = 1$; if it's a loss, substitute $V_{t+1} = -1$.
4. Update the weights:

$$\Delta w = \eta(r_t + \lambda V_{t+1} - V_t) \sum_{k=1}^{k=t} \lambda^{t-k} \frac{\partial V_k}{\partial w}$$

5. If there are hidden units, then use the Backpropagation Algorithm to update their weights.

The central idea is that any general problem can be solved the same way. The program that solves it has many possibly different steps and these are represented in terms of a state/action table. This table has the current way of solving the problem represented as a policy. As we take actions recommended by the table we are transported to new states and new parts of the table. New actions are chosen and the process repeats until the goal state is reached meaning that the problem has been solved. The idea is that an everyday task like making tea can be broken up into discrete sensory/motor steps and that these steps can be translated into a state/action code and put in the table.

5.7 Summary and Key Ideas

The last two chapters have sketched the core of the way the brain could use computation to manage behavior. The organization is fundamentally constrained by the slowness of the brain's neural circuitry. This slowness forces the brain to resort to a table look-up strategy where in tried and true behavioral sequences are stored so that they just have to be looked up much in the way we would

find a telephone number in a telephone book. The set of such sequences that are relevant to the current situation can be scored with respect to reward or the expected gain from using them in this context.

So at this point you have a pretty good idea, albeit in broad outline, of how programs in the brain might work:

1. You need to do hundreds of different behaviors in order to survive.
2. Each of these behaviors can be broken down into steps.
3. The part of an individual step that controls the body is represented as a state/action pair.
4. The collection of state/action pairs can be represented as a table.
5. The cortex is the best candidate to be the neural substrate of that table.
6. Actions in the table are scored using neuros.
7. The brain is designed to repeatedly pick possible actions with the highest score.

In this broad outline we have finessed the way neural codes actually get the body to do things, but we will have quite a bit to say about that in the next Chapter. We also have glibly sidestepped issues that might arise in managing all the information in the tables. At any moment, should all the possible actions that could be taken compete to be chosen, or should there be ways of triaging possibilities? This tricky issue will be taken up in Chapter 8.

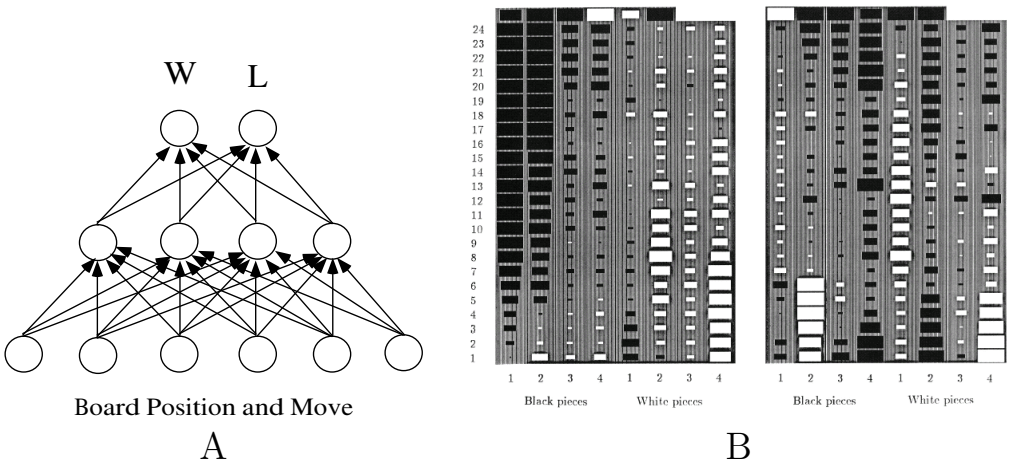

Figure 5.11: A) The architecture of the network used to estimate reward for backgammon. The board is encoded as a 24×8 set of 'point' neurons denoting whether the occupants of a point is black or white as well as whether one, two, three or more than three pieces are on the point. Additional neurons record whether pieces are on the 'bar,' that is, have been sent back to the starting line,white pieces borne off, black pieces borne off, white's turn, and black's turn. B)The weights from two hidden neurons after training. Black squares represent negative values; white squares positive values. The size of the square is proportional to the magnitude of the weight. Columns 1–24 represent the 24 positions on the board. The column at the end represents (from top to bottom): white pieces on the bar, black pieces on the bar, white pieces borne off, black pieces borne off, white's turn, and black's turn. These neurons represent crucial board features used in rating the position. If the actual board does not have pieces where negative synapses are and does have pieces where white (positive synapses are, then that board position is the kind of feature the neurons is looking for. How this feature is weighted in the actual final rating depends on the sign associated with the synapses that connect it to neurons in the final layer.

Bibliography

[1] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 1986.

[2] Wolfram Schultz, Peter Dayan, and P. Read Montague. A neural substrate of prediction and reward. *Science*, 1997.

[3] Gerald Tesauro. Temporal difference learning and td-gammon. *Communications of the ACM*, 1952.