A Connectionist Architecture for Sequential Decision Learning

Yves Chauvin*
Psychology Department
Stanford University
Stanford, CA 94305
chauvin@psych.stanford.edu

Abstract

A connectionist architecture and learning algorithm for sequential decision learning are presented. The architecture provides representations for probabilities and utilities. The learning algorithm provides a mechanism to learn from long-term rewards/utilities while observing information available locally in time. The mechanism is based on gradient ascent on the current estimate of the long-term reward in the weight space defined by a "policy" network. The learning principle can be seen as a generalization of previous methods proposed to implement "policy iteration" mechanisms with connectionist networks. The algorithm is simulated for an "agent" moving in an environment described as a simple one-dimensional random walk. Results show the agent discovers optimal moving strategies in simple cases and learns how to avoid short-term suboptimal rewards in order to maximize long-term rewards in more complex cases.

Introduction

Learning from Long-Term Rewards

If we imagine an agent (machine, animal or human) making decisions and acting in an environment, how can long-term payoffs received from the environment influence present decisions? Similar questions have been raised in different disciplines, including human and animal psychology, machine learning, engineering, robotics and economics (e.g., Bellman, 1957; Sutton & Barto, 1987; Samuel, 1959; Slovic, Lichtenstein & Fishoff, 1988; Watking, 1989). A number of mathematical and numerical tools in the decision sciences have been proposed to answer these questions. In particular, the theory of *dynamic programming* (Bellman, 1957) was specifically developed as an optimization method to solve sequential decision problems.

Recently, Barto, Sutton, and Watkins (In Press) integrated methods from dynamic programming and parameter estimation methods to construct a framework for sequential decision learning. In the same spirit, this paper integrates the classical decision theory framework with learning principles from connectionist theories. Whereas Barto et al. mainly suggested connectionist networks could be used as parametric models to compute an evaluation function, this paper mainly proposes the use of connectionist networks to compute action policies. First, "connectionist representations" for probabilities and utilities in static environments are presented. The framework is then extended to dynamic environments and compared to previous formalisms (e.g., Sutton 1990). Simulations of a simple random walk then show how an agent can maximize the long-term expected utility computed over the decision period.

*also with Net-ID, Inc., Menlo Park, CA.

A Connectionist Framework for Decision Making

The connectionist decision making framework suggested in Chauvin (1991) introduces "connectionist representations" of probabilities and utilities. The final layers of a connectionist network are composed of sets of *e-units*, *p-units* and *u-units*. The *e-units* are exponential units with activations $e_i = e^{\beta s_i}$ where s_i is the input to the *e-units* and β a sensitivity parameter. The *p-units* compute probabilities using the Boltzman distribution:

$$\pi_i = \frac{e_i}{\sum_j^n e_j} = \frac{e^{\beta s_i}}{\sum_j e^{\beta s_j}} \qquad (1)$$

Utilities (considered as monotonic functions of rewards) are represented as "utility weights" u_{ij} from the *p-units* to the linear *u-units*. This set of weights can be given *a priori*, observed from the environment, or estimated during learning. The set of *u-units* then computes an expected utility:

$$U_i = \sum_j^n u_{ij}\pi_j \qquad (2)$$

where i is an index taken over a given set of categories.

In this framework, learning consists in maximizing the expected utility computed by the *u-units*. Back-

propagation principles can be used to compute the gradient of expected utility with respect to the inputs to the *e-units*:

$$\frac{\partial U_t}{\partial s_j} = \beta \pi_j (u_{tj} - U_t) \tag{3}$$

where t represents a target category given an input pattern. Note there is no "error" in this formulation: The algorithm directly maximizes an expected utility. Also note that the *p-units* compute "decision beliefs" π_j representing a decision behavior rather than estimated probabilities of future environmental events. These probabilities represent how the network should classify input patterns to maximize expected utilities. With simple environments, it is possible to show the algorithm converges to optimal behavior. In particular, Bayesian optimality (pure decisions) is obtained when the environment is stochastic. It is also possible to show that, using the Widrow-Hoff procedure, the network parameters (decision and utility weights) can be adapted "on-line" after each observation of the environmental response (Chauvin, 1991).

Sequential Decision Learning

Markovian Decision Hypotheses

Markovian decision problems are defined in terms of a finite set of states X and state transition probabilities p_{xy}. At each time step k, an agent makes a decision a among a set of permissible actions A_x function of the current state x. Depending on the chosen action, the environment will switch from state x to $y \in Y(x,a)$ with probability $p_{xy}(a)$. The set of permissible actions for each state x can be characterized by a probability distribution of actions π_{xa} called a *policy* P. For each state transition, the agent receives a reward/utility (or incurs a cost) u_{xy}.

The goal of the agent is to maximize the long-term expected utility V_i^P from state $x(0) = i$:

$$V_i^P = E^P[\sum_{t=0}^{\infty} \gamma^t u_t | x(0) = i] \tag{4}$$

where $u_t = u_{x_t y_t}$ is the utility received at time t by moving from state x_t to y_t and where γ is a discount factor. The term V_i^P is called the evaluation function of state i given the policy P. For the rest of this paper, we assume that the environment has absorbing states with 0 utilities and set γ to 1.

In the most general case, the learning environment is supposed to be stochastic: $0 < p_{xy}(a) < 1$. For interesting optimized functions, the agent's optimal policy can be shown to be deterministic: $\pi_{xa} \in \{0,1\}$. Dynamic programming approaches to sequential decision problems generally consider *a priori* that the agent's actions have to be deterministic and provide mechanims to maximize V over a set of finite policies. Barto et al. (In Press) and others consider cases where no model of the environment is known *a priori*. Using parameter adaptation methods, they assume the agent's actions can be stochastic and can be continuously adapted as the agent *learns* about its environment.

In this paper, the stochasticity of the agent is assumed *a priori* and is an essential property of the learning method. The environment itself is for now considered as deterministic: There is a one-to-one mapping between actions a and resulting states y. We will see that the prior assumption of agent stochasticity allows us to derive an interesting gradient ascent method on the current estimation of the evaluation function in a "policy" network.

Proposed Formalism

We now extend the "static" connectionist representations for probabilities and utilities to dynamic environments by introducing time and delayed rewards. The agent's total expected utility is now a function of the decision behavior over the complete decision period. Suppose the agent is in state x, for a given policy P, the immediate expected utility can be written as:

$$U_x^P = \sum_a u_{xa} \pi_{xa} \tag{5}$$

where the action probabilities π_{xa} characterize the policy P. From the current state x, suppose the agent can reach a state y by taking action $a \in A$, the long-term expected utility from state x can then be written as:

$$V_x^P = U_x^P + \sum_{a/y} V_y^P \pi_{xa} \tag{6}$$

For a fixed policy P, that is for a fixed set of action probabilities π_{xa}, we could compute V_x^P by "backing up" the state evaluation function one step from V_y^P.

Suppose that at stage k, only an estimation $\hat{V}_y^P(k)$ of V_y^P is available, we can then compute the estimation $\hat{V}_x^P(k+1)$ of V_x^P using:

$$\hat{V}_x^P(k+1) = \sum_{a/y} [u_{xa} + \hat{V}_y^P(k)] \pi_{xa} \tag{7}$$

This process is similar to *value iteration* in dynamic programming. For a given policy P, with a small number of assumptions about the environment and about the order of computations, value iteration will converge to the value V_x^P for each state x.

Equations 7 can be written as:

$$\hat{V}_x^P(k+1) = \sum_a \hat{v}_{xa}(k) \pi_{xa} \tag{8}$$

with $\hat{v}_{xa}(k) = u_{xa} + \hat{V}_y^P(k)$. This equation has the same form as Equation 2. (Note, however, that the indices have different interpretations.) The set of action probabilities π_{xa} can be computed using a set of *e-units* and *p-units* and from a given connectionist representation of each state x. The resulting network can then be called the *policy network*. At each time step k,

the estimate of the long-term expected utility V_x^P can be estimated with one linear *u-unit* where the utility weights u_{ij} of Equation 2 now become $\hat{v}_{xa} = u_{xa} + \hat{V}_y^P$.

Our goal is to find the optimal policy P^* which maximizes the long term expected utility $V_x^{P^*} = V_x^*$:

$$V_x^* = Max_{\pi_{xa}} \sum_{a/y} (u_{xa} + V_y^*)\pi_{xa} \qquad (9)$$

The idea is then to maximize V_x^P by gradient ascent on the current estimate of the evaluation function with respect to the parameters of the policy network. From Equations 3 and 8, for each time step, we can obtain the gradient of $\hat{V}_x^P(k)$ with respect to the inputs s_a of the exponential *e-units*. Simplifying the notation for clarity, we obtain:

$$\begin{aligned} \frac{\partial V_x^P}{\partial s_a} &= \beta \pi_{xa}(v_{xa} - V_x) \\ &= \beta \pi_{xa}(u_{xa} + V_y^P - V_x^P) \end{aligned} \qquad (10)$$

We can now imagine various methods to organize the computations of the evaluation function and of the corresponding policy. A possible *on-line* method is the following. At each time step, an action a is chosen in function of the current action probability distribution implemented by the policy network. From the resulting state y, the current estimation $\hat{V}_y^P(k)$ and the state transitions utilities u_{xa} are used to compute $\hat{V}_x^P(k)$ using Equation 7. The weights of the policy network are then changed by gradient ascent on the current estimate of the evaluation function. Equation 10 computes this gradient with respect to the inputs of the *e-units*. Back-propagation techniques can be used to propagate this gradient further in the policy network as a function of the chosen architecture.

With this organization, the policy is adapted *on-line*, in function of the estimation of the state evaluation function at each time step. Furthermore, the state evaluation updating schedule is itself a function of the current policy. Exploitation by gradient ascent is therefore simultaneous with exploration, determined by the set of action probabilities. The balance of exploitation and exploration may be obtained by tuning the various model parameters and by modifications of the organizations of the computations.

Various connectionist network architectures can be used to compute the action probabilities π_{xa} from the set of possible states. The simplest network consists in having one unit per state and direct connections between states and *e-units*. Such a network architecture can be called "exhaustive" since there is one parameter per state-action pair. Such an exhaustive network is used in the simulations below and is shown in Figure 2. For an exhaustive network, the weight update obtained by gradient ascent can be derived from Equation 10:

$$\Delta w_{ax} = \alpha \beta \pi_{xa}(u_{xa} + V_y^P - V_x^P) \qquad (11)$$

where α is a learning rate. Of course, it might be more interesting to provide state representations and network architectures which are specifically adapted to the environment and to the application. In particular, layers of hidden units might be used to discover compact internal state representations that would be generated by the learning algorithm.

Decision Learning and Parameter Estimation

Equation 7 can be seen as a standard backward dynamic programming technique. It is also related to what Sutton (1988) calls the *Temporal Difference* method. Barto et al. (In Press) point out the relationships between Temporal Difference methods and dynamic programming in more complex situations. They also suggest how an evaluation function V could be estimated using connectionist networks. By contrast, in the framework proposed above, V is actually a table look-up whereas the policy P is implemented with a connectionist network. Of course, we can imagine combinations of evaluation networks and policy networks and various techniques to integrate the connectionist computations of the evaluation functions and corresponding policies.

Barto, Bradtke and Singh (1991) use a variety of algorithms to estimate evaluation functions, also inspired from dynamic programming procedures. In their examples, policies are implemented using a Boltzman distribution:

$$p_{xa} = \frac{e^{\hat{V}_y(k)/T}}{\sum_{z \in Y} e^{\hat{V}_z(k)/T}} \qquad (12)$$

where T acts as a "computational" temperature. In their simulations, the temperature is annealed as a function of the learning performance. The agent's behavior becomes deterministic over the complete set of states simultaneously as the temperature decreases. In the framework proposed above, the level of determinism depends on the weights of the policy network. These weights are updated by gradient ascent on the current estimation of the total expected utility. If from a given state x, the current estimates are identical for all permissible states y, the gradient is null and actions remain equally random for that state. If for a given state, the long-term expected utilities are well differentiated over the set of admissible actions, the gradient descent approach will make the behavior's agent deterministic for that particular state. The agent should learn how and where it should take deterministic decisions, or whether it should keep exploring or not, as a function of the value of this gradient. If reaching a goal provides a high reward for the agent, the agent's behavior will quickly become deterministic when getting close to the goal. The "amount" of determinism will then "move backward" from the goal.

Sutton (1990) also suggests a Boltzman distribution

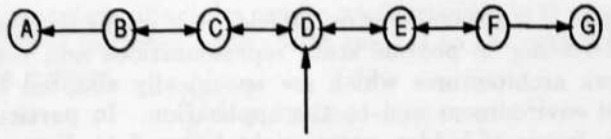

Figure 1: Environment used in the simulations. The starting state is D. The goal of the agent is to reach G.

for action probabilities:

$$p_{xa} = \frac{e^{w_{ax}}}{\sum_{i \in A} e^{w_{ix}}} \tag{13}$$

Furthermore, after each action taken by the agent, the distribution parameters are changed according to:

$$\Delta w_{ax}(t+1) = \alpha(u_{xa} + V_y - V_x) \tag{14}$$

Note the similarities between Equation 14 and Equation 11. The difference in the two weight update equations resides in the action probability π_{xa}. But in Sutton, actions are chosen according to the multinomial probability distribution parametrized by the p_{xa}. Therefore, if we imagine the evaluation function is updated only after a large number of actions have been sampled according to this probability distribution, the weight update becomes $\Delta w_{ax} = \alpha' f_{xa}(u_{xa} + V_y - V_x)$ where f_{xa} represents a frequency of actions and is an unbiased estimate of the "propensity" of action π_{xa}. Therefore, Sutton's weight update equation can be seen as a stochastic form of a gradient ascent on the estimation of the evaluation function in an exhaustive network.

Simulations

Random Walk Environment

The environment used in the simulations is inspired from the random walk process introduced by Sutton (1988). It consists of seven possible states, as shown in Figure 1. The agent's initial position is state D. In each state, the agent has to make a decision about the direction of the following move, left or right. When the agent moves, it might receive a payoff which depends on the current state and on the moving decision. These payoffs may be negative (e.g., they may represent an amount of energy being spent for the move) or positive (e.g., they may represent a received amount of food). These payoffs may then be represented in a two-dimensional *utility* matrix. When the agent reaches the absorbing states A or G, it is put back to the initial state D. The goal of the agent is to maximize the total utility received from the initial state to the goal state G. In the simulations below, we look at the agent's learning behavior as a function of given arbitrary utility matrices.

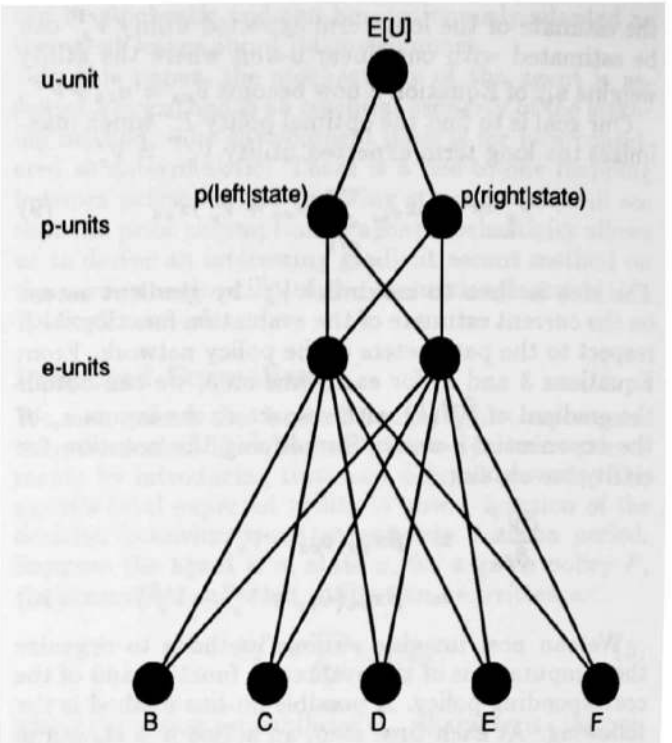

Figure 2: Policy network. The architecture is "exhaustive": there is one parameter per state-action pair.

Policy Network Architecture

The policy network architecture is exhaustive (Figure 2) and can be seen as composed of two subnetworks. The first subnetwork computes move decision probabilities $\pi_{xl} = p(left|state)$ and $\pi_{xr} = p(right|state)$ from each possible state, where states are represented by single binary input units. The second decision subnetwork computes long-term expected utility from decision probabilities and utilities. During learning, the decision weights between states and *e-units* are changed by gradient ascent on the long-term expected utility using the algorithm described above. In this framework, we suppose the agent stores present and estimated future utilities in memory. Although an evaluation network could compute these utilities as needed (Barto et al., In Press), we simply assume for now that learning operates through utilities perfectly retrieved from memory by the agent.

Results

Case 1: Learning from Long-Term Rewards

The approach is first illustrated with the utility matrix shown in Table 1. A simulation *run* is defined as a new set of initial decision weights, representing a new agent. A simulation *trial* is defined as a sequence of moves from the initial state D to the goal G. The network performance can then be judged by the num-

State	B	C	D	E	F
Left move	0	0	0	0	0
Right move	0	0	0	0	10

Table 1: Cost/utility matrix for case 1.

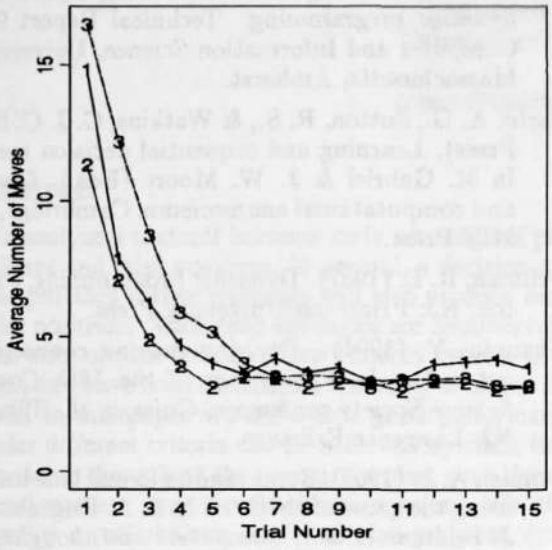

Figure 3: Average number of moves to reach the goal as a function of trial number in cases 1, 2, and 3.

ber of moves (averaged over a given number of runs) it takes for the agent to reach the absorbing state G as a function of trial number. If the agent first reaches absorbing state A, it is put back to the initial state D, incrementing the move counter by 1. Figure 3 shows the agent's performance up to 15 trials averaged over 50 runs. Simulations show the agent always reaches optimal performance (3 steps to the right from state D to G). Absolute performance characteristics obviously depend on the learning parameters (such as the learning rate).

As the network learns how to estimate the correct long-term expected utilities, it also learns how to estimate optimal decision probabilities. This process works "backward in time". At first, the agent reaches goal G from initial state D by chance. In doing so, it observes current payoffs, updates future utility estimation, and adjusts its behavior through the learning mechanism. Eventually, the agent's behavior in state F then converges to optimal behavior. The total expected utility subsequently converges to the optimal value V_F^*. When in state E, the agent reaches state F by chance: the long-term utilities and policy will then be adapted using the estimations obtained for state F. The learning mechanism back-propagates the gradient of long-term expected utility through updated utility weights to modify the decision weights. The utility received from state F to G will make the agent's behavior more deterministic in state F, then backward from F to D. In some sense, both evaluation function *and* policy's determinism are "backed up" from the goal to the initial state.

State	B	C	D	E	F
Left move	-2	-2	-2	-2	-2
Right move	-2	-2	-2	-2	10

Table 2: Cost/utility matrix for case 2.

State	B	C	D	E	F
Left move	-1	-1	+1	-1	-1
Right move	-1	-1	-1	-1	10

Table 3: Cost/utility matrix for case 3.

Of course, the agent's behavior resulting from the implementation of this process might not be as sequential as it sounds. As explained above, evaluation and policies are changed as a function of the current state and of the current decision move, which are stochastic and depend on the organization of the computations. Similarly to on-line policy iteration methods, the agent does not wait to reach the goal to update action probabilities. The agent just looks one step ahead to adjust its "propensities" of action for the current state. For the given utility matrix, because the agent eventually visits the goal and because no other reward may modify the adaptation of behavior, it should always reach optimal behavior.

Case 2: Learning from Long-Term Rewards with Moving Costs

In the second set of simulations, the agent obtains a +10 utility when it reaches the absorbing goal G and a -2 utility when moving from any state to a neighboring state. The corresponding utility matrix is shown in Table 2. Figure 3 shows the agent's learning performance up to 15 trials averaged over 50 runs. With this new utility matrix, the agent reaches optimal performance faster than with the utility matrix used in case 1. This result might not be intuitive since the state expected utilities have now become smaller in reason of the -2 costs "spent" between step moves. The reason for this result is actually that there is now a differential expected utility between going left and going right. For example, the optimal long-term expected utilities V_E^* and V_F^* from states E and F are respectively 8 and 10. This differential expected utility creates a differential utility gradient between each move, forcing the agent to become deterministic earlier and to learn more rapidly how to move in the correct direction.

Case 3: Avoiding Suboptimal Immediate Rewards

One of the motivations for studying sequential decision making and for using dynamic programming methodologies is to avoid suboptimal short-term decisions which may prevent future optimal decisions. The utility matrix shown in Table 3 illustrates this situation. In this case, the agent gets an immediate reward by moving left from the initial state. However, the late reward from state F to G should force the agent to ignore the immediate reward on the left, to move right and to receive the late reward at the goal. When the agent learns about the environment, it will probably be attracted to the immediate reward at first. But by exploration, the agent should learn about the delayed reward and should adjust its behavior over time to ignore the early reward. The short-term reward from D to C should simply delay learning of the optimal strategy. Figure 3 shows the network learning performance for case 3. The learning curve reflects the predicted behavior. At first, it takes more steps to reach the goal because the short-term reward leads the agent in the wrong direction. During early learning, the agent actually learns how to move to the left. However, after sufficient learning, for the given utility matrix, the agent always learns how to avoid short-term rewards and to move directly to the goal. In general, the exact behavior learned by the agent will depend on the balance between exploration and exploitation, which in turn will depend on the model parameters and on the organization of the computations.

Conclusion

A sequential decision learning formalism is proposed which integrates elements of standard decision theory with connectionist principles. In statistical pattern recognition, standard procedures may first estimate model parameters to estimate class probabilities. Costs may then be invoked to compute minimal risk classification. In dynamic environments, dynamic programming techniques, such as *policy iteration* may generated successive evaluate decision strategies and long-term expected rewards until optimal decision behavior is obtained. The present approach directly updates the parameters of a policy network by gradient ascent of the current estimate of the long-term expected utility. The formalism may be seen as a generalization of some of the policy adaptation methods proposed by Sutton (1990) and Barto et al. (1991).

The learning procedure was simulated and tested in a simple environment. In various cases, the procedure was actually shown to generate interesting and intelligent looking learning dynamics. There are many ways the proposed formalism could now be integrated with other dynamic programming concepts or combined with other parameter estimation methods. Of course, it remains to be seen if these learning principles may be powerful enough to generate intelligent decision behavior in more complex environments. But the formalism can be seen as a generalization of previously proposed mechanisms and the gradient ascent approach appears to be conceptually satisfying and promising.

References

Barto, A. G., Bradtke, S. J., & Singh, S. P. (1991). *Real-time learning and control using asynchronous dynamic programming.* Technical Report 91-57, Computer and Information Science, University of Massachusetts, Amherst.

Barto, A. G., Sutton, R. S., & Watkins, C. J. C. H. (In Press). Learning and sequential decision making. In M. Gabriel & J. W. Moore (Eds.), *Learning and computational neuroscience.* Cambridge, MA: MIT Press.

Bellman, R. E. (1957). *Dynamic Programming.* Princeton, NJ: Princeton University Press.

Chauvin, Y. (1991). Decision making connectionist networks. In *Proceedings of the 13th Cognitive Science Society conference, Chicago, IL.* Hillsdale, NJ: Lawrence Erlbaum.

Samuel, A. L. (1959). Some studies in machine learning using the game of checkers. In E. A. Feigenbaum & J. Feldman (Eds.), *Computers and thoughts.* New York: McGraw-Hill.

Slovic, P., Lichtenstein, S., & Fischhoff, B. (1988). Decision making. In R. C. Atkinson, R. J. Herrnstein, G. Lindsey, & R. D. Luce (Eds.), *Steven's handbook of experimental psychology.* New York: Wiley. 2nd. Edition

Sutton, R. S. (1988). Learning to predict by the method of temporal differences. *Machine Learning, 3,* 9–44.

Sutton, R. S. (1990). Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. In *Proceedings of the Seventh International Conference on Machine Learning.* San Mateo, CA: Morgan Kaufmann.

Sutton, R. S. & Barto, A. G. (1987). A temporal-difference model of classical conditioning. In *Proceedings of the 9th Cognitive Science Society conference, Seattle, WA.* Hillsdale, NJ: Lawrence Erlbaum.

Watkins, C. J. C. H. (1989). *Learning from delayed rewards.* PhD thesis, Cambridge University, Cambridge, England.